Author:	Sierra Wireless						Date:	March 12, 2012			
APN Content Level	BASIC	✓	INTERMEDIATE		ADVANCED		Confidentiality	Public	✓	Private	

Hardware Compatibility	Product Line	AirPrime	Series	Q26xx	SL60xx
				WMPxx	

Software Compatibility	Series	Q26xx : >6.6x
		OTHERS : ALL

1 Version

Application Notes may be updated over their lifetime. To ensure you design with the correct version, check the application notes page at www.sierrawireless.com for latest versions.

2 Introduction

This Application Note (APN) is provided to Sierra Wireless distributors and clients to aid more rapid development of embedded applications using the Sierra Wireless portfolio of cellular solutions. To request a new application note, contact your regional Sierra Wireless Product Marketing Manager.

3 Overview

Bus services can be used to communicate between two or more devices which are connected together. The communication between the devices can follow different protocols which are explained in detail in the document.

The document describes about the three types of the bus services.

1. SPI
2. I^2C
3. Parallel bus*

*AirPrime Q2686 supports only SPI bus and I^2C bus. Parallel bus is not supported.

4 Glossary

Initials	Definition
EEPROM	Electrically Erasable Programmable Read-only memory
I2C	Point to Point Protocol
LCD	Liquid Crystal Display
SPI	Serial Peripheral Interface

5 SPI Bus

5.1 Introduction

SPI bus is a synchronous serial data link that operates in full duplex (signals carrying data go in both directions simultaneously).

Devices communicate using a master/slave relationship, in which the master initiates the data frame. When the master generates a clock and selects a slave device, data may be transferred in either or both directions simultaneously. In fact, as far as SPI is concerned, data are always transferred in both directions. It is up to the master and slave devices to know whether a received byte is meaningful or not. So a device must discard the received byte in a "transmit only" frame or generate a dummy byte for a "receive only" frame.

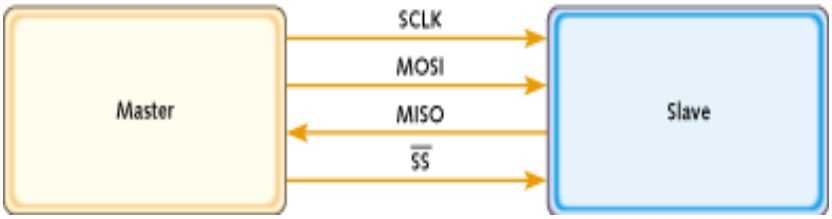

Figure 1. SPI Signals in Single-Slave Configuration

SPI specifies four signals:

1. Clock (SCLK)
2. Master data output, slave data input (MOSI)
3. Master data input, slave data output (MISO)
4. Slave select or chip select signal (ÇSS).

Figure 1 shows these signals in a single-slave configuration. SCLK is generated by the master and input to all slaves. MOSI carries data from master to slave. MISO carries data from slave back to master. A slave device is selected when the master asserts its CSS signal.

If multiple slave devices exist, the master generates a separate slave select signal for each slave. These relationships are illustrated in Figure 2.

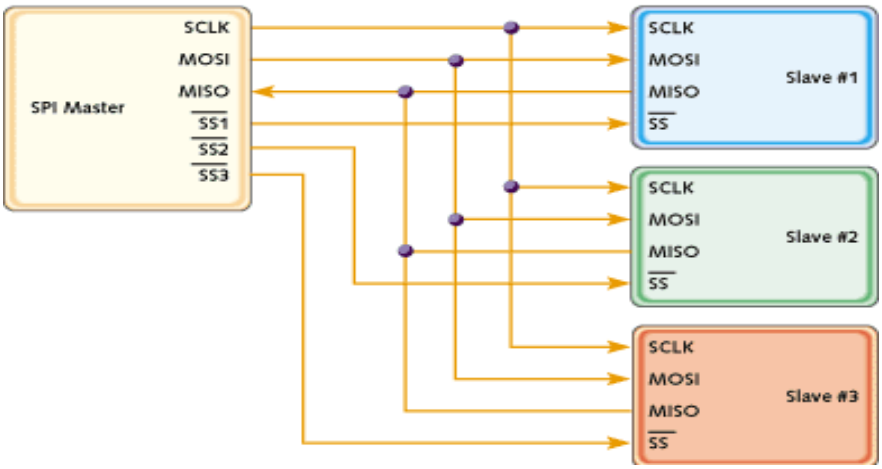

Figure 2. SPI Multiple Slave Relationships

The master generates slave select signals using general-purpose discrete input/output pins or other logic. This consists of old-fashioned bit banging and can be pretty sensitive. You have to time it relative to the other signals and ensure, for example, that you don't toggle a select line in the middle of a frame.

5.2 Advantages of SPI

SPI can achieve significantly higher data rates. SPI-compatible interfaces often range into the tens of megahertz. SPI gains efficiency in applications that take advantage of its duplex capability, such as the communication between a "codec" (coder-decoder) and a digital signal processor, which consists of simultaneously sending samples in and out.

5.3 Disadvantages of SPI

SPI devices communicate using a master-slave relationship. Due to its lack of built-in device addressing, SPI requires more effort and more hardware resources when more than one slave is involved. But SPI tends to be simpler and more efficient in point-to-point (single master, single slave) applications.

5.4 Examples

Devices like analog-to-digital and digital-to-analog converters, LCDs, and temperature sensors, support serial interfaces.

5.5 SPI with Open AT Framework

To use SPI bus with Open AT Framework, perform the steps listed below, as defined in the following subsections.

1. Configure the SPI bus.
2. Subscribe to the SPI bus.
3. Read or write from the SPI bus.
4. Unsubscribe to the bus.

5.5.1 Configure the SPI Bus

To configure the SPI bus, the following structure has to be declared with the appropriate values.

```
typedef struct
{
u32 Clk_Speed;
u32 Clk_Mode;
u32 ChipSelect;
u32 ChipSelectPolarity;
u32 LsbFirst;
adl_ioDefs_t GpioChipSelect;
u32 LoadSignal;
u32 DataLinesConf;
u32 MasterMode;
u32 BusySignal;
} adl_busSPISettings_t;
```

Parameters

- Clk_Speed: This parameter allows to set the SPI bus Clock speed.
- ChipSelect: This parameter allows the user to select the Chip Select signal.
- ChipSelectPolarity: This parameter sets the polarity of the Chip Select signal.
- LsbFirst : This parameter defines the priority for data transmission through the SPI bus, LSB or MSB first. This applies only to data. The Opcode and Address fields sent are always sent with MSB first.
- GpioChipSelect: This parameter defines the GPIO Chip Select. This parameter is used only if the ChipSelect parameter is set to the ADL_BUS_SPI_ADDR_CS_GPIO value. It sets the GPIO label to use as the chip select signal.
- LoadSignal: This parameter defines the LOAD signal behavior.
- DataLinesConf: This parameter defines if the SPI bus uses one single pin to handle both input and output data signals, or two pins to handle them separately.
- MasterMode: This parameter defines whether the SPI is running as master or slave.
- BusySignal: This parameter defines the LOAD signal Behavior.

For more information on the SPI parameters, refer to the ADL User Guide for Open AT Framework OS.

Example

```
adl_busSPISettings_t SpiSettings =
{
5,                          // Clk_Speed    (2 MHz)
ADL_BUS_SPI_CLK_MODE_0,     // Clk_Mode;
             ADL_BUS_SPI_ADDR_CS_GPIO,    // ChipSelect;
             ADL_BUS_SPI_CS_POL_LOW,      // ChipSelectPolarity;
             ADL_BUS_SPI_MSB_FIRST,       // LsbFirst;
             ADL_IO_GPIO | 35,            // GpioChipSelect;
             ADL_BUS_SPI_FRAME_HANDLING, // WriteHandling;
```

```
                ADL_BUS_SPI_DATA_UNIDIR,     // DataLinesConf;
                ADL_BUS_SPI_MASTER_MODE,   //SPI in master mode
                ADL_BUS_SPI_BUSY_UNUSED    //Busy signal is not used
        };
```

5.5.2 Subscribe to the SPI BUS

The following API should be used to subscribe to the SPI bus.

```
adl_busSubscribe (adl_busID_e BusId, u32 BlockId, void *  BusParam ).
```

Parameters

- BusId: Open AT Framework supports to connect the device to two SPI buses (SPI1 and SPI2). This parameter describes the type of the bus to subscribe to.
- BlockId: ID of the block to use (in the range 1-N, where N is spi_NbBlocks).
- BusParam: Subscribed bus configuration parameters.

Example

To subscribe to SPI-1 Bus:

```
Spi_Handle = adl_busSubscribe ( ADL_BUS_SPI1,  2,  &SpiSettings )
```

To Subscribe to SPI-2 Bus:

```
Spi_Handle  = adl_busSubscribe (ADL_BUS_SPI2,  1,  &I2CSettings )
```

5.5.3 Read/Write to SPI Bus

Bus Read and write operations can be performed in two modes.

1. Synchronous mode: In synchronous mode, bus related operations are not allowed from low level handlers. This is because with synchronous bus operations, there is the possibility of low level interrupt handler's calls being blocked due to an ongoing bus operation.
2. Asynchronous mode: In this mode, the bus related operations are allowed from the low level handlers.

These two modes can be set using adl_busIOCtl () API. The syntax of this API is:

```
adl_busIOCtl (u32 Handle, adl_busIoCtlCmd_e Cmd, void * Param );
```

refer to the ADL User Guide for Open AT Framework OS for more information about this API.

5.5.3.1 Write to SPI

To Write on to the SPI bus, the following API has to be used.

```
s8 adl_busWrite (u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void *
pDataToRead)
```

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- pAccessMode: This should be left NULL for SPI bus.
- Length : This is the number of bytes to be written into the SPI bus.
- pDatatoRead: This is the buffer to write on the bus.

5.5.3.2 Read from SPI

To read from the SPI bus, the following API needs to be used.

```
s8 adl_busRead (u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void *
pDataToRead)
```

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- pAccessMode: This should be left NULL for SPI bus.
- Length : This is the number of bytes to read from the bus.
- pDatatoRead: This is the buffer to copy the read bytes.

Example

To read from the bus

```
adl_busRead ( Spi_Handle, &AccessConfig, READ_SIZE, ReadBuffer);
```

To write onto the bus

```
funcRetValue = adl_busWrite ( Spi_Handle, &AccessConfig, WRITE_SIZE, WriteBuffer);
```

5.5.4 Unsubscribe SPI bus

The following API can be used to unsubscribe to the bus.

```
adl_busUnsubscribe (I2c_Handle);
```

6 I²C BUS

6.1 Introduction

The I²C bus is a half-duplex, synchronous, multi-master bus requiring only two signal wires: data (SDA) and clock (SCL). These lines are pulled high via pull-up resistors and controlled by the hardware via open-drain drivers, giving a wired-AND interface.

I²C uses an addressable communications protocol that allows the master to communicate with individual slaves using a 7-bit or 10-bit address. Each I²C-compatible hardware slave device comes with a predefined device address, the lower bits of which may be configurable at the board level. The master transmits the device address of the intended slave at the beginning of every transaction. Each slave is responsible for monitoring the bus and responding only to its own address. This addressing scheme limits the number of identical slave devices that can exist on an I²C bus without contention, with the limit set by the number of user-configurable address bits (typically two bits, allowing up to four identical slave devices).

6.2 Communication

Displayed in the figure below, the master begins the communication by issuing the start condition (S). The master continues by sending a unique 7-bit slave device address, with the most significant bit (MSB) first. The eighth bit after the start, read/not-write (), specifies whether the slave is now to receive (0) or to transmit (1). This is followed by an ACK bit issued by the receiver, acknowledging receipt of the previous byte. Then the transmitter (slave or master, as indicated by the bit) transmits a byte of data starting with the MSB. At the end of the byte, the receiver (whether master or slave) issues a new ACK bit. This 9-bit pattern is repeated if more bytes need to be transmitted.

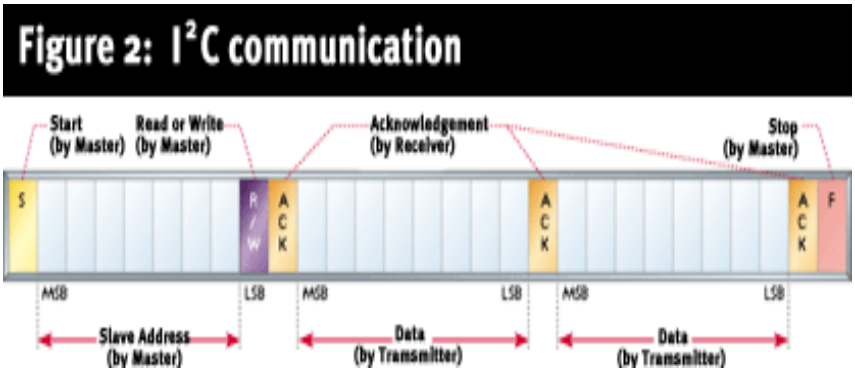

Figure 3. I²C Communication Diagram

In a write transaction (slave receiving), when the master is done transmitting all of the data bytes it wants to send, it monitors the last ACK and then issues the stop condition (P). In a read transaction (slave transmitting), the master does not acknowledge the final byte it receives. This tells the slave that its transmission is done. The master then issues the stop condition.

I²C has a rather interesting feature called clock stretching, which is done when the slave device is unable to process the bit and requires more time. When this happens, the slave pulls the SCL line low. Because the signal behaves as a wired-AND, when the master releases the SCL line while the slave is "stretching" the clock, the master should notice that the line stays low. Upon seeing this, the master waits until the slave has processed the data bit and released the line. Once released by the slave, the SCL line floats back high, signaling to the master to send the next data bit.

6.3 Advantages

As there is no need for chip select or arbitration logic, it is simple to implement in hardware which makes it an inexpensive solution. Also, when more than one device is used, it requires less hardware.

6.4 Disadvantages

It is a half duplex bus.

6.5 Examples

Examples of simple I²C-compatible devices found in embedded systems include EEPROMs, thermal sensors, and real-time clocks.

6.6 I²C with Open AT Framework

To use I²C bus with Open AT Framework, perform the steps listed below, as defined in the following subsections.

1. Configure the I²C bus
2. Subscribe to the I²C bus
3. Read or write from the I²C bus.
4. Unsubscribe the I²C bus.

6.6.1 Configure the I²C Bus

To configure the I²C bus, the following structure has to be declared with the appropriate values.

```
typedef struct
{
u32 ChipAddress;
u32 Clk_Speed;
u32 AddrLength;
u32 MasterMode;
} adl_busI2CSettings_t;
```

Parameters

- Chip Address: This parameter sets the remote chip N bit address on the I²C bus.

Note: From Open AT Framework 2.0 versions, the chip address are be defined from bit b0 to bit b(N-1), where as in previous versions it is from bit b1 to bit b(N). Hence, from Open AT Framework 2.0 versions the chip address should be defined as (chipaddress >>1), as the bus architecture is changed.

- Clk_Speed: This parameter sets the required I²C bus speed.
- AddrLength: This parameter sets the remote chip address length configuration.
- MasterMode: This parameter defines whether I²C is running in master or slave mode.

Refer to the ADL User Guide for Open AT Framework OS for more information on the I²C parameters.

Examples

Versions before Open AT Framework 2.0:

```
adl_busI2CSettings_t I2CSettings =
    {
         0xA0,                    // ChipAddress
         ADL_BUS_I2C_CLK_STD,      // Clk_Speed
         ADL_BUS_I2C_ADDR_7_BITS,
         ADL_BUS_I2C_MASTER_MODE
    };
```

Versions from Open AT Framework 2.0:

```
adl_busI2CSettings_t I2CSettings =
    {
        0xA0 >> 1,                   // ChipAddress
```

```
            ADL_BUS_I2C_CLK_STD,      // Clk_Speed
            ADL_BUS_I2C_ADDR_7_BITS,
            ADL_BUS_I2C_MASTER_MODE
        };
```

6.6.2 Subscribe to the I²C bus

The following API should be used to subscribe to the I²C bus.

```
adl_busSubscribe (adl_busID_e BusId, u32 BlockId, void *  BusParam ).
```

Parameters

- BusId: This parameter describes the type of the bus to subscribe to.
- BlockId: ID of the block to use (in the range 1-N, where N is spi_NbBlocks).
- BusParam: Subscribed bus configuration parameters.

Example

```
I2c_Handle = adl_busSubscribe ( ADL_BUS_I2C, 1, &I2CSettings)
```

6.6.3 Read/Write to I²C Bus

6.6.3.1 Write to I²C

To write on to the I²C bus, the following API has to be used.

```
s8 adl_busWrite (u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void *
pDataToRead)
```

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- pAccessMode: This should be left NULL for I²C bus.
- Length : This is the number of bytes to be written into the I²C bus.
- pDatatoRead: This is the buffer to write on the bus.

6.6.3.2 Read from I²C

To read from the I²C bus, the following API needs to be used.

```
s8 adl_busRead (u8 Handle, adl_busAccess_t * pAccessMode, u32 Length, void *
pDataToRead)
```

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- pAccessMode: This should be left NULL for I²C bus.
- Length : This is the number of bytes to read from the bus.
- pDatatoRead: This is the buffer to copy the read bytes

Example

To read from the bus:

```
adl_busRead ( I2c_Handle, &AccessConfig, READ_SIZE, ReadBuffer);
```

To write onto the bus:

```
funcRetValue = adl_busWrite ( I2c_Handle, &AccessConfig, WRITE_SIZE, WriteBuffer);
```

6.6.4 Unsubscribe the I²C bus

The following API can be used to unsubscribe to the bus.

```
adl_busUnsubscribe (I2c_Handle).
```

7 PARALLEL BUS

7.1 Introduction

A parallel port is a type of interface found on computers (personal and otherwise) for connecting various peripherals. It is also known as a printer port or Centronics port. The IEEE 1284 standard defines the bi-directional version of the port.

Parallel bus transmission is the transmission of several bits at the same time, with each bit transmitted over a separate wire. An 8-bit parallel channel transmits eight bits (or a byte) simultaneously. A serial channel would transmit those bits one at a time. If both operated at the same clock speed, the parallel channel would be eight times faster.

Several computer interfaces use or used parallel transmission: ATA or IDE interface, SCSI Parallel Interface (SPI; the traditional SCSI), and the parallel port.

The Open AT Framework library has implemented Parallel bus. On the parallel bus, we can configure the parameters like AccessTime, SetupTime, HoldTime, TurnaroundTime and these values are required to be programmed while communicating with the parallel bus. The parallel bus does not have a clock signal like the serial protocol buses like SPI or I^2C. Hence, we don't specify the maximum frequency of the external device connected to the parallel bus. Rather, we talk in terms of the above mentioned parameters. The parameters specified above must be checked with the external peripheral to be connected, and the user has to check what is the maximum and minimum value for these parameters. (Normally, the parallel bus device manufacturer will provide the maximum and minimum value for the above mentioned parameters and the embedded module must configure these values to the correct ones to be able to send/receive data from the parallel bus.)

Caution: *Parallel bus is not supported on AirPrime Q2686.*

7.2 Advantages

Parallel buses, in particular PCI, are very widely deployed and abundant cheap silicon is available.

For lower data rates, Parallel bus performs adequately.

Hot-swappable PCI solutions are available, allowing users to add or remove PCI adapter cards without having to shut down the system.

7.3 Disadvantages

Scalability is limited by bus bandwidth and latency, and by available memory.

7.4 Examples

LCD, PCI and printers

7.5 Parallel Bus with Open AT Framework

To use Parallel bus with Open AT Framework, perform the steps listed below, as defined in the following subsections.

1. Configure the Parallel bus
2. Subscribe to the Parallel bus
3. Read or write from the Parallel bus.
4. Unsubscribe the Parallel bus.

7.5.1 Configure the Parallel Bus

To configure the Parallel bus, the following structure has to be declared with the appropriate values.

```
typedef struct {
u8 Width;
u8 Mode;
u8 pad [2];
adl_busParallelTimingsCfg_t ReadCfg;
adl_busParallelTimingsCfg_t WriteCfg;
adl_busParallelCs_t Cs;
adl_busParallelPageCfg_t PageCfg;
adl_busParallelSynchronousCfg_t SynchronousCfg;
u32 AddressPin;
} adl_busParallelSettings_t;
```

Parameters

- Width: This parameter defines the read/write process data buffer items bit size, using the adl_busParallelSize_e type.
- Mode: This parameter defines the required parallel bus standard mode to be used, using the adl_busParallel_Bus_Mode_e type.
- ReadCfg: Define the timing configuration for each read and write process, using the adl_busParallelTimingCfg_t type.
- WriteCfg: Define the timing configuration for each read and write process, using the adl_busParallelTimingCfg_t type.
- Cs: Configuration parameters for the page mode.During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.
- PageCfg: Configuration parameters for the page mode.During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.
- SynchronousCfg : Configuration of the synchronous mode.This structure hosts the parameters used to configure the synchronous mode accesses..
- AddressPin : Select the pin used for the parallel bus.This is a bitfield, each bit represents a pin of the parallel bus.e.g.: 0x03, two address pin are used (A0 and A1).

For more information on the Parallel parameters, refer to the ADL User Guide for Open AT Framework OS.

Example

```
adl_busParallelSettings_t ParallelBusConfig =
 {
       ADL_BUS_PARALLEL_WIDTH_16_BITS,  // 16 bits parallel bus
       ADL_BUS_PARALLEL_MODE_ASYNC_INTEL ,   // Motorola mode, Low E signal polarity
              {0x00,0x00},
              { 2, 1, 1, 2, 0, 0, 0 , 0 },  // Read timing settings
              { 2, 1, 1, 2, 0, 0, 0 , 0  },  // Write timing settings
              {
                     ADL_BUS_PARA_CS_TYPE_CS,    3, // Use CS3 pin
                            {0x00,0x00}
                     },
              { 0, 0 },
              { 0, 0, 0, 0, 0 },
0x07FFFFF,
};
```

7.5.2 Subscribe to the Parallel bus

The following API should be used to subscribe to the Parallel bus.

```
adl_busSubscribe (adl_busID_e BusId, u32BlockId, void *  BusParam ).
```

Parameters

- BusId: Open AT Framework supports to connect the device to two SPI buses (SPI1 and SPI2). This parameter describes the type of the bus to subscribe to.
- BlockId: ID of the block to use (in the range 1-N, where N is spi_NbBlocks).
- BusParam: Subscribed bus configuration parameters.

Example:

To subscribe to Parallel Bus:

```
parallel_Handle = adl_busSubscribe ( ADL_BUS_PARALLEL , 1 , ( adl_busSettings_u * )
&ParallelBusConfig );
```

7.5.3 Read/Write to Parallel Bus

7.5.3.1 Write to Parallel

To write on to the Parallel bus, the following API has to be used.

```
s32 adl_busDirectWrite (s32 Handle, u32 ChipAddress, u32 Length, void * pDataToWrite
);
```

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- ChipAddress: This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.
- Length : This is the number of bytes to be written into the Parallel bus.
- pDatatoWrite: Data buffer to write on the bus, item bit size (8 or 16 bits) is defined at subscription time in the configuration structure (see adl_busParallelSettings_t).

7.5.3.2 Read from Parallel

To read from the Parallel bus, the following API needs to be used.

s32 adl_busDirectRead (s32 Handle, u32 ChipAddress, u32 DataLen, void * Data);

Parameters

- Handle: The handle returned during the subscription of the bus using adl_busSubscribe () function.
- ChipAddress: This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.
- Length: This is the number of bytes to be read from the Parallel bus.
- Data: Buffer into which the read items are copied, items bit size (8 or 16 bits) is defined at subscription time in the configuration structure (see adl_busParallelSettings_t).

Example

To read from the bus:

```
adl_busDirectRead (Parallel_Handle, ChipAddress, READ_SIZE, ReadBuffer);
```

To write onto the bus:

```
funcRetValue = adl_busDirectWrite (Parallel_Handle, ChipAddress, WRITE_SIZE,
WriteBuffer );
```

7.5.4 Unsubscribe the Parallel Bus

The following API can be used to unsubscribe to the bus.

```
adl_busUnsubscribe (Parallel_Handle) .
```

8 Software Compatibility Matrix

List all current software configurations and compatibility with this application note.

Firmware	Open AT Framework/SDK	Libraries
6.6x	Open AT® SDK v4.2x	N/A
R6.5	Open AT® Software Suite v1.X	N/A
R7.x	Open AT Framework v2.X	N/A

9 Reference Documents

	Filename	Comment
[1]	ADL User Guide for Open AT Framework OS	A section in this document describes BUS API's as well its configurations.

10 Support

For direct clients: contact your Sierra Wireless FAE

For distributor clients: contact your distributor FAE
For distributors: contact your Sierra Wireless FAE

11 Document History

Version	Date	History
001	October 21, 2009	Creation
002	February 18, 2010	Updated
3.0	March 12, 2012	Updated legal boilerplate contents. New reference: 2170026 Old reference: WM_DEV_OAT_APN_019 Updated the exception for Q26XX.

12 Legal Notice

Important Notice

Due to the nature of wireless communications, transmission and reception of data can never be guaranteed. Data may be delayed, corrupted (i.e., have errors) or be totally lost. Although significant delays or losses of data are rare when wireless devices such as the Sierra Wireless modem are used in a normal manner with a well-constructed network, the Sierra Wireless modem should not be used in situations where failure to transmit or receive data could result in damage of any kind to the user or any other party, including but not limited to personal injury, death, or loss of property. Sierra Wireless accepts no responsibility for damages of any kind resulting from delays or errors in data transmitted or received using the Sierra Wireless modem, or for failure of the Sierra Wireless modem to transmit or receive such data.

Safety and Hazards

Do not operate the Sierra Wireless modem in areas where blasting is in progress, where explosive atmospheres may be present, near medical equipment, near life support equipment, or any equipment which may be susceptible to any form of radio interference. In such areas, the Sierra Wireless modem **MUST BE POWERED OFF**. The Sierra Wireless modem can transmit signals that could interfere with this equipment. Do not operate the Sierra Wireless modem in any aircraft, whether the aircraft is on the ground or in flight. In aircraft, the Sierra Wireless modem **MUST BE POWERED OFF**. When operating, the Sierra Wireless modem can transmit signals that could interfere with various onboard systems.

Note: Some airlines may permit the use of cellular phones while the aircraft is on the ground and the door is open. Sierra Wireless modems may be used at this time.

The driver or operator of any vehicle should not operate the Sierra Wireless modem while in control of a vehicle. Doing so will detract from the driver or operator's control and operation of that vehicle. In some states and provinces, operating such communications devices while in control of a vehicle is an offence.

Limitations of Liability

This manual is provided "as is". Sierra Wireless makes no warranties of any kind, either expressed or implied, including any implied warranties of merchantability, fitness for a particular purpose, or noninfringement. The recipient of the manual shall endorse all risks arising from its use.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Sierra Wireless. SIERRA WIRELESS AND ITS AFFILIATES SPECIFICALLY DISCLAIM LIABILITY FOR ANY AND ALL DIRECT, INDIRECT, SPECIAL, GENERAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES INCLUDING, BUT NOT LIMITED TO, LOSS OF PROFITS OR REVENUE OR ANTICIPATED PROFITS OR REVENUE ARISING OUT OF THE USE OR INABILITY TO USE ANY SIERRA WIRELESS PRODUCT, EVEN IF SIERRA WIRELESS AND/OR ITS AFFILIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR THEY ARE FORESEEABLE OR FOR CLAIMS BY ANY THIRD PARTY.

Notwithstanding the foregoing, in no event shall Sierra Wireless and/or its affiliates aggregate liability arising under or in connection with the Sierra Wireless product, regardless of the number of events, occurrences, or claims giving rise to liability, be in excess of the price paid by the purchaser for the Sierra Wireless product.

Patents

This product may contain technology developed by or for Sierra Wireless Inc.

This product includes technology licensed from QUALCOMM®.

This product is manufactured or sold by Sierra Wireless Inc. or its affiliates under one or more patents licensed from InterDigital Group.

Copyright

Trademarks

AirCard® is a registered trademark of Sierra Wireless. Sierra Wireless™, AirPrime™, AirLink™, AirVantage™, Watcher™ and the Sierra Wireless logo are trademarks of Sierra Wireless.

wavecom®, , , ®, inSIM®, WAVECOM®, WISMO®, Wireless Microprocessor®, Wireless CPU®, Open AT® are filed or registered trademarks of Sierra Wireless S.A. in France and/or in other countries.

Windows® and Windows Vista® are registered trademarks of Microsoft Corporation.

Macintosh and Mac OS are registered trademarks of Apple Inc., registered in the U.S. and other countries.

QUALCOMM® is a registered trademark of QUALCOMM Incorporated. Used under license.

Other trademarks are the property of the respective owners.