# Promira Serial Platform – I²C/SPI Active Applications

The Promira Serial Platform with I²C/SPI Active applications allows developers to interface a host PC to a downstream embedded system environment and transfer serial messages using the I²C and SPI protocols.

### Promira Serial Platform – I²C/SPI Active Applications Features

- I²C – Two-wire interface
  - Standard mode (100 kHz)
  - Fast mode (400 kHz)
  - Fast mode Plus (1 MHz)
  - High Speed mode (3.4 MHz)
  - Master and slave functionality
  - Master / Slave Bit Rate 1 kHz to 3.4 MHz
- SPI – Four- to Six-wire Interface
  - Single, Dual, and Quad SPI functionality
  - Master and Slave functionality
  - Master Bit Rate 31 kHz to 80 MHz
  - Slave Bit Rate 31 kHz to 20 MHz
  - Up to eight SS signals
  - Configurable slave select polarity
- GPIO – General Purpose Input/Output
  - Up to sixteen general purpose signals on shared and dedicated pins
  - Selectable polarity
- Integrated Level Shifting
  - Target Power 5V or 3.3V
  - IO Power 0.9V - 3.45V
- Software
  - Windows, Linux, and Mac OS X compatible
  - Easy to integrate application interface
  - Upgradeable Firmware over USB

TOTAL PHASE

Supported products:

PROMIRA

Promira Serial Platform
I²C/SPI Active Applications
User Manual v1.33.003

January 28, 2016

# 1 Revision History

## 1.1 Changes in version 1.33

- Initial revision.
- Fix $I^2C$ 10 bit address issue.
- Update $I^2C$ API examples.
- Update $I^2C$ pull-ups values.

# 2 General Overview

The Promira Serial Platform with I²C/SPI Active applications supports I²C master/slave and Single, Dual, and Quad SPI master/slave modes. The Promira platform with the I²C/SPI Active applications supports up to 8 SPI SS signals and up to 16 GPIO signals depending on purchased application. Control Center Serial supports up to 8 GPIO signals, GPIO00-GPIO07, while the API supports all available signals. The Promira platform connects to an analysis computer via Ethernet or Ethernet over USB. The applications installed on the Promira Serial Platform are field-upgradeable and future-proof.

## 2.1 I²C Background

### 2.1.1 I²C History

When connecting multiple devices to a microcontroller, the address and data lines of each devices were conventionally connected individually. This would take up precious pins on the microcontroller, result in a lot of traces on the PCB, and require more components to connect everything together. This made these systems expensive to produce and susceptible to interference and noise.

To solve this problem, Philips developed Inter-IC bus, or I²C, in the 1980s. I²C is a low-bandwidth, short distance protocol for on board communications. All devices are connected through two wires: serial data (SDA) and serial clock (SCL).

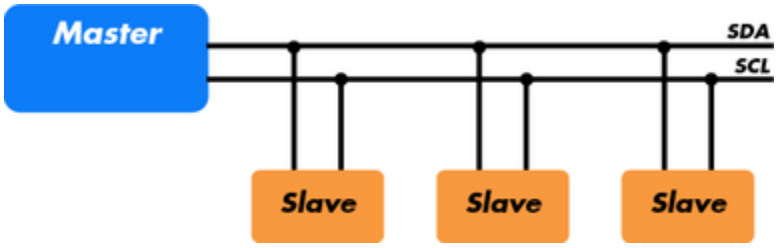


***Figure 1** : Sample I²C Implementation. – Regardless of how many slave units are attached to the I²C bus, there are only two signals connected to all of them. Consequently, there is additional overhead because an addressing mechanism is required for the master device to communicate with a specific slave device.*

Because all communication takes place on only two wires, all devices must have a unique address to identify it on the bus. Slave devices have a predefined address, but the lower bits of the address can be assigned to allow for multiples of the same devices on the bus.

In addition to stand-alone I²C, I²C is the basis of other common communication standards including SMBus and PMBus.

### 2.1.2 I²C Theory of Operation

I²C has a master/slave protocol. The master initiates the communication. Here is a simplified description of the protocol. For precise details, please refer to the Philips I²C specification. The sequence of events are as follows:

1. The master device issues a start condition. This condition informs all the slave devices to listen on the serial data line for their respective address.
2. The master device sends the address of the target slave device and a read/write flag.
3. The slave device with the matching address responds with an acknowledgment signal.
4. Communication proceeds between the master and the slave on the data bus. Both the master and slave can receive or transmit data depending on whether the communication is a read or write. The transmitter sends 8 bits of data to the receiver, which replies with a 1 bit acknowledgment.
5. When the communication is complete, the master issues a stop condition indicating that everything is done.

Figure 2 shows a sample bitstream of the I²C protocol.

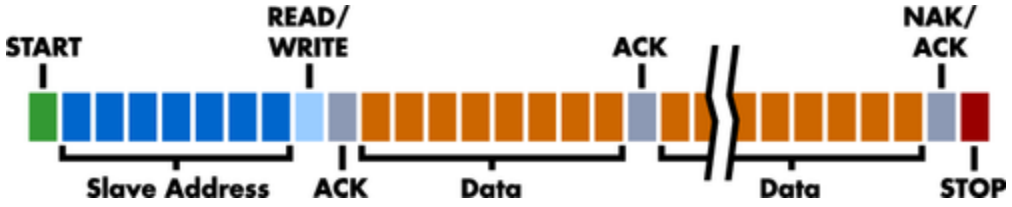


***Figure 2** : I²C Protocol. – Since there are only two wires, this protocol includes the extra overhead of the addressing and acknowledgement mechanisms.*

### 2.1.3 I²C Features

I²C has many features other important features worth mentioning. It supports multiple data speeds: standard (100 kbps), fast (400 kbps), fast mode plus (1000 kbps), and high speed (3.4 Mbps) communications.

Other features include:

- Built-in collision detection,
- 10-bit Addressing,
- Multi-master support,
- Data broadcast (general call).

For more information about other features, see the references at the end of this section.

### 2.1.4 $I^2C$ Benefits and Drawbacks

Since only two wires are required, $I^2C$ is well suited for boards with many devices connected on the bus. This helps reduce the cost and complexity of the circuit as additional devices are added to the system.

Due to the presence of only two wires, there is additional complexity in handling the overhead of addressing and acknowledgments. This can be inefficient in simple configurations and a direct-link interface such as SPI might be preferred.

### 2.1.5 $I^2C$ References

- $I^2C$ – Embedded Systems Academy $I^2C$ Bus Technical Overview and Frequently Asked Questions
- $I^2C$ – NXP $I^2C$ bus specification and user manual

## 2.2 SPI Background

### 2.2.1 SPI Background

The SPI protocol includes single SPI interface, dual SPI interface, and quad SPI interface. This protocol functions on a master-slave paradigm that is ideally suited to data streaming applications.

### 2.2.2 Single SPI interface

Single SPI interface requires four signals: Serial Clock (SCLK), Slave Select (SS), Master Out Slave In (MOSI), and Master In Slave Out (MISO). One SPI master device can be connected to multiple SPI slave devices. The SCLK, MOSI, and MISO signals can be shared by multiple slave devices. However, each slave device has a unique SS signal.

SCLK is generated by the master device and is used for synchronization. SS is pulled low or high by the master in order to select a device for communication. MOSI is used to transfer data from the master device to the slave device. MISO is used to transfer data from the slave device to the master device. Data is always transferred in both directions in SPI, but an SPI device interested in only transmitting data can choose to ignore the receive bytes. Likewise, a device only interested in the incoming bytes can transmit dummy bytes.

The exchange itself has no pre-defined protocol. This makes it ideal for data-streaming applications. Data can be transferred at high speed, often into the range of the tens of megahertz. The flipside is that there is no acknowledgment, no flow control, and the master may not even be aware of the slave's presence.

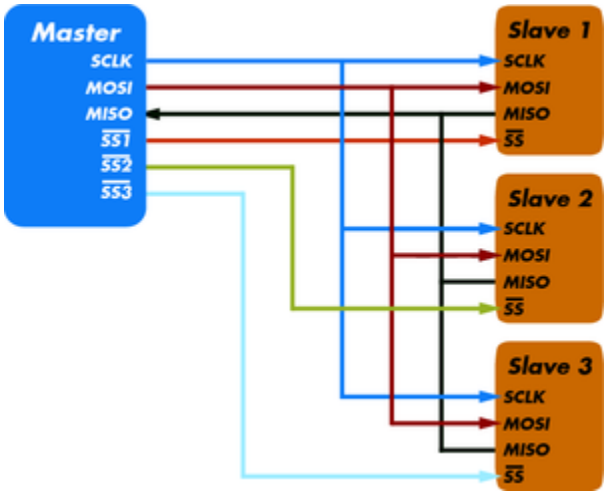


***Figure 3** : Single SPI Interface Block Diagram*

### 2.2.3 Dual SPI interface

Dual SPI interface requires four signals: Serial Clock (SCLK), Slave Select (SS), IO0 and IO1. One SPI master device can be connected to multiple SPI slave devices. The SCLK, IO0 and IO1 signals can be shared by multiple slave devices. However, each slave device has a unique SS signal.

SCLK is generated by the master device and is used for synchronization. SS is pulled low or high by the master in order to select a device for communication. IO1-0 are half-duplex data signals that are used for both transmitting and receiving data for up to twice the performance of single SPI.

Data is sent from the master to the slave as bit pairs on IO0 and IO1. Data is returned from the slave to the master similarly as bit pairs on IO0 and IO1.

### 2.2.4 Quad SPI interface

Quad SPI interface requires six signals: Serial Clock (SCLK), Slave Select (SS), IO0, IO1, IO2, and IO3. One SPI master device can be connected to multiple SPI slave devices. The SCLK and IO0-3 signals can be shared by multiple slave devices. However, each slave device has a unique SS signal.

SCLK is generated by the master device and is used for synchronization. SS is pulled low or high by the master in order to select a device for communication. IO0-3 are half-duplex data signals that are used for both transmitting and receiving data for up to four times the performance of single SPI.

Data is sent from the master to the slave as four bit (nibble) groups on IO0, IO1, IO2, and IO3. Data is returned from the slave to the master similarly as four bit (nibble) groups on IO0, IO1, IO2, and IO3.

### 2.2.5 SPI Modes

The master and slave need to agree about the data frame for the exchange. The data frame is described by two parameters: clock polarity (CPOL) and clock phase (CPHA). Both parameters have two states which results in four possible combinations. These combinations are shown in figure 4.

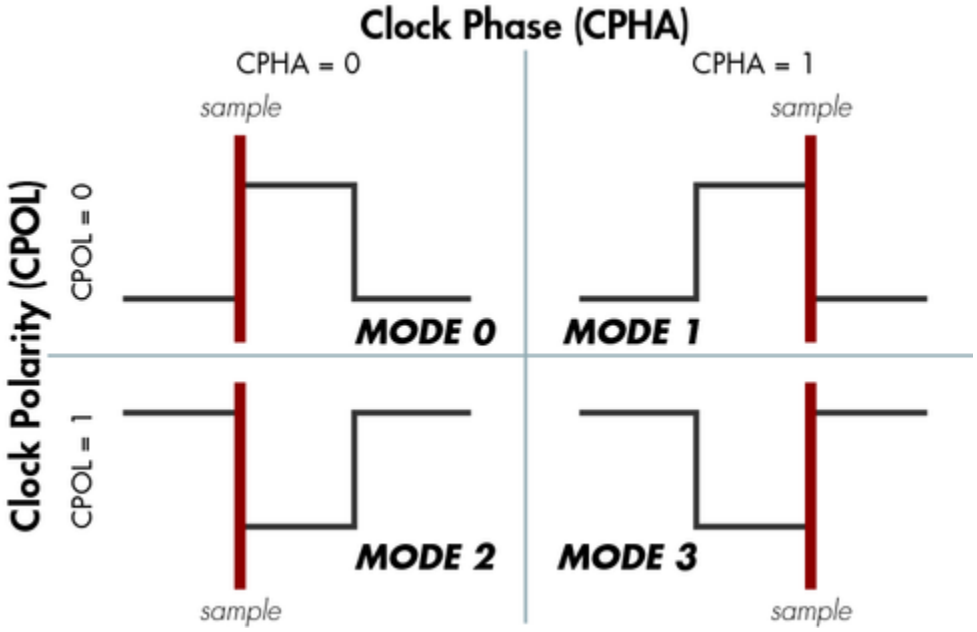

***Figure 4** : SPI Modes –*

### 2.2.6 SPI Clock modes

SPI has two clocks modes: Single Data Rate (SDR) and Double Data Rate (DDR).

In SDR mode the data is latched on one edge, either the rising edge or the falling edge of the clock depend on the SPI mode (CPOL / CPHA). SDR mode supports single SPI, dual SPI and quad SPI interfaces.

In DDR mode the data is latched on both the rising edge and the falling edge of the clock depend on the SPI mode (CPOL / CPHA). DDR mode supports dual SPI and quad SPI interfaces.

### 2.2.7 SPI Benefits and Drawbacks

SPI is a very simple communication protocol. It does not have a specific high-level protocol which means that there is almost no overhead. Data can be shifted at very high rates. This makes it very simple and efficient in a single-master single-slave scenario.

Because each slave needs its own SS, the number of traces required for single SPI is n +3, where n is the number of SPI devices. This means increased board complexity when the number of slaves is increased.

### 2.2.8 SPI References

- SPI – Wikipedia Serial Peripheral Interface Description

# 3 Hardware Specifications

## 3.1 Pinouts

### 3.1.1 Connector Specification

The Promira Serial Platform with I$^2$C/SPI active applications target connector is a standard 2x17 IDC male type connector 0.079x0.079″ (2x2 mm). The Promira platform target connector allows for up to a 34-pin ribbon cable and connector.

Two cables are provided with the Promira platform:

- 34-10 cable: A standard ribbon cable 0.039″ (1 mm) pitch that is 5.12″ (130mm) long with 2x17 IDC female 2x2mm (0.079x0.079) connector and 2x5 IDC female 2.54x2.54mm (0.10x0.10) connector. This provided target ribbon cable will mate with a standard keyed boxed header.
- 34-34 cable: A standard ribbon cable 0.039″ (1 mm) pitch that is 5.12″ (130mm) long with two 2x17 IDC female 2x2mm (0.079x0.079) connectors. This provided target ribbon cable will mate with a standard keyed boxed header.

### 3.1.2 Orientation

The pin order of the 2x5 IDC female connector in the provided target ribbon 34-10 cable is described in figure 5. When looking at the Promira platform front position with the 34-10 ribbon cable (figure 5), pin 1 is in the top left corner and pin 10 is in the bottom right corner.


***Figure 5** : The Promira platform front position with 34-10 cable.*

The pin order of the 2x17 IDC female connector in the provided target ribbon 34-34 cable is described in figure 6. When looking at the Promira platform front position with

the 34-34 ribbon cable (figure 6), pin 1 is in the top left corner and pin 34 is in the bottom right corner.


***Figure 6** : Promira platform front position with 34-34 cable*

### 3.1.3 Pin Description

**Table 1 (1)** : Pin Description - Target Connector

| Pin | Symbol | Description |
|---|---|---|
| 1 | SCL / GPIO00 | $I^2C$ Clock / GPIO signal 00. |
| 3 | SDA / GPIO01 | $I^2C$ Data / GPIO signal 01. |
| 4 | $V_{TGT}$ | Software configurable Vcc target power supply. NC/3.3V/5 V. |
| 5 | MISO / IO1 | SPI Master In Slave Out / Dual/Quad SPI DQ1 |
| 6 | $V_{TGT}$ | Software configurable Vcc target power supply. NC/3.3V/5V. |
| 7 | SCLK | SPI Clock. |
| 8 | MOSI / IO0 | SPI Master Out Slave In / Dual/Quad SPI DQ0 |
| 9 | SS0 / GPIO02 | SPI Slave Select (Chip Select) 0 / GPIO signal 02. |
| 11 | IO2 | Quad SPI DQ2. |
| 13 | IO3 | Quad SPI DQ3. |
| 14 | SS2 / GPIO03 | SPI Slave Select (Chip Select) 2 / GPIO signal 03. |
| 15 | SS1 / GPIO04 | SPI Slave Select (Chip Select) 1 / GPIO signal 04. |
| 17 | SS3 / GPIO05 | SPI Slave Select (Chip Select) 3 / GPIO signal 05. |
| 19 | SS4 / GPIO06 | SPI Slave Select (Chip Select) 4 / GPIO signal 06. |
| 20 | SS5 / GPIO07 | SPI Slave Select (Chip Select) 5 / GPIO signal 07. |
| 21 | GPIO09 | GPIO signal 09. |
| 22 | $V_{IO}$ | Software configurable Vcc IO level power supply. NC/0.9V to 3.45V |

| | | |
|---|---|---|
| 23 | GPIO10 | GPIO signal 10. |
| 24 | $V_{IO}$ | Software configurable Vcc IO level power supply. NC/0.9V to 3.45V. |
| 25 | GPIO11 | GPIO signal 11. |
| 26 | SS6 / GPIO08 | SPI Slave Select (Chip Select) 6 / GPIO signal 08. |
| 27 | GPIO12 | GPIO signal 12. |
| 29 | NC | |
| 31 | GPIO13 | GPIO signal 13. |
| 32 | GPIO14 | GPIO signal 14. |
| 33 | SS7 / GPIO15 | SPI Slave Select (Chip Select) 7 / GPIO signal 15. |
| 2, 10, 12, 16, 18, 28, 30, 34 | GND | Ground Connection. |

Note:

(1) The pin description in this table is for Promira FW v1.30/v1.30 and above. Promira FW v1.30/v1.30 has new features (including higher SPI speed, dual SPI mode and DHCP). Therefore, Promira FW v1.30/v1.30 pinout was modified compare to Promira FW v1.10/v1.10. Specifically, the GPIO and SS signals were changed. For the pinout description for Promira FW v1.10/v1.10, take a look at the Promira manual v1.10/v1.10.

## 3.2 I²C Signaling Characteristics

### 3.2.1 Speed

The Promira Serial Platform I²C master can operate at a maximum bitrate of 3.4 MHz and supports many intermediate bitrates between 1 kHz and 3.4 MHz. The power-on default bitrate for the I²C master unit is 100 kHz.

For slave functionality, the Promira Serial Platform can operate at any rate between 1 kHz and 3.4 MHz.

It is not possible to send bytes at a throughput of exactly 1/8 times the bitrate. The I²C protocol requires that 9 bits are sent for every 8 bits of data. In addition, even though there is no inter-byte delay for the most part of the I²C transaction, the Promira Serial Platform occasionally requires additional time to process the received bytes and set up the next portion of the transaction. In this case, delay is inserted on the I²C bus.

There can be extra overhead introduced by the operating system between calls to the Promira API. These delays will further reduce the overall throughput across multiple

transactions. To achieve the fastest throughput, it is advisable to send as many bytes as possible in a single transaction (i.e., a single call to the Promira API).

### 3.2.2 Pull-up Resistors

There is a pull up resistor on each I²C line (SCL, SDA). The lines are effectively pulled up to 0.9V-3.45V. For HW version v1.7 and above, I²C Standard/Fast/Fast Plus mode, and I²C master mode, the pull up resistor is 2.2K OHM (for 2.2V - 3.45V I²C signal level), 470 OHM (for 1.2V - 2.2V I²C signal level), or 387 OHM (for 0.9V - 1.2V I²C signal level). For additional information take a look at the "Known I²C Limitations" section below. If the Promira Serial Platform is connected to an I²C bus that also includes pull-up resistors, the total pull-up current could be potentially larger. The I²C specification allows for a maximum of 3 mA pull-up current on each I²C line.

A good rule of thumb is that if a downstream I²C device can sink more than 5 mA of current, the protocol should operate properly. Stronger pull-up resistors and larger sink currents may be required for fast bitrates, especially if there is a large amount of capacitance on the bus. The Promira Serial Platform is able to sink approximately 10 mA per pin, so it is possible to have two Promira Serial Platforms communicate with each other as master and slave, with both devices pull-up resistors enabled.

The Promira platform I²C pull up resistors are off by default. Promira platform I²C pull up resistors can be switched and configured through the software GUI and API. Refer to the API section for more details.

### 3.2.3 I²C Clock Stretching

When the Promira Serial Platform is configured as an I²C master, it supports both inter-bit and inter-byte slave clock-stretching. If a slave device pulls SCL low during a transaction, the adapter will wait until SCL has been released before continuing with the transaction.

### 3.2.4 Known I²C Limitations

Promira platform with FW (system/application) v1.10/v1.10 or above works successfully without the delays described in this paragraph. Promira platform with FW v1.01/v1.00 or less I²C master mode occasionally requires additional time to process the received bytes and set up the next bytes. In this case the Promira platform inserts delay on the I²C bus. In this case, every I²C master read transaction will have a delay before the last byte, and there may be additional delays between bytes during I²C master read and write.

Promira FW v1.31/v1.31 in High Speed I²C mode can perform write with no stop that is followed by read. However, it can't perform either read; or write with stop that is followed by read.

The Promira Serial Platform can keep the slave functions enabled even while master operations are executed through the same adapter.

Multi-master is also supported: If there is a bus collision during data transmission and the Promira Serial Platform loses the bus, the transaction will be cut short and the host API will report that fewer bytes were transmitted than the requested number. This condition can be distinguished from the case in which the downstream slave cuts short the transmission by sending a NACK by using the function `ps_i2c_read`.

This constraint can be phrased in a different manner. Say that $I^2C$ master device **A** has a packet length of **X** bytes. If there is a second $I^2C$ master device **B**, that sends packets of length greater than **X** bytes, the first **X** bytes should never contain exactly the same data as the data sent by device **A**. Otherwise the results of the arbitration will be undefined.

This is a constraint found with most $I^2C$ master devices used in a multi-master environment.

When $I^2C$ pull-ups are on, the $I^2C$ pull-ups values, across HW versions, $I^2C$ modes, level shift voltages and $I^2C$ configurations are described in the table below.

For HW version v1.7 and above, $I^2C$ Standard/Fast/Fast Plus mode, and $I^2C$ master mode, when $I^2C$ pull-ups are off, there is no pull up resistor. For all other combinations, when $I^2C$ pull-ups are off, the $I^2C$ pull-ups value is 560 OHM.

**Table 2** : $I^2C$ Pull-ups Values

| HW Version | $I^2C$ Mode | Level Shift Voltage (V) | $I^2C$ Configuration | $I^2C$ Pull-ups Nominal Values (OHM) |
|---|---|---|---|---|
| v1.5 and below | Standard/Fast/Fast Plus/High Speed | 0.9-1.2 | Master/Slave | 229 |
| v1.5 and below | Standard/Fast/Fast Plus/High Speed | 1.2-2.2 | Master/Slave | 255 |
| v1.5 and below | Standard/Fast/Fast Plus/High Speed | 2.2-3.45 | Master/Slave | 446 |
| v1.7 and above | High Speed | 0.9-1.2 | Master/Slave | 229 |
| v1.7 and above | High Speed | 1.2-2.2 | Master/Slave | 255 |
| v1.7 and above | High Speed | 2.2-3.45 | Master/Slave | 446 |
| v1.7 and above | Standard/Fast/Fast Plus | 0.9-1.2 | Master | 387 |
| v1.7 and above | Standard/Fast/Fast Plus | 1.2-2.2 | Master | 470 |
| v1.7 and above | Standard/Fast/Fast Plus | 2.2-3.45 | Master | 2.2K |

| | | | | |
|---|---|---|---|---|
| v1.7 and above | Standard/Fast/Fast Plus | 0.9-1.2 | Slave | 229 |
| v1.7 and above | Standard/Fast/Fast Plus | 1.2-2.2 | Slave | 255 |
| v1.7 and above | Standard/Fast/Fast Plus | 2.2-3.45 | Slave | 446 |

# 3.3 SPI Signaling Characteristics

## 3.3.1 Speeds

The Promira Platform SPI master can operate at bitrates between 31 kHz and 80 MHz. The power-on default bitrate is 1 MHz. The maximum bitrates are only achievable within each individual transaction and does not extend across transactions. The GUI and the OS may add delay due to internal overhead. The Promira platform has also latency, which is caused by the Ethernet / USB link between the computer and the Promira platform. Rarely, there can be also delays across the Ethernet / USB bus even within a transaction.

The Promira Serial Platform SPI slave can operate at any bitrate from 31 kHz up to 20 MHz.

See also section 2.3.3

## 3.3.2 Pin Driving

When the SPI interface is activated as a master, the slave select line (SS) is actively driven low. The MOSI, SCK, IO2, and IO3 lines are driven as appropriate for the SPI mode. After each transmission is complete, these lines are returned to a high impedance state. This feature allows the Promira Serial Platform, following a transaction as a master SPI device, to be then reconnected to another SPI environment as a slave. The Promira Platform will not fight the master lines in the new environment.

It is advisable that every slave also have passive pull-ups on the MOSI and SCK lines. These pull-up resistors can be relatively weak – 100k should be adequate.

As a slave, the MOSI, SCK, and SS lines are configured as an input and the MISO line is configured as an output. This configuration is held as long as the slave mode is enabled (see the API documentation later in the manual).

## 3.3.3 Known SPI Limitations

The implementation uses buffer sizes of 2 MB (less one byte) for send and receive transaction buffers as an SPI master or slave.

# 4 Software

## 4.1 Rosetta Language Bindings: API Integration into Custom Applications

### 4.1.1 Overview

The Promira Rosetta language bindings make integration of the Promira API into custom applications simple. Accessing Promira functionality simply requires function calls to the Promira API. This API is easy to understand, much like the ANSI C library functions, (e.g., there is no unnecessary entanglement with the Windows messaging subsystem like development kits for some other embedded tools).

First, choose the Rosetta bindings appropriate for the programming language. Different Rosetta bindings are included in the software download package available on the Total Phase website. Currently the following languages are supported: C/C++, C#, VB, Python. Next, follow the instructions for each language binding on how to integrate the bindings with your application build setup. As an example, the integration for the C language bindings is described below. (For information on how to integrate the bindings for other languages, please see the example code available for download on the Total Phase website.)

1. Include the `promira.h` and `promact_is.h` files in any C or C++ source module. The module may now use any API call listed in `promira.h` and `promact_is.h`.
2. Compile and link `promira.c` and `promact_is.c` with your application. Ensure that the include path for compilation also lists the directory in which `promira.h` and `promact_is.h` is located if the two files are not placed in the same directory.
3. Place the Promira DLL (promira.dll), included with the API software package, in the same directory as the application executable or in another directory such that it will be found by the previously described search rules.

### 4.1.2 Aardvark Compatibility

The Aardvark Compatibility Rosetta language bindings make it simple to integrate the Aardvark API into a custom application using the Promira Serial Platform. Similar to the Promira language bindings above, follow the instructions for each language binding on

how to integrate the bindings with your application build setup. As an example, the integration for the C language bindings is described below.

1. Include the `aa_pm.h` file included with the API software package in any C or C++ source module. The module may now use any Aardvark API call listed in `aa_pm.h`.
2. Compile and link `aa_pm.c` with your application. Ensure that the include path for compilation also lists the directory in which `aa_pm.h` is located if the two files are not placed in the same directory.
3. Place the Promira DLL (promira.dll) and the Aardvark Compatibility DLL (aa_pm.dll), included with the API software package, in the same directory as the application executable or in another directory such that it will be found by the previously described search rules.

### 4.1.3 Versioning

Since a new Promira DLL and Aardvark Compatibility DLL can be made available to an already compiled application, it is essential to ensure the compatibility of the Rosetta binding used by the application (e.g., `aa_pm.c` ) against the DLL loaded by the system. A system similar to the one employed for the DLL-Firmware cross-validation is used for the binding and DLL compatibility check.

Here is an example.

```
DLL v1.20: compatible with Binding >= v1.10
Binding v1.15: compatible with DLL >= v1.15
```

The above situation will pass the appropriate version checks. The compatibility check is performed within the binding. If there is a version mismatch, the API function will return an error code, `PS_APP_INCOMPATIBLE_LIBRARY`.

### 4.1.4 Customizations

While provided language bindings stubs are fully functional, it is possible to modify the code found within this file according to specific requirements imposed by the application designer.

For example, in the C bindings one can modify the DLL search and loading behavior to conform to a specific paradigm. See the comments in `promira.c` or `aa_pm.c` for more details.

# 5 API Documentation

## 5.1 Introduction

The Promira API documentation that follows is oriented toward the Promira Rosetta C bindings. The set of Promira API functions and their functionality is identical regardless of which Rosetta language binding is utilized. The only differences will be found in the calling convention of the functions. For further information on such differences please refer to the documentation that accompanies each language bindings in the Promira API Software distribution

## 5.2 General Data Types

The following definitions are provided for convenience. All Promira data types are unsigned.

```
typedef unsigned char       u08;
typedef unsigned short      u16;
typedef unsigned int        u32;
typedef unsigned long long  u64;
typedef signed   char       s08;
typedef signed   short      s16;
typedef signed   int        s32;
typedef signed   long long  s64;
typedef float               f32;
```

## 5.3 Notes on Status Codes

Most of the Promira API functions can return a status or error code back to the caller. The complete list of status codes is provided at the end of this chapter. All of the error codes are assigned values less than 0, separating these responses from any numerical values returned by certain API functions.

Each API function can return one of two error codes with regard to the loading of the underlying Promira DLL, `PS_APP_UNABLE_TO_LOAD_LIBRARY` and `PS_APP_INCOMPATIBLE_LIBRARY`. If these status codes are received, refer to the previous sections in this manual that discuss the DLL and API integration of the Promira software. Furthermore, all API calls can potentially return the error `PS_APP_UNABLE_TO_LOAD_FUNCTION`. If this error is encountered, there is likely a serious version incompatibility that was not caught by the automatic version checking system. Where appropriate, compare the language binding versions (e.g., `PM_HEADER_VERSION` found in `promira.h` and `PM_CFILE_VERSION` found in `promira.c` or `PS_APP_HEADER_VERSION` found in `promact_is.h` and

PS_APP_CFILE_VERSION found in promact_is.c) to verify that there are no mismatches. Next, ensure that the Rosetta language binding (e.g., promira.c and promira.h or promact_is.c and promact_is.h) are from the same release as the Promira DLL. If all of these versions are synchronized and there are still problems, please contact Total Phase support for assistance.

Any API function that accepts any type of handle can return the error PS_APP_INVALID_HANDLE if the handle does not correspond to a valid instance that has already been opened or created. If this error is received, check the application code to ensure that the open or create command returned a valid handle and that this handle is not corrupted before being passed to the offending API function.

Finally, any function call that communicates with an Promira device can return the error PS_APP_COMMUNICATION_ERROR. This means that while the handle is valid and the communication channel is open, there was an error receiving the acknowledgment response from the Promira application. This can occur in situations where the incoming data stream has been saturated by asynchronously received messages an outgoing message is sent to the Promira application, but the incoming acknowledgment is dropped by the operating system as a result of the incoming USB receive buffer being full. The error signifies that it was not possible to guarantee that the connected Promira device has processed the host PC request, though it is likely that the requested action has been communicated to the Promira device and the response was simply lost. For example, if the slave functions are enabled and the incoming communication buffer is saturated, an API call to disable the slave may return PS_APP_COMMUNICATION_ERROR even though the slave has actually been disabled.

If either the $I^2C$ or SPI subsystems have been disabled by ps_app_configure, all other API functions that interact with $I^2C$ or SPI will return PS_I2C_NOT_ENABLED or PS_SPI_NOT_ENABLED, respectively.

These common status responses are not reiterated for each function. Only the error codes that are specific to each API function are described below.

All of the possible error codes, along with their values and status strings, are listed following the API documentation.

## 5.4 Application Management Interface

All functions starting with pm_ are for Application Management. Please refer to the Promira Serial general user manual for the details.

# 5.5 General Application Interface

## 5.5.1 General Application

### Overview

After opening the device with `pm_open` and starting an application with `pm_load`, a connection needs to be established with `ps_app_connect`. See the language specific sample programs for examples of this connection process.

### Connect to the Application (ps_app_connect)

```
PromiraConnectionHandle ps_app_connect (const char * net_addr)
```

*Connect to the application launched by pm_load.*

#### Arguments

| | |
|---|---|
| `net_addr` | The net address of the Promira Serial Platform. It could be an IPv4 address or a host name. |

#### Return Value

This function returns a connection handle, which is guaranteed to be greater than zero if valid.

#### Specific Error Codes

| | |
|---|---|
| `PS_APP_UNABLE_TO_OPEN` | Unable to connect to the application. |
| `PS_APP_UNABLE_TO_INIT_SUBSYSTEM` | Failed to initialize one of subsystems ($I^2C$, SPI, or GPIO) in the Promira application. |

#### Details

More than one connection can be made to the application.

### Disconnect to the Application (ps_app_disconnect)

```
int ps_app_disconnect (PromiraConnectionHandle conn)
```

*Disconnect to the application.*

**Arguments**

| | |
|---|---|
| `conn` | handle of the connection to the application |

**Return Value**

The number of the connections to applications disconnected is returned on success. This will usually be 1.

**Specific Error Codes**

`None.`

**Details**

If the `conn` argument is zero, the function will attempt to disconnect all possible handles, thereby disconnecting all connected handles. The total number of handle disconnected is returned by the function.

### Version (ps_app_version)

```
int ps_app_version (PromiraChannelHandle channel,
                    PromiraAppVersion *  version);
```

*Return the version matrix for the application connected to the given handle.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `version` | pointer to pre-allocated structure |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

The `PromiraAppVersion` structure describes the various version dependencies of application components. It can be used to determine which component caused an incompatibility error.

```
struct PromiraAppVersion {
```

```
    /* Software, firmware, and hardware versions. */
    u16 software;
    u16 firmware;
    u16 hardware;

    /* FW requires that SW must be >= this version. */
    u16 sw_req_by_fw;

    /* SW requires that FW must be >= this version. */
    u16 fw_req_by_sw;

    /* API requires that SW must be >= this version. */
    u16 api_req_by_sw;
};
```

If the handle is 0 or invalid, only `software`, `fw_req_by_sw`, and `api_req_by_sw` version are set.

### Sleep (ps_app_sleep_ms)

```
int ps_app_sleep_ms (u32 milliseconds);
```

*Sleep for given amount of time.*

**Arguments**

| | |
|---|---|
| `milliseconds` | number of milliseconds to sleep |

**Return Value**

This function returns the number of milliseconds slept.

**Specific Error Codes**

`None.`

**Details**

This function provides a convenient cross-platform function to sleep the current thread using standard operating system functions.

The accuracy of this function depends on the operating system scheduler. This function will return the number of milliseconds that were actually slept.

#### Status String (ps_app_status_string)

```
const char *ps_app_status_string (int status);
```

*Return the status string for the given status code.*

**Arguments**

`status` status code returned by a Promira application function.

**Return Value**

This function returns a human readable string that corresponds to status. If the code is not valid, it returns a NULL string.

**Specific Error Codes**

`None.`

**Details**

None.

### 5.5.2 Channel

#### Channel Overview

Creating a channel opens a logical pipe to send and receive data with the device.

#### Open a Channel (ps_channel_open)

```
PromiraChannelHandle ps_channel_open (PromiraConnectionHandle conn);
```

*Open a logical communication channel.*

**Arguments**

`conn` handle of the connection to the application

**Return Value**

This function returns a channel handle, which is guaranteed to be greater than zero if valid.

**Specific Error Codes**

None.

**Details**

The channel is a logical communication layer that talks to the application. All commands to the application will be executed through a channel.

### Close the Channel (ps_channel_close)

```
int ps_channel_close (PromiraChannelHandle channel);
```

*Close the logical communication channel with the specified handle.*

**Arguments**

`channel` handle of the channel

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

None.

**Details**

None.

### Get the Number of Queues Submitted (ps_channel_submitted_count)

```
int ps_channel_submitted_count (PromiraChannelHandle channel);
```

*Return the number of queues submitted, but not collected through the channel.*

**Arguments**

`channel` handle of the channel

**Return Value**

The number of queues submitted.

**Specific Error Codes**

None.

**Details**

None.

#### Get the Number of Queues Uncollected (ps_channel_uncollected_count)

```
int ps_channel_uncollected_count (PromiraChannelHandle channel);
```

*Return the number of queues completed and uncollected through the channel.*

**Arguments**

`channel` handle of the channel

**Return Value**

None.

**Specific Error Codes**

None.

**Details**

None.

### 5.5.3 Queue

#### Queue Overview

In order to use the Promira Serial Platform to send data across the bus at high speed, data and commands can be accumulated in a queue until the queue is submitted to shift all of the queued data and commands.

A single queue must contain commands of the same type (i.e. $I^2C$, SPI, or GPIO) and the commands will be executed in order for that subsystem. However, commands across queues with different types will be executed in parallel. For example, if two $I^2C$ queues are submitted followed by one SPI queue, all the $I^2C$ commands will be executed in order and the SPI commands will be executed at the same time as the $I^2C$ commands.

To synchronize data across queues of different types, use `ps_queue_sync` to queue a sync command. When the device receives a sync command, it waits for all previously received commands to execute and then continues. Therefore, in the example above, if

a sync command was submitted between the two I²C queues and the SPI queue, all the I²C commands will complete before the SPI commands begin.

Each queue can have a maximum of 255 commands.

I²C Transfer Example with Promira API Queue Mechanism:

1. Call `ps_queue_create` to create a queue, and specify the queue type (e.g. `PS_MODULE_ID_I2C_ACTIVE`).
2. Call `ps_queue_clear` to clear the queue.
3. Call `ps_queue_i2c_write` to add an I²C write command to the queue.
4. Call `ps_queue_i2c_read` to add an I²C read command to the queue.
5. Call `ps_queue_submit` to send out the accumulated commands on the I²C bus.
6. Call `ps_collect_resp` to collect the next response.
    a. The return value of `ps_collect_resp` will be `PS_I2C_CMD_WRITE`.
    b. Call `ps_collect_i2c_write` to collect the response of the I²C write.
7. Call `ps_collect_resp` to collect the next response.
    a. The return value of `ps_collect_resp` will be `PS_I2C_CMD_READ`.
    b. Call `ps_collect_i2c_read` to collect the response of the I²C read.
8. Repeat step 7 until `ps_collect_resp` returns `PS_APP_NO_MORE_CMDS_TO_COLLECT`.
9. Call `ps_queue_destroy` to destroy the queue or go back to step 6 to submit the queue again.

SPI Transfer Example with Promira API Queue Mechanism:

1. Call `ps_queue_create` to create a queue, and specify the queue type (e.g. `PS_MODULE_ID_SPI_ACTIVE`).
2. Call `ps_queue_clear` to clear the queue.
3. Call `ps_queue_spi_oe` to add a command to the queue to enable SPI outputs.
4. Call `ps_queue_spi_ss` to add a command to the queue to assert SS line.
5. Call `ps_queue_spi_write` or `ps_queue_spi_write_word` or `ps_queue_spi_read` to add an SPI command that's shifts data on the bus.
6. Call `ps_queue_spi_ss` to add a command to the queue to deassert SS line.

7. Call `ps_queue_spi_oe` to add a command to the queue to disable master output.
8. Call `ps_queue_submit` to send the accumulated commands on the SPI bus.
9. Call `ps_collect_resp` to collect the first/next response from the previously submitted queue.
    a. If the return value is `PS_SPI_CMD_READ`, call `ps_collect_spi_read` to collect the response of with the SPI data.
    b. Other wise, no additional functions need to be called.
10. Repeat step 9 until you receive from the function `ps_collect_resp` the return value `PS_APP_NO_MORE_CMDS_TO_COLLECT`.
11. Call `ps_queue_destroy` to destroy the queue or go back to step 6 to submit the queue again.

### Create a Queue (ps_queue_create)

```
PromiraQueueHandle ps_queue_create (
                        PromiraConnectionHandle conn,
                        u08                     queue_type);
```

*Create a queue.*

#### Arguments

| | |
|---|---|
| `conn` | handle of the connection to the application |
| `queue_type` | type of queue. See Table 3 |

**Table 3** : `queue_type` enumerated types

| | |
|---|---|
| `PS_MODULE_ID_I2C_ACTIVE` | An $I^2C$ queue. |
| `PS_MODULE_ID_SPI_ACTIVE` | A SPI queue. |
| `PS_MODULE_ID_GPIO` | A GPIO queue. |

#### Return Value

This function returns a queue handle, which is guaranteed to be greater than zero if valid.

#### Specific Error Codes

`None.`

#### Details

The queue can contain only data and commands of type `queue_type` (i.e. an $I^2C$ queue can only contain $I^2C$ commands).

### Destroy the Queue (ps_queue_destroy)

```
int ps_queue_destroy (PromiraQueueHandle queue);
```

*Destroy the queue.*

**Arguments**

`queue` handle of the queue

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

None.

### Clear the Queue (ps_queue_clear)

```
int ps_queue_clear (PromiraQueueHandle queue);
```

*Clear the queue.*

**Arguments**

`queue` handle of the queue

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

All queued data and commands are removed from the queue.

### Queue a Delay in Milliseconds (ps_queue_delay_ms)

```
int ps_queue_delay_ms (PromiraQueueHandle queue,
                       int                milliseconds);
```

*Queue a delay on the bus in units of milliseconds.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `milliseconds` | amount of time for delay in milliseconds |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

Queue `milliseconds` amount of delay on the bus.

### Queue a Sync Command (ps_queue_sync)

```
int ps_queue_sync (PromiraQueueHandle queue);
```

*Queue a sync command that waits for all previous commands to be executed.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

None.

### Get a number of commands (ps_queue_size)

```
int ps_queue_size (PromiraQueueHandle queue);
```

*Get a number of commands in a queue.*

**Arguments**

| | |
|---|---|
| queue | handle of the queue |

**Return Value**

The number of command is the queue will be returned.

**Specific Error Codes**

None.

**Details**

None.

### Submit the Queue (ps_queue_submit)

```
PromiraCollectHandle ps_queue_submit (
                          PromiraQueueHandle   queue,
                          PromiraChannelHandle channel,
                          u08                  ctrlId,
                          u08 *                queue_type);
```

*Perform the current batch queue.*

**Arguments**

| | |
|---|---|
| queue | handle of the queue |
| channel | handle of the channel |
| ctrlId | index of the subsystem. This argument is for future use. Set 0 for this argument. |
| queue_type | type of queue |

**Return Value**

This function returns a collect handle, which is guaranteed to be greater than zero if valid.

#### Specific Error Codes

None.

#### Details

This function performs all of the accumulated commands in the queue and shifts them in-order to the subsystem ($I^2C$, SPI or GPIO). After the operation completes, the queue is not cleared. Therefore, this function may be called repeatedly if the same sequence of commands is to be shifted across the bus multiple times.

When there are any queues uncollected, this function will return `PS_APP_PENDING_ASYNC_CMD`.

The `queue_type` tells what type of queue commands are executed.

This function blocks and will return when the host receives the response for the first command in the queue. The `ps_collect_resp` function can then be used to retrieve this initial response and the remaining responses.

If `ps_queue_submit` is called again before the previous responses are collected, all uncollected responses of the previous queue will be discarded.

### Submit an Asynchronous Shift (ps_queue_async_submit)

```
int ps_queue_async_submit (PromiraQueueHandle   queue,
                           PromiraChannelHandle channel,
                           u08                  ctrlId);
```

*Submit the shift operations in the queue for asynchronous execution.*

#### Arguments

| | |
|---|---|
| `queue` | handle of the queue |
| `channel` | handle of the channel |
| `ctrlId` | index of the subsystem. This argument is for future use. Set 0 for this argument. |

#### Return Value

A status code is returned with `PS_APP_OK` on success.

#### Specific Error Codes

None.

#### Details

This function will submit the current batch queue asynchronously. A temporary outgoing buffer will be created to store the batch queue. An internal incoming buffer will also be created to asynchronously capture the slave response data. The application programmer does not have to explicitly manage these two buffers. The function will immediately return after queuing this batch onto the Ethernet or the Ethernet over USB rather than waiting for the shift to complete.

At this point, the user application can submit another queue. This can be done immediately by submitting the same queue a second time without altering it the application simply needs to call `ps_queue_async_submit` again. Or, the user application may clear the queue, create a different queue all together or append more commands to the existing queue. Any subsequent calls to `ps_queue_async_submit` will again create a temporary outgoing buffer and copy the current batch into it. Likewise, a temporary incoming buffer will also be created.

Note that the submitted queue should be sufficiently long (in real time) so that it does not complete before the user application can submit more queues (and also collect the first queue). This will allow the adjacent batches to shift with very little delay between them. How long to be safe? First, there is always the possibility that the user applications process could be scheduled out by the operating system before it has an opportunity to submit the subsequent batch. The operating system scheduler time slice may be as much as 10ms. Therefore, submitted batches should be long enough to bridge one, if not two, time slices. Second, if the user application is performing its own functions between the submission of two batches, the length of the batches should be long enough to accommodate the CPU time of those functions.

Keep in mind the overhead for each queue:

1. In the worst case there is an additional 4KB of internal data required for each queue.
2. The incoming data is stored internally before being collected. This incoming data may have up to an additional 4KB of internal data.

Additionally, only a fixed number of submissions can be made and left pending prior to collection. This number is fixed to 127.

Finally, the asynchronous interface is only useful if the outgoing data of any asynchronous submission does not rely on the return data of a previous asynchronous submission.

#### Collect an Asynchronous Submission (ps_queue_async_collect)

```
PromiraCollectHandle ps_queue_async_collect (
                        PromiraChannelHandle channel,
                        u08 *                queue_type);
```

*Collect a previously submitted asynchronous queue.*

**Arguments**

| | |
|---|---|
| `channel` | handle of channel |
| `queue_type` | type of queue |

**Return Value**

This function returns a collect handle, which is guaranteed to be greater than zero if valid.

**Specific Error Codes**

`None.`

**Details**

This function can be called at anytime after submitting a queue for asynchronous processing. It will block until the first command in the pending queue completes.

If `ps_channel_close` is called without collecting pending asynchronous batches, those batches will be canceled, even if they are in progress. All temporary buffers will be freed as well.

### 5.5.4 Collect

#### Collect the Response of the Command (ps_collect_resp)

```
int ps_collect_resp (PromiraCollectHandle collect,
                     int *                length,
                     int *                result,
                     int                  timeout);
```

*Collect the response of one command from a previously submitted asynchronous queue with the associated collect handle.*

**Arguments**

| | |
|---|---|
| `collect` | handle of the collection |
| `length` | The actual number of bytes received |
| `result` | The status code returned when it is executed separately |
| `timeout` | time to wait for the response |

**Return Value**

This function returns the identifier of the response read. See Table 4

**Table 4** : Identifier of the response

| | |
|---|---|
| `PS_I2C_CMD_WRITE` | Response for `ps_queue_i2c_write`.<br>`length` will be 0 and `result` will be $I^2C$ status code (see Table 8). In order to get the number of bytes written, use function `ps_collect_i2c_write`. |
| `PS_I2C_CMD_READ` | Response for `ps_queue_i2c_read`.<br>`length` will be the buffer size in bytes required to get the received data and `result` will be $I^2C$ status code (see Table 8). Use `ps_collect_i2c_read` to get the data. |
| `PS_I2C_CMD_DELAY_MS` | Response for `ps_queue_delay_ms`.<br>`length` and `result` will be the actual delays in milliseconds. |
| `PS_SPI_CMD_OE` | Response for `ps_queue_spi_oe`.<br>`length` will be 0 and `result` will be the actual result of enabling the outputs. |
| `PS_SPI_CMD_SS` | Response for `ps_queue_spi_ss`.<br>`length` will be 0 and `result` will be the bitmask of slave select states. |
| `PS_SPI_CMD_DELAY_CYCLES` | Response for `ps_queue_spi_delay_cycles`.<br>`length` and `result` will be the actual delays in clock cycle. |
| `PS_SPI_CMD_DELAY_MS` | Response for `ps_queue_delay_ms`.<br>`length` and `result` will be the actual delays in milliseconds. |
| `PS_SPI_CMD_DELAY_NS` | Response for `ps_queue_spi_delay_ns`.<br>`length` and `result` will be the actual delays in nanoseconds. |
| `PS_SPI_CMD_READ` | Response for `ps_queue_spi_write`, `ps_queue_spi_write_word`, and `ps_queue_spi_read`.<br>`length` will be the buffer size in bytes required to get the received data and `result` will be the actual number of words. Use `ps_collect_spi_read` to get the data. |
| `PS_GPIO_CMD_DIRECTION` | Response for `ps_queue_gpio_direction`.<br>`length` and `result` will be 0. |

| | |
|---|---|
| PS_GPIO_CMD_GET | Response for ps_queue_gpio_get.<br>length will be 0 and result will be the actual state of the GPIO input lines. |
| PS_GPIO_CMD_SET | Response for ps_queue_gpio_set.<br>length and result will be 0. |
| PS_GPIO_CMD_CHANGE | Response for ps_queue_gpio_change.<br>length will be 0 and result will be the actual state of the GPIO input lines. |
| PS_GPIO_CMD_DELAY_MS | Response for ps_queue_delay_ms.<br>length and result will be the actual delays in milliseconds. |

**Specific Error Codes**

| | |
|---|---|
| PS_APP_NO_MORE_CMDS_TO_COLLECT | Already collect all the responses for the command in the batch |

**Details**

It is also possible to ignore to receive the information come with either `length` or `result` by passing NULL.

For some commands ($I^2C$ write/read or SPI write), additional function call is required to get data and the information.

Once `ps_collect_resp` gets called, the previous response is no longer available.

### 5.5.5 Configuration

**Configure (ps_app_configure)**

```
int ps_app_configure (PromiraChannelHandle channel,
                      int                  config);
```

*Activate/deactivate individual subsystems ($I^2C$, SPI, GPIO).*

**Arguments**

| | |
|---|---|
| channel | handle of the channel |
| config | enumerated type specifying configuration. See Table 5 |

**Table 5** : `config` enumerated types

| | |
|---|---|
| PS_APP_CONFIG_GPIO | Configure pins as GPIO. Disable both I²C and SPI. |
| PS_APP_CONFIG_SPI | Enable SPI. Configure I²C pins as GPIO. |
| PS_APP_CONFIG_I2C | Enable I²C. Configure SPI SS pins as GPIO. |
| PS_APP_CONFIG_SPI\|PS_APP_CONFIG_I2C | Enable both I²C and SPI. |
| PS_APP_CONFIG_QUERY | Queries existing configuration (does not modify). |

**Return Value**

The current configuration will be returned. The configuration will be described by the same values in `config`.

**Specific Error Codes**

`None.`

**Details**

If either the I²C or SPI subsystems have been disabled by this API call, all other API functions that interact with I²C or SPI will return `PS_APP_CONFIG_ERROR`.

If configurations are switched, the subsystem specific parameters will be preserved. For example if the SPI bitrate is set to 500 kHz and the SPI system is disabled and then enabled, the bitrate will remain at 500 kHz. This also holds for other parameters such as the SPI mode, SPI slave response, I²C bitrate, I²C slave response, etc.

However, if a subsystem is shut off, it will be restarted in a quiescent mode. That is to say, the I²C slave function will not be reactivated after re-enabling the I²C subsystem, even if the I²C slave function was active before first disabling the I²C subsystem.

### Target Power (ps_phy_target_power)

```
int ps_phy_target_power (PromiraChannelHandle channel,
                         u08                  power_mask);
```

*Activate/deactivate target power pins 4, 6 and/or 22, 24.*

**Arguments**

`channel` handle of the channel

`power_mask` enumerated values specifying power pin state. See Table 6.

**Table 6** : `power_mask` enumerated types

| | |
|---|---|
| `PS_PHY_TARGET_POWER_NONE` | Disable target power pins 4, 6, 22, 24. Pins 4, 6, 22, 24 at GND level. |
| `PS_PHY_TARGET_POWER_TGT1_5V` | Enable 5V on target power pins 4 and 6. |
| `PS_PHY_TARGET_POWER_TGT1_3V` | Enable 3.3V on target power pins 4 and 6. |
| `PS_PHY_TARGET_POWER_TGT2` | Enable target power pins 22 and 24 with the same voltage as the $I^2C$/SPI signals voltage level as programed by API function `ps_phy_level_shift`. The $I^2C$/SPI logic level can be programed to 0.9V to 3.45V. The precision level of the level shifter is approximately 0.015V. |
| `PS_PHY_TARGET_POWER_BOTH` | Enable 5V on target power pins 4 and 6, and enable target power pins 22 and 24 with the same voltage as the $I^2C$/SPI signals voltage level as programed by API function `ps_phy_level_shift`. |
| `PS_PHY_TARGET_POWER_QUERY` | Queries the target power pin state. |

**Return Value**

The current state of the target power pins will be returned. The configuration will be described by the same values as in the table above.

**Specific Error Codes**

`None.`

**Details**

None.

### Level Shift (ps_phy_level_shift)

```
f32 ps_phy_level_shift (PromiraChannelHandle channel,
                        f32                  level);
```

*Shift the logic level for all signal pins including target power pin 22 and 24.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `level` | logic level from 0.9V to 3.45V |

**Return Value**

The Actual logic level on the Promira host adapter will be returned.

**Specific Error Codes**

`None.`

**Details**

The call with `PS_PHY_LEVEL_SHIFT_QUERY` returns existing configuration and does not modify.

# 5.6 I²C Interface

## 5.6.1 I²C Notes

1. It is not necessary to set the bitrate for the Promira I²C slave.
2. An I²C master operation read or write operation can be transacted while leaving the I²C slave functionality enabled. In a multi-master situation it is possible for the I²C subsystem to lose the bus during the slave addressing portion of the transaction. If the other master that wins the bus subsequently addresses this I²C subsystem slave address, the I²C subsystem will respond appropriately to the request using its slave mode capabilities.
3. It is always advisable to set the slave response before first enabling the slave. This ensures that valid data is sent to any requesting master.
4. It is not possible to receive messages larger than approximately 64 KiB-1 as a slave due to operating system limitations on the asynchronous incoming buffer. As such, one should not queue up more than 64 KiB-1 of total slave data between calls to the Promira API.
5. It is possible for the Promira I²C master to employ some of the advanced features of I²C. This is accomplished by the `PromiraI2CFlags` argument type that is included in the `ps_i2c_read` and `ps_i2c_write` argument lists. The options in Table 7 are available can be logically ORed together to combine them for one operation.

**Table 7** : I²C Advanced Feature Options

| | |
|---|---|
| PS_I2C_NO_FLAGS | Request no options. |
| PS_I2C_10_BIT_ADDR | Request that the provided address is treated as a 10-bit address. The Promira I$^2$C subsystem will follow the Philips I$^2$C specification when transmitting the address. |
| PS_I2C_COMBINED_FMT | Request that the Philips combined format is followed during a I$^2$C read operation. Please see the Philips specification for more details. This flag does not have any effect unless a master read operation is requested and the PS_I2C_10_BIT_ADDR is also set. |
| PS_I2C_NO_STOP | Request that no stop condition is issued on the I$^2$C bus after the transaction completes. It is expected that the PC will follow up with a subsequent transaction at which point a repeated start will be issued on the bus. Eventually an I$^2$C transaction must be issued without the "no stop" option so that a stop condition is issued and the bus is freed. |
| PS_I2C_SIZED_READ | See ps_i2c_read below. |
| PS_I2C_SIZED_READ_EXTRA1 | See ps_i2c_read below. |

6. It is possible for the Promira I$^2$C master to return an extended status code for master read and master write transactions. These codes are described in Table 8 and are returned by the ps_i2c_read and ps_i2c_write functions, as well as the analogous slave API functions.

**Table 8** : I$^2$C Extended Status Code

| | | |
|---|---|---|
| PS_I2C_STATUS_BUS_ERROR | 1 | A bus error has occurred. Transaction was aborted. |
| PS_I2C_STATUS_SLAVE_ACK | 2 | Bus arbitration was lost during master transaction; another master on the bus has successfully addressed this Promira Serial Platforms slave address. As a result, this Promira adapter has automatically switched to slave mode and is responding. |

| | | |
|---|---|---|
| PS_I2C_STATUS_SLAVE_NACK | 3 | The Promira application failed to receive acknowledgment for the requested slave address during a master operation. |
| PS_I2C_STATUS_DATA_NACK | 4 | The last data byte in the transaction was not acknowledged by the slave. |
| PS_I2C_STATUS_ARB_LOST | 5 | Another master device on the bus was accessing the bus simultaneously with this Promira Serial Platform. That device won arbitration of the bus as per the $I^2C$ specification. |
| PS_I2C_STATUS_BUS_LOCKED | 6 | An $I^2C$ packet is in progress, and the time since the last $I^2C$ event executed or received on the bus has exceeded the bus lock timeout. This is most likely due to the clock line of the bus being held low by some other device, or due to the data line held low such that a start condition cannot be executed by the Promira application. The bus lock timeout can be configured using the `ps_i2c_bus_timeout` function. The Promira application resets its own $I^2C$ interface when a timeout is observed and no further action is taken on the bus. |
| PS_I2C_STATUS_LAST_DATA_ACK | 7 | When the $I^2C$ slave is configured with a fixed length transmit buffer, it will detach itself from the $I^2C$ bus after the buffer is fully transmitted. The $I^2C$ slave also expects that the last byte sent from this buffer is NACKed by the opposing master device. This status code is returned by the $I^2C$ slave (see Slave Write Statistics API) if the master device instead ACKs the last byte. The notification can be useful when debugging a third-party master device. |

These codes can provide hints as to why an impartial transaction was executed by the Promira Serial Platform. In the event that a bus error occurs while the Promira Serial Platform is idle and enabled as a slave

(but not currently receiving a message), the adapter will return the bus error through the `ps_i2c_slave_read` function. The length of the message will be 0 bytes but the status code will reflect the bus error.

### 5.6.2 General I²C

#### I²C Pullups (ps_i2c_pullup)

```
int ps_i2c_pullup (PromiraChannelHandle channel,
                   u08                  pullup_mask);
```

*Activate/deactivate I²C pull-up resistors on SCL and SDAFree the I²C subsystem from a held.*

**Arguments**

`channel` handle of the channel

`pullup_mask` enumerated values specifying pullup state. See Table 9.

**Table 9** : `pullup_mask` enumerated types

| | |
|---|---|
| PS_I2C_PULLUP_NONE | Disable SCL/SDA pull-up resistors |
| PS_I2C_PULLUP_BOTH | Enable SCL/SDA pull-up resistors |
| PS_I2C_PULLUP_QUERY | Queries the pull-up resistor state |

**Return Value**

The current state of the I²C pull-up resistors on the Promira platform will be returned. The configuration will be described by the same values as in the table above.

**Specific Error Codes**

None.

**Details**

Both pull-up resistors are controlled together. Independent control is not supported. This function may be performed in any operation mode.

These pull-up resisters vary on the voltage level of SCL/SDA line which can be set by `ps_phy_level_shift`. For HW version v1.7 and above, I²C Standard/Fast/ Fast Plus mode, and I²C master mode, see Table 10. For additional information take a look at the "Known I²C Limitations" section above.

**Table 10** : I²C pull-up resistor

| Level Shift Voltage (V) | I²C Pull-ups Nominal Values (OHM) |
|---|---|
| 0.9-1.2 | 387 |
| 1.2-2.2 | 470 |
| 2.2-3.45 | 2.2K |

### Free bus (ps_i2c_free_bus)

```
int ps_i2c_free_bus (PromiraChannelHandle channel);
```

*Free the I²C subsystem from a held bus condition (e.g., "no stop").*

#### Arguments

`channel` handle of the channel

#### Return Value

A status code is returned with `PS_APP_OK` on success.

#### Specific Error Codes

| | |
|---|---|
| `PS_I2C_BUS_ALREADY_FREE` | The bus was already free and no action was taken. |

#### Details

If the I²C subsystem had executed a master transaction and is holding the bus due to a previous `PS_I2C_NO_STOP` flag, this function will issue the stop command and free the bus. If the bus is already free, it will return the status code `PS_I2C_BUS_ALREADY_FREE`.

Similarly, if the I²C subsystem was placed into slave mode and in the middle of a slave transaction, this command will disconnect the slave from the bus, flush the last transfer, and re-enable the slave. Such a feature is useful if the Promira application was receiving bytes but then was forced to wait indefinitely on the bus because of the absence of the terminating stop command. After disabling the slave, any pending slave reception will be available to the host through the usual `ps_i2c_slave_write_stats` and `ps_i2c_slave_read` API calls.

The bus is always freed (i.e., a stop command is executed if necessary) and the slave functions are disabled at software opening and closing of the device.

### Set Bus Lock Timeout (ps_i2c_bus_timeout)

```
int ps_i2c_bus_timeout (PromiraChannelHandle channel,
                        u16                  timeout_ms);
```

*Set the I²C bus lock timeout in milliseconds.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `timeout_ms` | the requested bus lock timeout in ms. |

#### Return Value

This function returns the actual timeout set.

#### Specific Error Codes

`None.`

#### Details

The power-on default timeout is 200ms. The minimum timeout value is 10ms and the maximum is 450ms. If a timeout value outside this range is passed to the API function, the timeout will be restricted. The exact timeout that is set can vary based on the resolution of the timer within the Promira application. The nominal timeout that was set is returned back by the API function.

If `timeout_ms` is 0, the function will return the bus lock timeout presently set on the Promira application and the bus lock timeout will be left unmodified.

If the bus is locked during the middle of any I²C transaction (master transmit, master receive, slave transmit, slave receive) the appropriate extended API function will return the status code `PS_I2C_STATUS_BUS_LOCKED` as described in the preceding Notes section. The bus lock timeout is measured between events on the I²C bus, where an event is a start condition, the completion of 9bits of data transfer, a repeated start condition, or a stop condition. For example, if a full 9 bits are not completed within the bus lock timeout (due to clock stretching or some other error), the bus lock error will be triggered.

Please note that once the Promira application detects a bus lock timeout, it will abort its I²C interface, even if the timeout condition is seen in the middle of a byte. When

the Promira application is acting as an I²C mater device, this may result in only a partial byte being executed on the bus.

### 5.6.3 I²C Master

#### Set Bitrate (ps_i2c_bitrate)

```
int ps_i2c_bitrate (PromiraChannelHandle channel,
                    int                  bitrate_khz);
```

*Set the I²C bitrate in kilohertz.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `bitrate_khz` | the requested bitrate in khz. |

**Return Value**

This function returns the actual bitrate set.

**Specific Error Codes**

`None.`

**Details**

The power-on default bitrate is 100 kHz.

Only certain discrete bitrates are supported by the I²C master interface. As such, this actual bitrate set will be less than or equal to the requested bitrate.

If `bitrate_khz` is 0, the function will return the bitrate presently set on the I²C subsystem and the bitrate will be left unmodified.

#### Master Read (ps_i2c_read)

```
int ps_i2c_read (PromiraChannelHandle handle,
                 u16                  slave_addr,
                 PromiraI2CFlags      flags,
                 u16                  num_bytes,
                 u08 *                data_in
                 u16 *                num_read);
```

*Read a stream of bytes from the I²C slave device.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `slave_addr` | the slave from which to read |
| `flags` | special operations as described in "Notes" section and below |
| `num_bytes` | the number of bytes to read (maximum 65535) |
| `data_in` | array into which the data read are returned |
| `num_read` | the actual number of bytes read |

**Return Value**

Status code (see "Notes" section).

**Specific Error Codes**

| | |
|---|---|
| `PS_I2C_READ_ERROR` | There was an error reading from the Promira application. This is most likely a result of a communication error. |

**Details**

For ordinary 7-bit addressing, the lower 7 bits of `slave_addr` should correspond to the slave address. The topmost bits are ignored. The $I^2C$ subsystem will assemble the address along with the `R/W` bit after grabbing the bus. For 10-bit addressing, the lower 10 bits of `addr` should correspond to the slave address. The $I^2C$ subsystem will then assemble the address into the proper format as described in the Philips specification, namely by first issuing an write transaction on the bus to specify the 10-bit slave and then a read transaction to read the requested number of bytes. The initial write transaction can be skipped if the "Combined Format" feature is requested in conjunction with the 10-bit addressing functionality.

The `data_in` pointer should be allocated at least as large as `num_bytes`. When the `data_in` is NULL, this function discards the actual received bytes and returns only `num_read`. When the `num_read` is NULL, this function fills the actual received bytes, but doesn't return the number of bytes received.

It is possible to read zero bytes from the slave. In this case, `num_bytes` is set to `0` and the `data_in` argument is ignored (i.e., it can be 0 or point to invalid memory). However, due to the nature of the $I^2C$ protocol, it is not possible to address the slave and not request at least one byte. Therefore, one byte is actually received by the host, but is subsequently thrown away.

If the number of bytes read is zero, the following conditions are possible.

- The requested slave was not found.
- The requested slave is on the bus but refuses to acknowledge its address.

- The I²C subsystem was unable to seize the bus due to the presence of another I²C master. **Here, the arbitration was lost during the slave addressing phase – results can be unpredictable**.
- Zero bytes were requested from a slave. The slave acknowledged its address and returned 1 byte. That byte was dropped.

Ordinarily the number of bytes read, if not `0`, will equal the requested number of bytes. One special scenario in which this will not happen is if the I²C subsystem loses the bus during the data transmission due to the presence of another I²C master.

If the slave has fewer bytes to transmit than the number requested by the master, the slave will simply stop transmitting and the master will receive `0xff` for each remaining byte in the transmission. This behavior is in accordance with the I²C protocol.

Additionally, the `flags` argument can be used to specify a sized read operation. If the flag includes the value `PS_I2C_SIZED_READ`, the I²C subsystem will treat the first byte received from the slave as a packet length field. This length denotes the number of bytes that the slave has available for reading (not including the length byte itself). The I²C subsystem will continue to read the minimum of `num_bytes-1` and the length field. The length value must be greater than `0`. If it is equal to `0`, it will be treated as though it is `1`. In order to support protocols that include an optional checksum byte (e.g., SMBus) the flag can alternatively be set to `PS_I2C_SIZED_READ_EXTRA1`. In this case the I²C subsystem will read one more data byte beyond the number specified by the length field.

The status code allows the user to discover specific events on the I²C bus that would otherwise be transparent given only the number of bytes transacted. The "Notes" section describes the status codes.

For a master read operation, the `PS_I2C_STATUS_DATA_NACK` flag is not used since the acknowledgment of data bytes is predetermined by the master and the I²C specification.

**Queue a Master Read (ps_queue_i2c_read)**

```
int ps_queue_i2c_read (PromiraQueueHandle queue,
                       u16                slave_addr,
                       PromiraI2CFlags    flags,
                       u16                num_bytes);
```

*Queue a command that reads a stream of bytes from the I²C slave device.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `slave_addr` | the slave from which to read |
| `flags` | special operations as described in "Notes" section and below |
| `num_bytes` | the number of bytes to read (maximum 65535) |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

The actual data read and the number of bytes read will be returned with the function `ps_collect_resp` and `ps_collect_i2c_read` when collecting.

### Collect a Master Read (ps_collect_i2c_read)

```
int ps_collect_i2c_read (PromiraCollectHandle collect,
                         u16                  num_bytes,
                         u08 *                data_in
                         u16 *                num_read);
```

*Collect the response of $I^2C$ master read.*

**Arguments**

| | |
|---|---|
| `collect` | handle of the collection |
| `num_bytes` | maximum size of the array |
| `data_in` | array into which the data read are returned |
| `num_read` | the actual number of bytes read |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

| | |
|---|---|
| `PS_APP_MISMATCHED_CMD` | The type of response is not `PS_I2C_CMD_READ`. |

#### Details

This function should be called right after the function `ps_collect_resp` returns `PS_I2C_CMD_READ`. Once the function `ps_collect_resp` is called again, then data for I²C read command will be discarded. However this function can be called many times before the function `ps_collect_resp` is called.

### Master Write (ps_i2c_write)

```
int ps_i2c_write (PromiraChannelHandle channel,
                  u16                  slave_addr,
                  PromiraI2CFlags      flags,
                  u16                  num_bytes,
                  const u08 *          data_out,
                  u16 *                num_written);
```

*Write a stream of bytes to the I²C slave device.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `slave_addr` | the slave from which to write |
| `flags` | special operations as described in "Notes" section |
| `num_bytes` | the number of bytes to write (maximum 65535) |
| `data_out` | pointer to data |
| `num_written` | the actual number of bytes written |

#### Return Value

Status code (see "Notes" section).

#### Specific Error Codes

| | |
|---|---|
| `PS_I2C_WRITE_ERROR` | There was an error reading the acknowledgment from the Promira application. This is most likely a result of a communication error. |

#### Details

For ordinary 7-bit addressing, the lower 7 bits of `slave_addr` should correspond to the slave address. The topmost bits are ignored. The I²C subsystem will assemble the address along with the R/W bit after grabbing the bus. For 10-bit addressing, the lower 10 bits of `addr` should correspond to the slave address. The I²C subsystem will then assemble the address into the proper format as described in the Philips

specification. There is a limitation that a maximum of only 65534 bytes can be written in a single transaction if the 10-bit addressing mode is used.

The `slave_addr 0x00` has been reserved in the I²C protocol specification for general call addressing. I²C slaves that are enabled to respond to a general call will acknowledge this address. The general call is not treated specially in the I²C master. The user of this API can manually assemble the first data byte if the hardware address programming feature with general call is required.

It is actually possible to write `0` bytes to the slave. The slave will be addressed and then the stop condition will be immediately transmitted by the I²C subsystem. No bytes are sent to the slave, so the `data_out` argument is ignored (i.e., it can be 0 or point to invalid memory).

If the number of bytes written is zero, the following conditions are possible.

- The requested slave was not found.
- The requested slave is on the bus but refuses to acknowledge its address.
- The I²C subsystem was unable to seize the bus due to the presence of another I²C master. **Here, the arbitration was lost during the slave addressing phase results can be unpredictable**.
- The slave was addressed and no bytes were written to it because `num_bytes` was set to `0`.

The number of bytes written can be less than the requested number of bytes in the transaction due to the following possibilities.

- The I²C subsystem loses the bus during the data transmission due to the presence of another I²C master.
- The slave refuses the reception of any more bytes.

The status code allows the user to discover specific events on the I²C bus that would otherwise be transparent given only the number of bytes transacted. The "Notes" section describes the status codes.

For a master write operation, the `PS_I2C_STATUS_DATA_NACK` flag can be useful in the following situation:

- Normally the I²C master will write to the slave until the slave issues a NACK or the requested number of bytes have been written.

- If the master has wishes to write 10 bytes, the I²C slave issues either an ACK or NACK on the tenth byte without affecting the total number of bytes transferred. The status code will distinguish the two scenarios. This status information could be useful for further communications with that particular slave device.

### Queue a Master Write (ps_queue_i2c_write)

```
int ps_queue_i2c_write (PromiraQueueHandle queue,
                        u16                slave_addr,
                        PromiraI2CFlags    flags,
                        u16                num_bytes,
                        const u08 *        data_out);
```

*Queue a command that writes a stream of bytes to the I²C slave device.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `slave_addr` | the slave from which to write |
| `flags` | special operations as described in "Notes" section |
| `num_bytes` | the number of bytes to write (maximum 65535) |
| `data_out` | pointer to data |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

The actual data written will be returned with the function `ps_collect_resp` and `ps_collect_i2c_write` when collecting.

### Collect a Master Write (ps_collect_i2c_write)

```
int ps_collect_i2c_read (PromiraCollectHandle collect,
                         u16 *                num_written);
```

*Collect the response of I²C master write.*

**Arguments**

| | |
|---|---|
| `collect` | handle of the collection |
| `num_written` | the actual number of bytes written |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

| | |
|---|---|
| `PS_APP_MISMATCHED_CMD` | The type of response is not `PS_I2C_CMD_READ`. |

**Details**

This function should be called right after the function `ps_collect_resp` returns `PS_I2C_CMD_WRITE`. Once the function `ps_collect_resp` is called again, then data for $I^2C$ write command will be discarded. However this function can be called many times before the function `ps_collect_resp` is called.

### 5.6.4 $I^2C$ Slave

#### Slave Enable (ps_i2c_slave_enable)

```
int ps_i2c_slave_enable (PromiraChannelHandle channel,
                         u08                  addr,
                         u16                  maxTxBytes,
                         u16                  maxRxBytes);
```

*Enable the $I^2C$ subsystem as an $I^2C$ slave device.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `addr` | address of this slave |
| `maxTxBytes` | max number of bytes to transmit per transaction |
| `maxRxBytes` | max number of bytes to receive per transaction |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

The lower 7 bits of `addr` should correspond to the slave address of this I²C subsystem. If the topmost bit of addr is set, the slave will respond to a general call transmission by an I²C master. After having been addressed by a general call, the I²C slave treats the transaction no differently than a single slave communication. There is no support for the hardware address programming feature of the general call that is described in the I²C protocol specification since that capability is not needed for the Promira application.

If `maxTxBytes` is `0`, there is no limit on the number of bytes that this slave will transmit per transaction. If it is non-zero, then the slave will stop transmitting bytes at the specified limit and subsequent bytes received by the master will be `0xff` due to the bus pull-up resistors. The response that is transmitted by the slave is set through the `ps_i2c_slave_set_response` function described below. If the maximum is greater than the response (as set through `i2cc_slave_set_response`) the I²C slave will wrap the response string as many times as necessary to send the requested number of bytes.

If `maxRxBytes` is `0`, the slave can receive an unlimited number of bytes from the master. However, if it is non-zero, the slave will send a not-acknowledge bit after the last byte that it accepts. The master should then release the bus. Even if the master does not stop transmitting, the slave will return the received data back to the host PC and then transition to a idle state, waiting to be addressed in a subsequent transaction.

It is never possible to *restrict* a transmit or receive to 0 bytes. Furthermore, once the slave is addressed by a master read operation it is always guaranteed to transmit at least 1 byte.

If a master transaction is executed after the slave features have been enabled, the slave features will remain enabled after the master transaction completes.

### Slave Disable (ps_i2c_slave_disable)

```
int ps_i2c_slave_disable (PromiraChannelHandle channel);
```

*Disable the I²C subsystem as an I²C slave device.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

#### Specific Error Codes

None.

#### Details

None.

### Slave Set Response (ps_i2c_slave_set_resp)

```
int ps_i2c_slave_set_resp (PromiraChannelHandle channel,
                           u08                  num_bytes,
                           const u08 *          data_out);
```

*Set the slave response in the event the I²C subsystem is put into slave mode and contacted by a master.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `num_bytes` | number of bytes for the slave response |
| `data_out` | pointer to the slave response |

#### Return Value

The number of bytes accepted by the I²C subsystem.

#### Specific Error Codes

None.

#### Details

The value of `num_bytes` must be greater than zero. If it is zero, the response string is undefined until this function is called with the correct parameters.

If more bytes are requested in a transaction, the response string will be wrapped as many times as necessary to complete the transaction.

The buffer space is 256 bytes.

### Asynchronous Polling (ps_i2c_slave_poll)

```
int ps_i2c_slave_poll (PromiraChannelHandle channel,
                       int                  timeout);
```

*Check if there is any asynchronous data pending from the I²C subsystem.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `timeout` | timeout in milliseconds |

**Return Value**

A status code indicating which types of asynchronous messages are available for processing. See Table 11.

**Table 11** : Status code enumerated types

| | |
|---|---|
| `PS_I2C_SLAVE_NO_DATA` | No asynchronous data is available. |
| `PS_I2C_SLAVE_READ` | I²C slave read data is available. Use function `ps_i2c_slave_read` to get data. |
| `PS_I2C_SLAVE_WRITE` | I²C slave write stats are available. Use function `ps_i2c_slave_write_stats` to get data. |
| `PS_I2C_SLAVE_DATA_LOST` | I²C slave data lost stats are available. Use function `ps_i2c_slave_data_lost_stats` to get data. |

**Specific Error Codes**

`None.`

**Details**

Recall that, like all other Promira API functions, this function is not thread-safe.

If the timeout value is negative, the function will block indefinitely until data arrives. If the timeout value is 0, the function will perform a non-blocking check for pending asynchronous data.

This function sends a command to collect all slave data to I²C subsystem and saves it I²C asynchronous slave queue. If there is any slave data in the queue, then it returns the type of first slave data.

One can employ the following technique to guarantee that all pending asynchronous slave data have been captured during each service cycle:

1. Call the polling function with a specified timeout.
2. If the polling function indicates that there is data available, call the appropriate service function once for each type of data that is available.

3. Next, call the polling function with a 0 timeout.
4. Call the appropriate service function once for each type of data that is available.
5. Repeat steps 3 and 4 until the polling function reports that there is no data available.

### Slave Write Statistics (ps_i2c_slave_write_stats)

```
int ps_i2c_slave_write_stats (PromiraChannelHandle channel,
                              u08 *                addr,
                              u16 *                num_written);
```

*Return number of bytes written from a previous Promira $I^2C$ slave to $I^2C$ master transmission.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `addr` | the address to which the sent message was received |
| `num_written` | the number of bytes written by the slave |

#### Return Value

Status code (see "Notes" section).

#### Specific Error Codes

| | |
|---|---|
| `PS_I2C_SLAVE_TIMEOUT` | There was no recent slave transmission. |
| `PS_I2C_SLAVE_READ_ERROR` | This slave data is not $I^2C$ slave write. |

#### Details

The transmission of bytes from the Promira slave, when it is configured as an $I^2C$ slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous write transactions.

The only possible status code is `PS_I2C_STATUS_BUS_ERROR` which can occur when an illegal START, STOP, or RESTART condition appears on the bus during a transaction. In this case the `num_written` may not exactly reflect the number of bytes written by the slave. It can be off by 1.

### Slave Read (ps_i2c_slave_read)

```
int ps_i2c_slave_read (PromiraChannelHandle channel,
                       u08 *                addr,
```

```
                                u16                 num_bytes,
                                u08 *               data_in
                                u16 *               num_read);
```

*Read the bytes from an I²C slave reception.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `addr` | the address to which the received message was sent |
| `num_bytes` | the maximum size of the data buffer |
| `data_in` | array into which the data read are returned |
| `num_read` | the actual number of bytes read by the slave |

**Return Value**

Status code (see "Notes" section).

**Specific Error Codes**

| | |
|---|---|
| `PS_I2C_SLAVE_TIMEOUT` | There was no recent slave transmission. |
| `PS_I2C_DROPPED_EXCESS_BYTES` | The msg was larger than num_bytes. |
| `PS_I2C_SLAVE_READ_ERROR` | This slave data is not I²C slave read. |

**Details**

If the message was directed to this specific slave, `*addr` will be set to the value of this slaves address. However, this slave may have received this message through a general call addressing. In this case, `*addr` will be `0x80` instead of its own address.

The num_bytes parameter specifies the size of the memory pointed to by data. It is possible, however, that the received slave message exceeds this length. In such a situation, `PS_PS_I2C_DROPPED_EXCESS_BYTES` is returned, meaning that `num_bytes` was placed into data but the remaining bytes were discarded

There is no cause for alarm if the number of bytes read is less than `num_bytes`. This simply indicates that the incoming message was short.

The reception of bytes by the Promira slave, when it is configured as an I²C slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous transactions.

The only possible status code is `PS_I2C_STATUS_BUS_ERROR` which can occur when an illegal START, STOP, or RESTART condition appears on the bus during a transaction.

#### Slave Data Lost Statistics (ps_i2c_slave_data_lost_stats)

```
int ps_i2c_slave_data_lost_stats (PromiraChannelHandle channel);
```

*Return number of slave read/write lost from a previous Promira $I^2C$ slave to $I^2C$ master transmission.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |

**Return Value**

The function returns the number of $I^2C$ slave read/write

**Specific Error Codes**

| | |
|---|---|
| `PS_I2C_SLAVE_TIMEOUT` | There was no recent slave transmission. |
| `PS_I2C_SLAVE_READ_ERROR` | This slave data is not $I^2C$ slave data lost. |

**Details**

There are two asynchronous slave queues, one in the host and the other is in the device. When the capacity of both queues is all 255. If the number of slave data exceeds 255 in the device, $I^2C$ slave read/write is counted as lost and returns back to the host.

## 5.7 SPI Interface

### 5.7.1 SPI Notes

1. The SPI master and slave must both be configured to use the same bit protocol (mode).
2. It is not necessary to set the bitrate for the Promira SPI slave.
3. It is always advisable to set the slave response before first enabling the slave. This ensures that valid data is sent to any requesting master.
4. The maximum amount of outstanding slave data to collect is 2 MB-1. It is advisable to collect the SPI slave data as soon as possible to not lose data.
5. The maximum data size of single command is 1MB. The maximum amount of data in a queue is 64MB-1.

6. The master and slave functionality cannot be used simultaneously. This means the slave must be disabled before calling any master related commands.

### 5.7.2 General SPI

#### Configure (ps_spi_configure)

```
int ps_spi_configure (PromiraChannelHandle channel,
                      PromiraSpiMode       mode,
                      PromiraSpiBitorder   bitorder,
                      u08                  ss_polarity);
```

*Configure the SPI master or slave interface.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `mode` | `PS_SPI_MODE_0`, `PS_SPI_MODE_1`, `PS_SPI_MODE_2` or `PS_SPI_MODE_3` |
| `bitorder` | `PS_SPI_BITORDER_MSB` or `PS_SPI_BITORDER_LSB` |
| `ss_polarity` | bitmask of the polarity of the slave select signals |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

These configuration parameters specify how to clock the bits that are sent and received on the Promira SPI interface.

The mode option configures the SPI mode. See the figure found in the "SPI Background" chapter for more details.

The bitorder option is used to indicate whether LSB or MSB is shifted first.

The ss_polarity option is a bitmask that indicates whether each SS line is active high or active low. For example, setting ss_polarity to 0x05 would mean that SS3 and SS1 are active high and all others are active low.

### Configure Delays (ps_spi_configure_delays)

```
int ps_spi_configure_delays (PromiraChannelHandle channel,
                             u08                  word_delay);
```

*Configure the delays.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `word_delay` | The number of clock cycles between data words. |

#### Return Value

A status code is returned with `PS_APP_OK` on success.

#### Specific Error Codes

`None.`

#### Details

The word_delay parameter is a user-definable delay between words. It can be 0 or greater than 1 and no gap is included after the last word.

### Enable SS Lines (ps_spi_enable_ss)

```
int ps_spi_enable_ss (PromiraChannelHandle channel,
                      u08                  ss_enable);
```

*Enable select SS lines and disable GPIO lines.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `ss_enable` | A bitmask based on the 8 SS lines where 1 corresponds to enable and 0 to disable. |

#### Return Value

A status code is returned with `PS_APP_OK` on success.

#### Specific Error Codes

None.

#### Details

ss_enable is to enable which pins are configured to ss line instead of GPIO. The least significant bit is SS0.

### Enable Master Outputs (ps_queue_spi_oe)

```
int ps_queue_spi_oe (PromiraQueueHandle queue,
                     u08                oe);
```

*Enable/disable the outputs.*

#### Arguments

| | |
|---|---|
| queue | handle of the queue |
| oe | 0 to disable the outputs, and 1 to enable |

#### Return Value

A status code is returned with PS_APP_OK on success.

#### Specific Error Codes

None.

#### Details

This function enables and disables the outputs on the Promira.

### Queue Slave Select Signals (ps_queue_spi_ss)

```
int ps_queue_spi_ss (PromiraQueueHandle queue,
                     u08                ss_assert);
```

*Queue the assertion and de-assertion of slave select signals.*

#### Arguments

| | |
|---|---|
| queue | handle of the queue |
| ss_assert | bitmask where 1 asserts the slave select and 0 de-asserts |

#### Return Value

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

The least significant bit is SS0.

The outputs should be enabled using `ps_queue_spi_oe` before calling this function.

### 5.7.3 SPI Master

**Set Bitrate (ps_spi_bitrate)**

```
int ps_spi_bitrate (PromiraChannelHandle channel,
                    int                  bitrate_khz);
```

*Set the SPI bitrate in kilohertz.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `bitrate_khz` | the requested bitrate in khz |

**Return Value**

This function returns the actual bitrate set.

**Specific Error Codes**

None.

**Details**

The power-on default bitrate is 1000 kHz (1 MHz).

Only certain discrete bitrates are supported by the SPI subsystem. As such, this actual bitrate set will be less than or equal to the requested bitrate unless the

requested value is less than 31 kHz, in which case the SPI subsystem will default to 31 kHz.

If `bitrate_khz` is 0, the function will return the bitrate presently set on the Promira application and the bitrate will be left unmodified.

### Queue a Delay in Cycles (ps_queue_spi_delay_cycles)

```
int ps_queue_spi_delay_cycles (PromiraQueueHandle queue,
                               u32                cycles);
```

*Queue a delay value on the bus in units of clock cycles.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `cycles` | cycles of delay to add to the outbound shift |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

Queues cycles amount of delay on the bus. These are in units of clock cycles as set with `ps_spi_bitrate`.

Actual number of cycles queued is returned when collecting a response using `ps_collect_resp`.

### Queue a Delay in Nanoseconds (ps_queue_spi_delay_ns)

```
int ps_queue_spi_delay_ns (PromiraQueueHandle queue,
                           u32                nanoseconds);
```

*Queue a delay value on the bus in units of nanoseconds.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `nanoseconds` | amount of time for delay in nanoseconds |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

Queues nanoseconds amount of delay on the bus. The fundamental unit of delay that can be queued on the Promira SPI bus is the clock period. Therefore, requested delay will be rounded up to this time span.

The requested number of nanoseconds must be greater than zero and less than or equal to 2 seconds. If the requested number of nanoseconds is out of bounds, no delay is queued.

The actual number of nanoseconds queued is returned when collecting a response using ps_collect_resp.

### Queue SPI Master Write (ps_queue_spi_write)

```
int ps_queue_spi_write (PromiraQueueHandle queue,
                        PromiraSpiIOMode   io,
                        u08                word_size,
                        u32                out_num_words,
                        const u08 *        data_out);
```

*Queue a command that writes a stream of words to the downstream SPI slave device.*

**Arguments**

| | |
|---|---|
| queue | handle of the queue |
| io | IO mode flag as defined in table 12 |
| word_size | number of bits for a word; between 2 and 32 |
| out_num_words | number of words to send |
| data_out | pointer to the array of words to send |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

The outputs should be enabled using `ps_queue_spi_oe` before calling this function.

`data_out` is a buffer containing a bitwise concatenation of words to be sent out. For instance, when word size is 4 and words are 0x1 0x2 0x3 0x4 0x5, then data in a buffer looks like 0x12, 0x34, 0x50. The size of `data_out` should be equal to or bigger than (`word_size` * `out_num_words` + 7) / 8.

The actual data read and the number of words read will be returned with the function `ps_collect_resp` and `ps_collect_spi_read` when collecting.

**Table 12** : SPI IO Modes

| | |
|---|---|
| `PS_SPI_IO_STANDARD` | Standard, full-duplux SPI. |
| `PS_SPI_IO_DUAL` | Dual mode SPI. |
| `PS_SPI_IO_QUAD` | Quad mode SPI. |

### Queue SPI Master Write Word (ps_queue_spi_write_word)

```
int ps_queue_spi_write_word (
                        PromiraQueueHandle queue,
                        PromiraSpiIOMode   io,
                        u08                word_size,
                        u32                out_num_words,
                        u32                word);
```

*Queue a command that writes a stream of the same word to the downstream SPI slave device*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `io` | IO mode flag as defined in table 12 |
| `word_size` | number of bits for a word; between 2 and 32 |
| `out_num_words` | number of words to send |
| `word` | value of the word to queue |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

Queues out_num_words number of words to send and sets each word to the value of word.

The outputs should be enabled using ps_queue_spi_oe before calling this function.

### Queue SPI Master Read (ps_queue_spi_read)

```
int ps_queue_spi_read (PromiraQueueHandle queue,
                       PromiraSpiIOMode   io,
                       u08                word_size,
                       u32                in_num_words);
```

*Queue a command that performs an SPI read operation.*

**Arguments**

| | |
|---|---|
| queue | handle of the queue |
| io | IO mode flag as defined in table 12 |
| word_size | number of bits for a word; between 2 and 32 |
| in_num_words | number of words to clock in |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

This function queues the command, it will be executed when the function ps_queue_submit or ps_queue_async_submit is called.

The outputs should be enabled using ps_queue_spi_oe before calling this function.

When `io` is `PS_SPI_IO_STANDARD`, this function is equivalent to `ps_queue_spi_write_word` with `word` equal to `0`. When `io` is `PS_SPI_IO_DUAL` or `PS_SPI_IO_QUAD`, the clock is generated and the data lines are set to inputs.

The actual data read and the number of words read will be returned with the function `ps_collect_resp` and `ps_collect_spi_read` when collecting.

### Collect a Master Write/Read (ps_collect_spi_read)

```
int ps_collect_spi_read (PromiraCollectHandle collect,
                         u08 *                word_size,
                         u32                  in_num_bytes,
                         u08 *                data_in);
```

*Collect the response of SPI master read.*

**Arguments**

| | |
|---|---|
| `collect` | handle of the collection |
| `word_size` | number of bits for a word received |
| `in_num_bytes` | number of bytes to receive |
| `data_in` | array into which the data read are returned |

**Return Value**

This function returns the total number of bytes read from the slave.

**Specific Error Codes**

| | |
|---|---|
| `PS_APP_MISMATCHED_CMD` | The type of response is not `PS_SPI_CMD_READ`. |

**Details**

This function should be called right after the function `ps_collect_resp` returns `PS_SPI_CMD_READ`. Once the function `ps_collect_resp` is called again, then

data received will be discarded. However this function can be called many times before the function `ps_collect_resp` is called.

`data_in` is returned with data containing a bitwise concatenation of words received. For instance, when word size is 4 and words received are 0x1 0x2 0x3 0x4 0x5, then data returned looks like 0x12, 0x34, 0x50.

### 5.7.4 SPI Slave

#### Slave Enable (ps_spi_slave_enable)

```
int ps_spi_slave_enable (PromiraChannelHandle channel,
                         PromiraSlaveMode     mode);
```

*Enable the SPI subsystem as an SPI slave device.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `mode` | slave mode |

**Table 13** : SPI Slave Modes

| | |
|---|---|
| `PS_SPI_SLAVE_MODE_STD` | Basic slave capabilities |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

None.

#### Slave Disable (ps_spi_slave_disable)

```
int ps_spi_slave_disable (PromiraChannelHandle channel);
```

*Disable the SPI subsystem as an SPI slave device.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

None.

### Configure SPI Slave (ps_spi_std_slave_configure)

```
int ps_spi_std_slave_configure(
                          PromiraChannelHandle channel,
                          PromiraSpiMode       io,
                          u08                  flags);
```

*Configure the SPI slave parameters.*

**Arguments**

| | |
|---|---|
| channel | handle of the channel |
| io | IO mode flag as defined in table 12 |
| flags | flags as defined in table 14 |

**Table 14** : SPI Slave Flags

| | |
|---|---|
| PS_SPI_SLAVE_NO_SS | Use the slave timeout instead of the SS line |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

When the PS_SPI_SLAVE_NO_SS flag is set, the timeout configured with ps_spi_slave_timeout is used to define SPI transaction boundaries instead of the slave select signal.

### Set SPI Slave Timeout (ps_spi_slave_timeout)

```
int ps_spi_slave_timeout (PromiraChannelHandle channel,
                          u32                  timeout_ns);
```

*Set the SPI slave timeout in nanoseconds.*

**Arguments**

| | |
|---|---|
| channel | handle of the channel |
| timeout_ns | SPI transaction timeout in nanoseconds (minimum of 1000 ns) |

**Return Value**

A status code is returned with PS_APP_OK on success.

**Specific Error Codes**

None.

**Details**

When the PS_SPI_SLAVE_NO_SS flag is set in ps_spi_std_slave_configure, this timeout is used to define SPI transaction boundaries instead of the slave select signal.

### Set SPI Slave Host Read Size (ps_spi_slave_host_read_size)

```
int ps_spi_slave_host_read_size (
                        PromiraChannelHandle channel,
                        u32                  read_size);
```

*Set the SPI slave host read size.*

**Arguments**

| | |
|---|---|
| channel | handle of the channel |
| read_size | amount of slave data to collect before sending to host |

**Return Value**

A status code is returned with PS_APP_OK on success.

#### Specific Error Codes

None.

#### Details

Instead of waiting for an entire SPI transaction to complete before sending the data to the PC, this function sets a limit for the amount of data to collect before sending it back to the PC.

For example, say the host read size is set to 64KB. If the Promira receives a large transaction of 1MB from the master, the user will receive the 1MB worth of data in 64KB chunks when calling the `ps_spi_slave_read` function.

When the last chunk is received, `ps_spi_slave_read` will set the `is_last` flag in `read_info`.

### Slave Set Response (ps_spi_std_slave_set_resp)

```
int ps_spi_std_slave_set_resp (PromiraChannelHandle channel,
                               u16                  num_bytes,
                               const u08 *          data_out);
```

*Set the slave response in the event the SPI subsystem is put into slave mode and contacted by a master.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `num_bytes` | number of bytes for the slave response |
| `data_out` | pointer to the slave response |

#### Return Value

The number of bytes accepted by the SPI subsystem for the response.

#### Specific Error Codes

None.

#### Details

The value of `num_bytes` must be greater than zero. If it is zero, the response string is undefined until this function is called with the correct parameters.

Due to limited buffer space on the SPI subsystem, the device may only accept a portion of the intended response. If the value returned by this function is less than `num_bytes` the SPI subsystem has dropped the remainder of the bytes.

If more bytes are requested in a transaction, the response string will be wrapped as many times as necessary to complete the transaction.

The buffer space will nominally be 256 bytes.

### Asynchronous Polling (ps_spi_slave_poll)

```
int ps_spi_slave_poll (PromiraChannelHandle channel,
                       int                  timeout);
```

*Check if there is any asynchronous slave data pending from the SPI subsystem.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `timeout` | timeout in milliseconds |

**Return Value**

A status code indicating which types of asynchronous messages are available for processing. See Table 11.

**Table 11** : Status code enumerated types

| | |
|---|---|
| `PS_SPI_SLAVE_NO_DATA` | No asynchronous slave data is available. |
| `PS_SPI_SLAVE_DATA` | SPI slave read data is available. Use `ps_spi_slave_read` to get data. |
| `PS_SPI_SLAVE_DATA_LOST` | SPI slave data lost stats are available. Use `ps_spi_slave_data_lost_stats` to get data. |

**Specific Error Codes**

`None.`

**Details**

This function is like the function `ps_i2c_slave_poll`. However the SPI slave data is separately handled and saved in the SPI asynchronous queue.

### Slave Read (ps_spi_slave_read)

```
int ps_spi_slave_read (PromiraChannelHandle      channel,
                       PromiraSpiSlaveReadInfo * read_info,
                       u32                       in_num_bytes,
                       u08 *                     data_in);
```

*Read the bytes from an SPI slave reception.*

#### Arguments

| | |
|---|---|
| `channel` | handle of the channel |
| `read_info` | index of the subsystem |
| `in_num_bytes` | the maximum size of the data_in buffer |
| `data_in` | array into which the data read are returned |

#### Return Value

This function returns the number of bytes read asynchronously.

#### Specific Error Codes

| | |
|---|---|
| `PS_SPI_SLAVE_TIMEOUT` | There was no recent slave transmission. |
| `PS_SPI_DROPPED_EXCESS_BYTES` | The data was larger than num_bytes. |
| `PS_SPI_SLAVE_READ_ERROR` | The slave data is not SPI slave data lost. |

#### Details

The `in_num_bytes` parameter specifies the size of the memory pointed to by data. It is possible, however, that the received slave message exceeds this length. In such a situation, `PS_SPI_DROPPED_EXCESS_BYTES` is returned, meaning that num_bytes was placed into data but the remaining bytes were discarded.

There is no cause for alarm if the number of bytes read is less than `in_num_bytes` . This simply indicates that the incoming message was short.

The reception of bytes by the SPI subsystem, when it is configured as an SPI slave, is asynchronous with respect to the PC host software. Hence, there could be multiple responses queued up from previous write transactions.

#### SPI Slave Read Info Struct

A SPI slave read info struct type, `PromiraSpiSlaveReadInfo`, is used to provide important meta information about the slave transactions that have happened asynchronously.

```
struct PromiraSpiSlaveReadInfo {
    u32 in_data_bits;
    u32 out_data_bits;
    u08 header_bits;
    u08 resp_id;
    u08 ss_mask;
    u08 is_last;
};
```

**Table 16** : `PromiraSpiSlaveReadInfo` field descriptions

| | |
|---|---|
| `in_data_bits` | Number of bits received in `data_in`. |
| `out_data_bits` | Number of bits the slave shifted out. When the slave is configured as `PS_SPI_IO_STANDARD`, `out_data_bits` equals `in_data_bits`. For `PS_SPI_IO_DUAL` and `PS_SPI_IO_QUAD`, `out_data_bits` equals 0. |
| `header_bits` | Always 0. |
| `resp_id` | Always 0xff. |
| `ss_mask` | This is a bitmask of the slave select line that is set (similar to the bitmask found in `ps_spi_enable_ss`). `ss_mask` will be `0xff` if the user selects `PS_SPI_SLAVE_NO_SS`. |
| `is_last` | When `ps_spi_slave_host_read_size` is set to a value less than the total data size for a single transaction, the reads will be broken into chunks. The `is_last` flag is set on the last chunk. |

### Slave Data Lost Statistics (ps_spi_slave_data_lost_stats)

```
int ps_spi_slave_data_lost_stats (PromiraChannelHandle channel);
```

*Return number of slave read/write lost from a previous Promira SPI slave to SPI master transmission.*

**Arguments**

`channel` handle of the channel

**Return Value**

The function returns the number of SPI slave read/write

**Specific Error Codes**

`PS_SPI_SLAVE_READ_ERROR` The slave data is not SPI slave data lost.

**Details**

Returns the number of SPI transactions lost due to the overflow of the internal Promira buffer.

# 5.8 GPIO Interface

## 5.8.1 GPIO Notes

1. There is no check in the GPIO API calls to see if a particular GPIO line is enabled in the current configuration. If a line is not enabled for GPIO, the get function will simply return `0` for those bits. Another example is if one changes the GPIO directions for $I^2C$ lines while the $I^2C$ subsystem is still active. These new direction values will be cached and will automatically be activate if a later call to `ps_app_configure` disables the $I^2C$ subsystem and enables GPIO for the $I^2C$ lines. The same type of behavior holds for `ps_gpio_set`.
2. Additionally, for lines that are not configured as inputs, a change in the GPIO line using `ps_gpio_set` will be cached and will take effect the next time the line is active and configured as an input.
3. When initially starting an application with `pm_load`, the directions default to all input. Also the GPIO subsystem is off by default. It must be activated by using `ps_app_configure`.

## 5.8.2 GPIO Interface

**Direction (ps_gpio_direction)**

```
int ps_gpio_direction (PromiraChannelHandle channel,
                       u32                  direction_mask);
```

*Change the direction of the GPIO lines between input and output directions.*

**Arguments**

`channel` handle of the channel

| | |
|---|---|
| `direction_mask` | each bit corresponds to the physical line. If a line's bit is `0`, the line is configured as an input. Otherwise it will be an output. |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

None.

### Queue a GPIO Direction (ps_queue_gpio_direction)

```
int ps_queue_gpio_direction (PromiraQueueHandle queue,
                             u32                direction_mask);
```

*Queue a command that changes the direction of the GPIO lines between input and output directions.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `direction_mask` | each bit corresponds to the physical line. If a line's bit is `0`, the line is configured as an input. Otherwise it will be an output. |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

### Get Available GPIOs (ps_gpio_query)

```
int ps_gpio_query (PromiraChannelHandle channel);
```

*Returns the bitmask of which GPIOs are available based on the current app configuration.*

**Arguments**

`channel` handle of the channel

**Return Value**

An integer value, organized as a bitmask in the fashion. Any GPIO pin that is available will have a its corresponding bit active. If the line is not available as GPIO the bit will not be active in the bit mask.

**Specific Error Codes**

`None.`

**Details**

None.

### Get (ps_gpio_get)

```
int ps_gpio_get (PromiraChannelHandle channel);
```

*Get the value of current GPIO inputs.*

**Arguments**

`channel` handle of the channel

**Return Value**

An integer value, organized as a bitmask. Any line that is logic high will have a its corresponding bit active. If the line is logic low the bit will not be active in the bit mask.

**Specific Error Codes**

`None.`

**Details**

A line's bit position in the mask will be `0` if it is configured as an output or if it corresponds to a subsystem that is still active.

### Queue a GPIO Get (ps_queue_gpio_get)

```
int ps_queue_gpio_get (PromiraQueueHandle queue);
```

*Queue a command that gets the value of current GPIO inputs.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

The current state of the GPIO input lines will be returned with the function `ps_collect_resp`.

### Set (ps_gpio_set)

```
int ps_gpio_set (PromiraChannelHandle channel,
                 u32                  value);
```

*Set the value of current GPIO outputs.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `value` | a bitmask specifying which outputs should be set to logic high and which should be set to logic low. |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

If a line is configured as an input or not activated for GPIO, the output value will be cached. The next time the line is an output and activated for GPIO, the output value previously set will automatically take effect.

### Queue a GPIO Set (ps_queue_gpio_set)

```
int ps_queue_gpio_set (PromiraQueueHandle queue,
                       u32                value);
```

*Queue a command that sets the value of current GPIO outputs.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `value` | a bitmask specifying which outputs should be set to logic high and which should be set to logic low. |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

### Change (ps_gpio_change)

```
int ps_gpio_change (PromiraChannelHandle channel,
                    u16                  timeout_ms);
```

*Block until there is a change on the GPIO input lines.*

**Arguments**

| | |
|---|---|
| `channel` | handle of the channel |
| `timeout_ms` | time to wait for a change in milliseconds |

**Return Value**

The current state of the GPIO input lines.

**Specific Error Codes**

`None.`

**Details**

The function will return either when a change has occurred or the timeout expires. Pins configured for $I^2C$ or SPI will be ignored. Pins configured as outputs will be ignored. The timeout, specified in milliseconds, has a precision of approximately 2 ms. The maximum allowable timeout is approximately 60 seconds. If the timeout expires, this function will return the current state of the GPIO lines. It is the applications responsibility to save the old value of the lines and determine if there is a change based on the return value of this function.

The function `ps_gpio_change` will return immediately with the current value of the GPIO lines for the first invocation after any of the following functions are called: `ps_app_configure`, `ps_gpio_direction`.

#### Queue a GPIO Change (ps_queue_gpio_change)

```
int ps_queue_gpio_change (PromiraQueueHandle queue,
                          u16                timeout_ms);
```

*Queue a command that blocks until there is a change on the GPIO input lines.*

**Arguments**

| | |
|---|---|
| `queue` | handle of the queue |
| `timeout_ms` | time to wait for a change in milliseconds |

**Return Value**

A status code is returned with `PS_APP_OK` on success.

**Specific Error Codes**

`None.`

**Details**

This function queues the command, it will be executed when the function `ps_queue_submit` or `ps_queue_async_submit` is called.

The current state of the GPIO input lines will be returned with the function `ps_collect_resp`.

## 5.9 Error Codes

**Table 17** : I2C/SPI Active Applications Error Codes

| Literal Name | Value | ps_app_status_string() return value |
|---|---|---|
| PS_APP_OK | 0 | ok |
| PS_APP_UNABLE_TO_LOAD_LIBRARY | -1 | unable to load library |
| PS_APP_UNABLE_TO_LOAD_DRIVER | -2 | unable to load USB driver |
| PS_APP_UNABLE_TO_LOAD_FUNCTION | -3 | unable to load binding function |
| PS_APP_INCOMPATIBLE_LIBRARY | -4 | incompatible library version |
| PS_APP_INCOMPATIBLE_DEVICE | -5 | incompatible device version |
| PS_APP_COMMUNICATION_ERROR | -6 | communication error |
| PS_APP_UNABLE_TO_OPEN | -7 | unable to open device |
| PS_APP_UNABLE_TO_CLOSE | -8 | unable to close device |
| PS_APP_INVALID_HANDLE | -9 | invalid device handle |
| PS_APP_CONFIG_ERROR | -10 | configuration error |
| PS_APP_MEMORY_ALLOC_ERROR | -11 | unable to allocate memory |
| PS_APP_UNABLE_TO_INIT_SUBSYSTEM | -12 | unable to initialize subsystem |
| PS_APP_INVALID_LICENSE | -13 | invalid license |
| PS_APP_PENDING_ASYNC_CMD | -30 | pending respones to collect |
| PS_APP_TIMEOUT | -31 | timeout to collect a response |
| PS_APP_CONNECTION_LOST | -32 | connection lost |
| PS_APP_CONNECTION_FULL | -33 | too many connections |
| PS_APP_QUEUE_FULL | -50 | queue is full |
| PS_APP_QUEUE_INVALID_CMD_TYPE | -51 | invalid command to be added |
| PS_APP_QUEUE_EMPTY | -52 | no command to send |
| PS_APP_NO_MORE_TO_COLLECT | -80 | no more response to collect |
| PS_APP_UNKNOWN_CMD | -81 | unknown response received |
| PS_APP_MISMATCHED_CMD | -82 | response doesn't match with the command |
| PS_APP_UNKNOWN_CMD | -83 | unknown command sent |
| PS_APP_LOST_RESPONSE | -84 | response queue in the device was full |
| PS_I2C_NOT_AVAILABLE | -100 | i2c feature not available |
| PS_I2C_NOT_ENABLED | -101 | i2c not enabled |
| PS_I2C_READ_ERROR | -102 | i2c read error |
| PS_I2C_WRITE_ERROR | -103 | i2c write error |
| PS_I2C_SLAVE_BAD_CONFIG | -104 | i2c slave enable bad config |
| PS_I2C_SLAVE_READ_ERROR | -105 | i2c slave read error |
| PS_I2C_SLAVE_TIMEOUT | -106 | i2c slave timeout |

| | | |
|---|---|---|
| PS_I2C_DROPPED_EXCESS_BYTES | -107 | i2c slave dropped excess bytes |
| PS_I2C_BUS_ALREADY_FREE | -108 | i2c bus already free |
| PS_SPI_NOT_AVAILBLE | -200 | spi feature not available |
| PS_SPI_NOT_ENABLED | -201 | spi not enabled |
| PS_SPI_WRITE_ERROR | -202 | spi write error |
| PS_SPI_SLAVE_READ_ERROR | -203 | spi slave read error |
| PS_SPI_SLAVE_TIMEOUT | -204 | spi slave timeout |
| PS_SPI_DROPPED_EXCESS_BYTES | -205 | spi slave dropped excess bytes |

# 6 Electrical Specifications

## 6.1 DC Characteristics

**Table 18** : Absolute Maximum Ratings

| Symbol | Parameter | Conditions and Notes | Min | Max | Units |
|---|---|---|---|---|---|
| $V_{IO}$ | Input voltage (1, 3, 9, 14, 15, 17, 19, 20, 26, 31, 32, 33 pins) | | -0.5 | 5.5 | V |
| $V_{IO}$ | Input voltage (5, 7, 8, 11, 13, 21, 23, 25, 27 pins) | | -0.5 | 4.6 | V |

**Table 19** : Operating Conditions

| Symbol | Description | Conditions & Notes | Min | Max | Units |
|---|---|---|---|---|---|
| $T_a$ | Ambient Operating Temperature | | 10 (50) | 35 (95) | C (F) |
| $I_{Core}$ | Core Current Consumption | (1) | | 500 | mA |

Notes:

(1) The core current consumption includes the current consumption for the entire internal Promira platform, but does not include the output signals current consumption. Typical current consumption example at 5 V with 12.5 MHz single SPI master read operation is 340 mA using USB connection. Add 70 mA for operation with gigabit Ethernet connection.

**Table 20** : DC Characteristics (1)

| Symbol | Paramater | Conditions and Notes | Min | Max | Units |
|---|---|---|---|---|---|
| $V_{VTGT}$ | Target power voltage (4 and 6 pins) | | 3.3 | 5.0 | V |
| $V_{VIO}$ | IO power voltage (22 and 24 pins) | | 0.9 | 3.45 | V |
| $V_{IL}$ | Input low voltage (1, 3 pins) | | -0.5 | 0.25Vlevel | V |
| $V_{IH}$ | Input high voltage (1, 3 pins) | | 0.7Vlevel | 5.5 | V |

| | | | | | |
|---|---|---|---|---|---|
| $V_{OL}$ | Output low voltage (1, 3 pins) | | | 0.2 | V |
| $V_{OH}$ | Output high voltage (1, 3 pins) | (2) | | | V |
| $V_{IL}$ | Input low voltage (5, 7, 8, 11, 13, 21, 23, 25, 27 pins) | Vlevel=3.3V | | 0.8 | V |
| $V_{IH}$ | Input high voltage (5, 7, 8, 11, 13, 21, 23, 25, 27 pins) | Vlevel=3.3V | 2 | | V |
| $V_{OL}$ | Output low voltage (5, 7, 8, 11, 13, 21, 23, 25, 27 pins) | Vlevel=3.3V | | 0.7 | V |
| $V_{OH}$ | Output high voltage (5, 7, 8, 11, 13, 21, 23, 25, 27 pins) | Vlevel=3.3V | 2.3 | | V |
| $V_{IL}$ | Input low voltage (9, 14, 15, 17, 19, 20, 26, 31, 32, 33 pins) | | | 0.1Vlevel | V |
| $V_{IH}$ | Input high voltage (9, 14, 15, 17, 19, 20, 26, 31, 32, 33 pins) | | 0.9Vlevel | | V |
| $V_{OL}$ | Output low voltage (9, 14, 15, 17, 19, 20, 26, 31, 32, 33 pins) | | | 0.1Vlevel | V |
| $V_{OH}$ | Output high voltage (9, 14, 15, 17, 19, 20, 26, 31, 32, 33 pins) | | 0.9Vlevel | | V |
| $I_{VTGT}$ | Target power current (4 and 6 pins) | (3) | | 50 | mA |
| $I_{VIO}$ | IO power current (22 and 24 pins) | (3) | | 50 | mA |
| $I_{IO}$ | Input/output current ($I^2C$/ SPI/GPIO pins) | (4) | | 10 | mA |
| $I_{IOL}$ | Input/output leakage current ($I^2C$/SPI/GPIO pins) | (5) | | 100 | µA |
| $C_{IN}$ | Input capacitance ($I^2C$/SPI/ GPIO pins) | 1 MHz | | 8 | pF |

Notes:

(1) Vlevel is the configured level shift voltage for the $I^2C$/SPI/GPIO signals. The level shifter resolution is approximately 0.015V.

(2) Outputs are open collector, and therefor they are set by their pull-ups values and pull-ups voltage rail.

(3) Option 1: Two pins have 50 mA each. Option 2: One pin has 100 mA, and one pin has 0 mA, etc. Total current consumption on both pins should not exceed 100 mA.

(4) Option 1: Six pins have 10 mA each. Option 2: One pin has 60 mA, and the other pins have 0 mA. Etc. Total current consumption on all six pins should not exceed 60 mA.

(5) All $I^2C$/SPI/GPIO inputs (except for SPI SS signals) are high-impedance. Each SS signal has 10k Ohm pull-up resistor.

**Table 21** : Current Consumption Calculation Example

| Pin | Symbol | Description | Conditions & Notes | Max per pin | Max all pins | Units |
|---|---|---|---|---|---|---|
| NA | $I_{Core}$ | Core Current Consumption | (1) | 500 | 500 | mA |
| 4, 6 | $V_{TGT}$ | Target Power | (2) | 50 | 100 | mA |
| 22, 24 | $V_{IO}$ | IO Power | (2) | 50 | 100 | mA |
| 1, 3, 9, 14, 15, 17, 19, 20, 26, 31, 32, 33. 5, 7, 8, 11, 13, 21, 23, 25, 27 | SCL/GPIO00, SDA/GPIO-01, SS0/GPIO02, SS2/GPIO03, SS1/GPIO04, SS3/GPIO05, SS4/GPIO06, SS5/GPIO07, SS6/GPIO08, GPIO13, GPIO14, SS7/GPIO15, MISO, SCLK, MOSI, IO2, IO3, GPIO9, GPIO10, GPIO11, GPIO12 | I2C/SPI Signals | (3) | 10 | 60 | mA |
| | Total Current Consumption For Promira Core and Outputs | | (4) | | 760 | mA |

Notes:

(1) The core current consumption includes the current consumption for the entire internal Promira platform, but does not include the output signals current consumption.

(2) Option 1: Two pins have 50 mA each. Option 2: One pin has 100 mA, and one pin has 0 mA. Etc. Total current consumption on both pins should not exceed 100 mA.

(3) Option 1: Six pins have 10 mA each. Option 2: One pin has 60 mA, and the other pins have 0 mA. Etc. Total current consumption on all pins should not exceed 60 mA.

(4) If the total current consumption for the Promira platform core and outputs is over 500 mA, then a USB 3.0 port and USB 2.0 cable or Total Phase external AC adapter should be used. A USB 3.0 port supplies up to 900 mA. A USB 2.0 port supplies up to 500 mA. Total Phase external AC adapter supplies up to 1.2 A. In this example the total current consumption for the Promira platform core and outputs is 760 mA, therefor USB 3.0 port and USB 2.0 cable or Total Phase external AC adapter should be used.

## 6.2 AC Characteristics

### 6.2.1 SPI AC Characteristics

**Table 22 (1)** : SPI Master Timing Parameters

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $t_{SSSCK}$ | SS# assert to first SCLK rising edge (User configurable (3)) | | 1 | | $t_{SCK}$ |
| $t_{SCKSS}$ | Last SCLK failing edge to SS# deassert (User configurable (3)) | | 1.5 | | $t_{SCK}$ |
| $t_{SCK}$ | Clock period | 12.5 | | | ns |
| $t_v$ | Output delay for data output w.r.t to SCLK (from enable edge of SCLK to data being stable on data output) | | 2 | | ns |
| $t_{HO}$ | Data hold output from enable edge of SCLK | | 0 | | ns |
| $t_{SU}$ | Setup time for data input w.r.t to capturing edge of SCLK | 3 | | | ns |
| $t_H$ | Hold time for data input w.r.t to capturing edge of SCLK | 3 | | | ns |
| $t_{WORD}$ | Delay between data words (user configurable (4)) | 0 | | | $t_{SCK}$ |
| $t_{TXN}$ | Delay between 2 data transactions (SS deassert to SS assert. User configurable (5)) | 0 | | | $t_{SCK}$ |

Notes:

(1) All values are for 3.3V SPI signals condition. See also sections 2.3.1 & 2.3.3.

(2) Promira Quad SPI master employs a flexible sampling scheme on the input data to enable the device to run at higher speeds. The data is internally sampled half a clock cycle later

(3) The user can configure the SPI timing parameters $t_{SSSCK}$ and $t_{SCKSS}$ by adding the API function ps_queue_spi_delay_cycles between the API functions ps_queue_spi_ss and ps_queue_spi_write.

(4) The user can configure the SPI timing parameter $t_{WORD}$ by using the API function ps_spi_configure_delays.

(5) The user can configure the SPI timing parameter $t_{TXN}$ by adding the API function ps_queue_spi_delay_cycles between submitting twice the function ps_queue_spi_ss (the first one for SS deassert and the second one for SS assert).

**Table 23 (1)** : SPI Slave Timing Parameters

| Symbol | Parameter | Min | Typical | Max | Units |
|---|---|---|---|---|---|
| $t_{SSSCK}$ | SS# assert to first SCLK rising edge | 1 | | | $t_{SCK}$ |
| $t_{SCKSS}$ | Last SCLK failing edge to SS# deassert | 1 | | | $t_{SCK}$ |
| $t_{SCK}$ | Clock period | 50 | | | ns |
| $t_v$ | Output delay for data output w.r.t to SCLK (from enable edge of SCLK to data being stable on data output) | | 20 | | ns |
| $t_{HO}$ | Data hold output from enable edge of SCLK | | 0.5 | | $t_{SCK}$ |
| $t_{SU}$ | Setup time for data input w.r.t to capturing edge of SCLK | 5 | | | ns |
| $t_H$ | Hold time for data input w.r.t to capturing edge of SCLK | 5 | | | ns |
| $t_{WORD}$ | Delay between data words | 0 | | | $t_{SCK}$ |
| $t_{TXN}$ | Delay between 2 data transactions (SS deassert to SS assert.) | 0 | | | $t_{SCK}$ |

Notes:

(1) All values are for 3.3V SPI signals condition. See also sections 2.3.1 & 2.3.3.

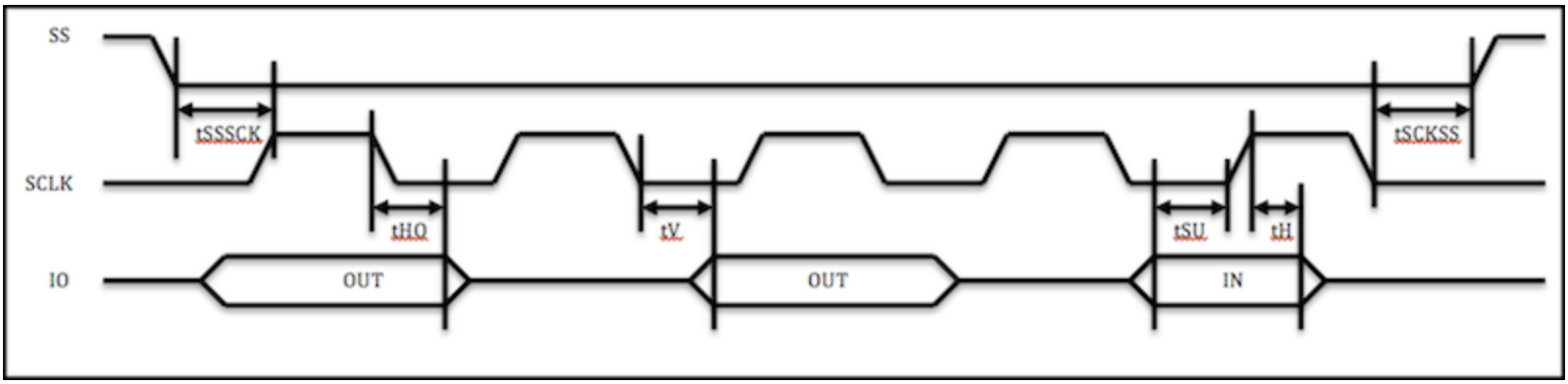


***Figure 7** : SPI Master / Slave (POL=0 PHA=0 SS=L) Waveform*

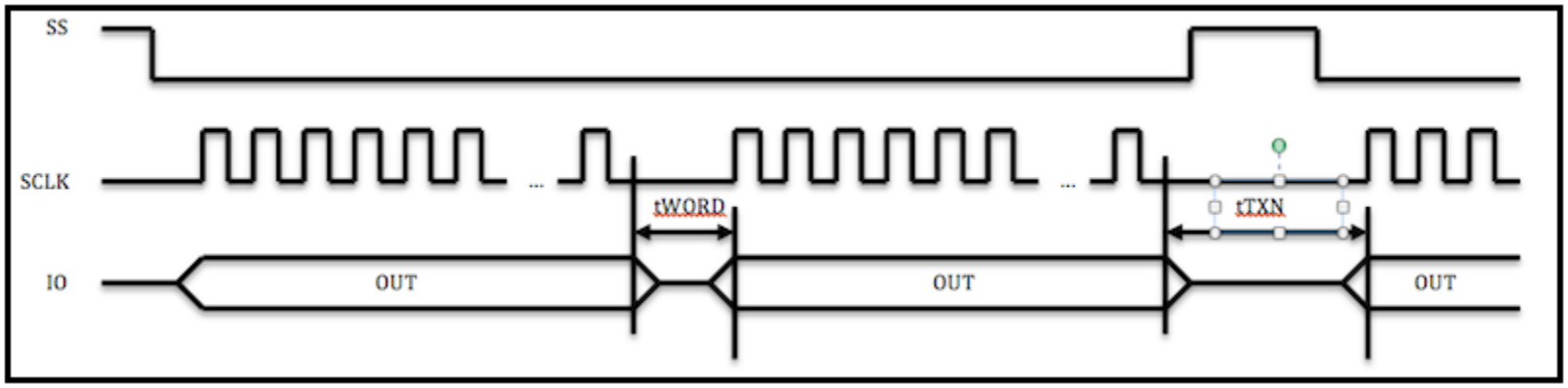


***Figure 8** : SPI Master / Slave (POL=0 PHA=0 SS=L) Byte Timing*

### 6.2.2 I²C AC Characteristics

**Table 24** : I²C Standard Mode Timing Parameters

| Symbol | Parameter | Condition | Min | Max | Units |
|---|---|---|---|---|---|
| $t_{SCL}$ | SCL Clock Frequency | | 0 | 100 | KHz |
| $t_{HD;SA}$ | Hold Time (repeated) START Condition | | 4.0 | - | µs |
| $t_{LOW}$ | Low period of the SCL Clock | | 4.7 | - | µs |
| $t_{HIGH}$ | High period of the SCL Clock | | 4.0 | - | µs |
| $t_{SU;STA}$ | Set-up time for a repeated START condition | | 4.7 | - | µs |
| $t_{HD;DAT}$ | Data hold time | | 0 | - | µs |
| $t_{SU;DAT}$ | Data setup time | | 250 | - | ns |

| | | | | | |
|---|---|---|---|---|---|
| $t_{SU;STO}$ | Setup time for STOP condition | | 4.0 | - | µs |
| $t_{BUF}$ | Bus free time between STOP and START condition | | 4.7 | - | µs |

**Table 25** : I²C Fast Mode Timing Parameters

| Symbol | Parameter | Condition | Min | Max | Units |
|---|---|---|---|---|---|
| $t_{SCL}$ | SCL Clock Frequency | | 0 | 400 | KHz |
| $t_{HD;SA}$ | Hold Time (repeated) START Condition | | 0.6 | - | µs |
| $t_{LOW}$ | Low period of the SCL Clock | | 1.3 | - | µs |
| $t_{HIGH}$ | High period of the SCL Clock | | 0.6 | - | µs |
| $t_{SU;STA}$ | Set-up time for a repeated START condition | | 0.6 | - | µs |
| $t_{HD;DAT}$ | Data hold time | | 0 | - | µs |
| $t_{SU;DAT}$ | Data setup time | | 100 | - | ns |
| $t_{SU;STO}$ | Setup time for STOP condition | | 0.6 | - | µs |
| $t_{BUF}$ | Bus free time between STOP and START condition | | 1.3 | - | µs |

**Table 26** : I²C Fast Mode Plus Timing Parameters

| Symbol | Parameter | Condition | Min | Max | Units |
|---|---|---|---|---|---|
| $t_{SCL}$ | SCL Clock Frequency | | 0 | 1000 | KHz |
| $t_{HD;SA}$ | Hold Time (repeated) START Condition | | 0.26 | - | µs |
| $t_{LOW}$ | Low period of the SCL Clock | | 0.5 | - | µs |
| $t_{HIGH}$ | High period of the SCL Clock | | 0.26 | - | µs |
| $t_{SU;STA}$ | Set-up time for a repeated START condition | | 0.26 | - | µs |
| $t_{HD;DAT}$ | Data hold time | | 0 | - | µs |
| $t_{SU;DAT}$ | Data setup time | | 50 | - | ns |
| $t_{SU;STO}$ | Setup time for STOP condition | | 0.26 | - | µs |
| $t_{BUF}$ | Bus free time between STOP and START condition | | 0.5 | - | µs |

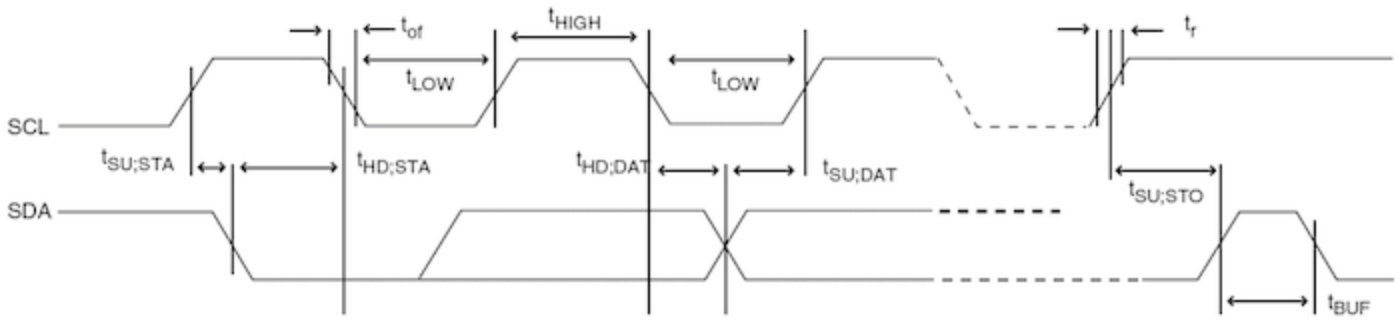


***Figure 9** : I²C Standard, Fast, & Fast Plus modes Waveform*

### 6.2.3 General AC Characteristics

**Table 27** : Internal Delay Parameters

| Voltage Vlvl (V) | Delay $t_{OPD}$, $t_{IPD}$(n) Typical |
|---|---|
| 3.3 | 1.4 |
| 2.5 | 1.8 |
| 1.8 | 2.2 |
| 1.5 | 2.7 |
| 1.2 | 3.4 |
| 1.0 | 4.6 |
| 0.9 | 5.8 |

**Table 28** : Standard 120 mm Cable Delay Parameters

| Delay $t_{COPD}$, $t_{CIPD}$ (n) Typical |
|---|
| 1.74 |

Note:

(1) Maximum I²C/SPI Signal Frequency = Min ($1/t_{SCK}$ ; $1/(t_{OPD}+t_{IPD}+t_{COPD}+t_{CIPD})$)

## 6.3 Signal Ratings

### 6.3.1 Logic High Levels

All signal levels are nominally 0.9-3.45 volts (+/- 10%) logic high. The Promira Serial Platform is also compatible with devices with 5V $I^2C$/SPI signals level.

### 6.3.2 ESD protection

The Promira Serial Platform has built-in electrostatic discharge protection to prevent damage to the unit from high voltage static electricity.

# 7 Legal / Contact

## 7.1 Disclaimer

All of the software and documentation provided in this manual, is copyright Total Phase, Inc. ("Total Phase"). License is granted to the user to freely use and distribute the software and documentation in complete and unaltered form, provided that the purpose is to use or evaluate Total Phase products. Distribution rights do not include public posting or mirroring on Internet websites. Only a link to the Total Phase download area can be provided on such public websites.

Total Phase shall in no event be liable to any party for direct, indirect, special, general, incidental, or consequential damages arising from the use of its site, the software or documentation downloaded from its site, or any derivative works thereof, even if Total Phase or distributors have been advised of the possibility of such damage. The software, its documentation, and any derivative works is provided on an "as-is" basis, and thus comes with absolutely no warranty, either express or implied. This disclaimer includes, but is not limited to, implied warranties of merchantability, fitness for any particular purpose, and non-infringement. Total Phase and distributors have no obligation to provide maintenance, support, or updates.

Information in this document is subject to change without notice and should not be construed as a commitment by Total Phase. While the information contained herein is believed to be accurate, Total Phase assumes no responsibility for any errors and/or omissions that may appear in this document.

## 7.2 Life Support Equipment Policy

Total Phase products are not authorized for use in life support devices or systems. Life support devices or systems include, but are not limited to, surgical implants, medical systems, and other safety-critical systems in which failure of a Total Phase product could cause personal injury or loss of life. Should a Total Phase product be used in such an unauthorized manner, Buyer agrees to indemnify and hold harmless Total Phase, its officers, employees, affiliates, and distributors from any and all claims arising from such use, even if such claim alleges that Total Phase was negligent in the design or manufacture of its product.

## 7.3 Contact Information

Total Phase can be found on the Internet at http://www.totalphase.com/. If you have support-related questions, please go to the Total Phase support page at http://www.totalphase.com/support/. For sales inquiries, please contact sales@totalphase.com.