Using the Hardware Serial Peripheral Interface (SPI) and Neurowire I/O Object Models to Interface with Peripherals and Microcontrollers

April 2004 LONWORKS® Engineering Bulletin

Introduction

When connecting the Smart Transceivers and Neuron® Chips to peripherals or other microcontrollers in a LONWORKS device, it is desirable to make use of serial interfaces that are readily available in many components and microcontroller architectures. One such serial interface is called the Serial Peripheral Interface or SPI (developed by Motorola) that offers synchronous, bidirectional, and fast data transfers over a serial link.

Applications for the SPI interface range from controlling simple shift registers to communications between microcontrollers. While this engineering bulletin will address the latter, (communications between Smart Transceivers or Neuron Chips and other microcontrollers) it is still useful and recommended reading for designs that intend to make use of the SPI interface for simpler applications and also for understanding how the SPI interfaces are implemented in the Smart Transceivers and Neuron Chips.
Neuron C code examples of an SPI interface are explained in this engineering bulletin, and the source code is available for download.

Background

Neuron Chips, from the beginning, have incorporated the Neurowire I/O object model, a software implementation of the SPI interface, supporting both master and slave configurations with data rates up to 20kbps. However, since the Neurowire is a driver implemented in the software (as a library), the Neuron Chip needs to dedicate the entire processing power of its application processor to SPI transfers while they are taking place. This means that no other task can be executed until the whole SPI transfer is complete.

With the release of the PL 31x0 Smart Transceivers in 2003, a new I/O object model was created to increase the performance of the Neurowire I/O object model. The new model is called SPI and is supported by a hardware implementation of the SPI interface plus an interrupt service routine that can be triggered (and this happens automatically) even when the application processor is performing other tasks. Refer to the Neuron Chip and Smart Transceiver documentation to find out which members support the new SPI I/O object model. At the time of release, the following Neuron Chips and Smart Transceivers support the SPI I/O object model.

Neuron Chip / Smart Transceiver Model	Manufacturer
PL 3120® Power Line Smart Transceiver	Echelon
PL 3150® Power Line Smart Transceiver	Echelon
CY7C53120L Neuron Chip	Cypress
CY7C53150L Neuron Chip	Cypress

Overview of the SPI Interface

Master – Slave Operation

The SPI interface provides bi-directional, synchronous serial communications between microcontrollers and peripherals. It is based upon a master-slave protocol where the master is the device that drives the clock signal. Multiple masters and multiple slaves are allowed on the bus; a simple example, shown below, contains a single master and two slave peripherals. The slave select signals are driven by the master, so that only one slave is accessed at any given time.

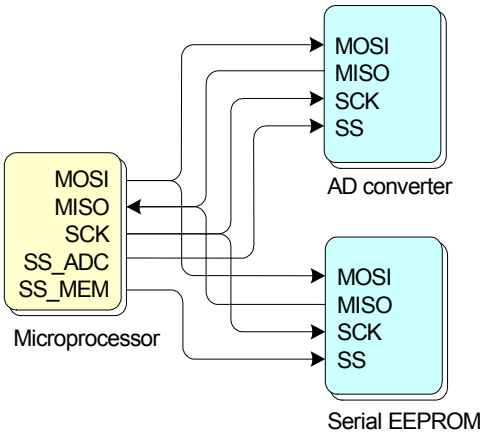

Figure 1. One Master and Two Slave Peripherals

The standard naming convention used for the SPI signals follows:

MOSI	Master Out - Slave In.
MISO	Master In - Slave Out.
SS	Slave Select signal (our example uses two of those, SS_ADC and SS_MEM)
SCK	Serial Clock (always driven by the Master)

The transfer of data using an SPI interface can be thought of as a large shift register shared between the master and slave devices. Data is clocked IN at the same time it is clocked OUT of the devices (the clock is shared by the two devices).

Figure 2 shows a high level view of how the internal SPI registers work.

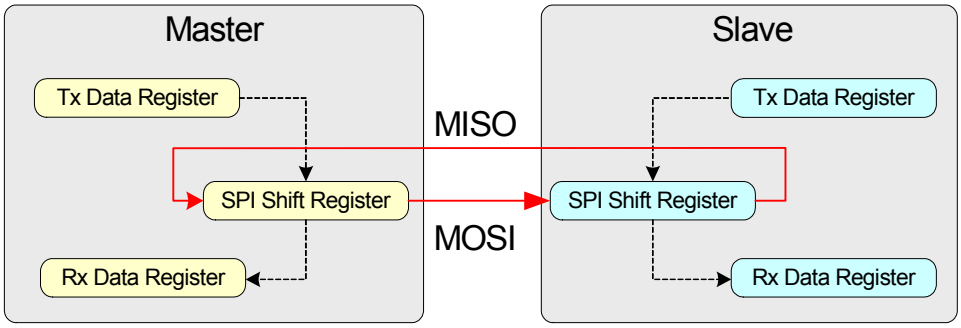

Figure 2. Internal SPI Registers

Clock Polarity and Clock Phase

The SPI interface requires the master and slave devices to be in agreement regarding the idle state of the clock signal as well as the way in which the data will be clocked during the SPI transfer.

The Clock Polarity (**CPOL**) parameter indicates the level of the clock signal when it is idle.

CPOL=0 Clock idle state is low.

CPOL=1 Clock idle state is high.

The Clock Phase (**CPHA**) parameter indicates when the data should be presented and when it should be sampled.

CPHA=0 The first edge on the SCK line is used to sample the first data bit and the second edge is used to present it (the first bit of the data must be ready before the first edge of the SCK line).

CPHA=1 The first edge on the SCK line is used to present the data and the second edge to sample it.

The following two diagrams show the different combinations of clock polarity and clock phase settings.

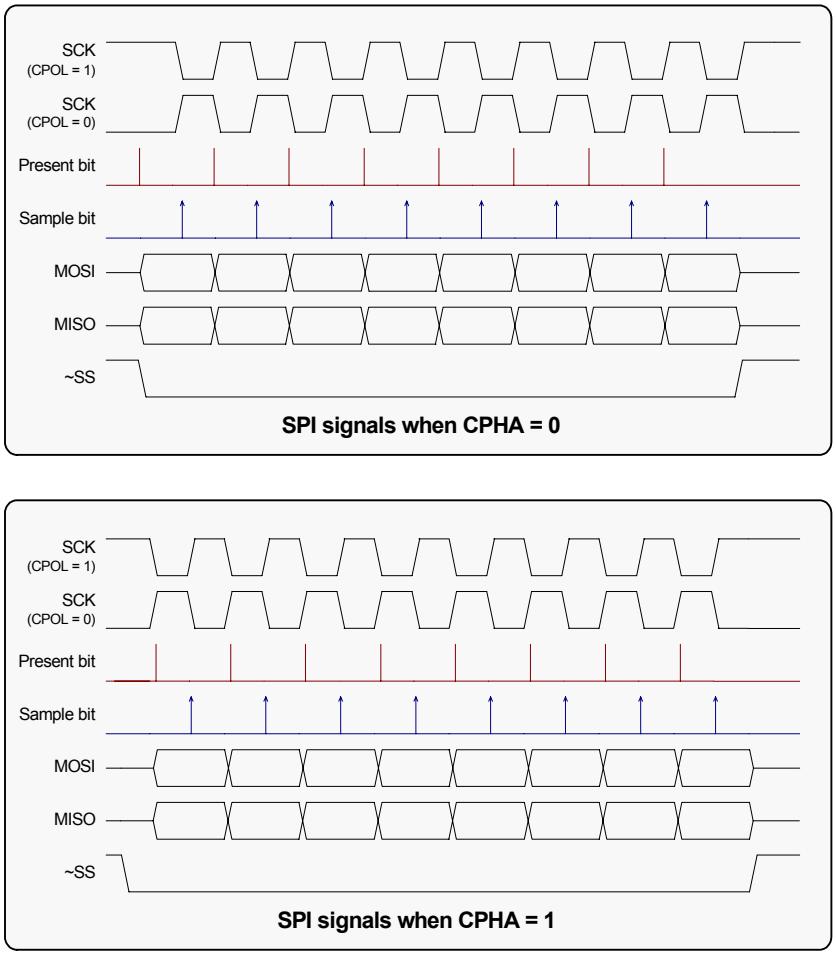

Figure 3. Clock Polarity and Clock Phase Settings

Neurowire and SPI I/O Object Models

The *Neuron C Programmer's Guide*, *Neuron C Reference Guide*, *FT 3120/FT 3150 Smart Transceiver Data Book*, and PL *3120/PL 3150 Smart Transceiver Data Book*, available at www.echelon.com, provide detailed explanations of the way both of these object models are implemented and how they should be used. The most current copy of these documents is always available at the Echelon web site.

Both I/O object models support master and slave operation, however, when used in the master mode, no other masters are allowed to coexist in the SPI bus, i.e. multiple master applications are not supported.

The following section shows how both I/O object models relate to the standard SPI interface. Differences between the two models are provided in order to best explain how these models should be used.

Neurowire I/O Object Model

The Neurowire I/O object model is available in all Smart Transceiver and Neuron Chip models, supporting both **master** and **slave** modes.

In Neurowire **master** mode, one or more of the pins IO_0 through IO_7 and IO_11 (if available) may be used as slave select signals driven by the Neuron C application, allowing multiple Neurowire devices to be connected on a 3-wire bus. When only one slave device is used, the *select( )* modifier can be used to define the slave signal, thus allowing the Neurowire object to drive the signal during data transfer. Also in master mode, the clock rate may be specified as 1, 10, or 20kbps at a Neuron Chip or Smart Transceiver input clock rate of 10MHz; these rates scale proportionally with the input clock speed.

In Neurowire **slave** mode, one of the pins IO_0 through IO_7 may be designated as a timeout pin. A logic one level on the timeout pin causes the Neurowire slave I/O operation to be terminated before the specified number of bits has been transferred. This prevents the Neuron Chip or Smart Transceiver watchdog timer from resetting the chip in the event that fewer than the requested number of bits is transferred by the master.

The Neurowire I/O object model supports both polarities CPOL=0 and CPOL=1, but only supports phase CPHA=1. Polarity is controlled by the *clockedge* modifier as follows.

CPOL:
1 = *clockedge*(-)
0 = *clockedge*(+)

In both master and slave modes, up to 255 bits of data may be transferred at a time. Because the Neurowire I/O object model is implemented in software, function calls associated with the Neurowire I/O object model suspend any other application processing until the Neurowire transfer is complete. This does not affect the Network and Media Access Control processors, which run independently. However, occupying the application processor for the entire duration of a Neurowire transfer, especially when transferring large data amounts using slow Neurowire clock rates, could degrade the overall device performance and cause application input buffer congestion and, in severe cases, loss of packets arriving over the network.

SPI I/O Object Model

Newer models of Smart Transceivers and Neuron Chips incorporate a more efficient SPI interface called the SPI I/O object model, allowing for the sending and/or receiving of up to 255 bytes of data in a single transfer (note that the Neurowire object only allows up to

255 bits). This implementation of the SPI interface is supported by hardware with dedicated registers and an interrupt service routine that handles the SPI transfers.

In master mode, the application is responsible for driving the slave select signal(s) to select an external slave. It is normally a good idea to drive the slave select low only when data is actually being transferred, even when there is only one external slave. This allows the slave to recognize the end of a packet transfer, which provides for better error detection and correction if the master and slave get out of synchronization.

In slave mode, the hardware automatically recognizes the assertion of the slave select pin (on IO7).

The SPI I/O object model supports all four modes of operation regarding ways the clock signal can be used. These combinations are accomplished by using the *clockedge* and *invert* Neuron C modifiers as follows.

CPHA:
1 = *clockedge(+)*
0 = *clockedge(-)*

CPOL:
1 = [default]
0 = *invert*

Note that the semantics of the *clockedge* keyword are different for the two models. For the Neurowire model, the *clockedge* modifier changes the idle state of the clock, however, for the SPI I/O object model, the *clockedge* modifier selects the clock phase.

Because the SPI I/O object model is interrupt driven, there are certain considerations the application developer should understand.

When calling *io_out( )* or *io_in( )* (which are identical for SPI), the firmware will set up the buffer, the transmit, and the receive counts and pointers, which initially are the same but are maintained separately. These buffer, counters, and pointers are all maintained and used by the SPI interrupt service routine when transferring data. As soon as *io_out()* or *io_in()* is called, the SPI transmitter shift and data registers should be empty, therefore the first byte in the buffer will be transferred into the SPI transmitter data register immediately. Without any master clock present, the second byte (if the count is greater than 1) will be transferred into the SPI transmitter data register under interrupt control, since the first byte would have already been moved into the SPI transmitter shift register. At this point the transmit pointer is already two bytes ahead of the receive pointer.

The implication is that, immediately following the *io_out( )* function call, as many as two bytes of data may have already moved out of the buffer and into the SPI transmit hardware. This means that a transfer length of only (2) could appear to the application as complete, even though the transfer may not have started yet.

Another consideration concerns a case where an SPI transfer has been set up by the slave mode for some large data transfer size but with the intent of truncating the transfer based on information that is part of the transferred data itself. For example, if the slave prepares for a transfer of 100 bytes and there is a need to stop the transfer after the first 50 bytes, the interrupt driven firmware routine might have already placed bytes 51 and 52 into the SPI transmit hardware.

Resetting the SPI object prior to re-use may therefore be necessary. There are two levels of resetting that should be used. First, there is a reset of the counters and pointers maintained by the ISR routine. Resetting these variables is accomplished by calling *io_out( )* with a count of 0, for example:

io_out (spiSlave, &spiBuffer, 0);

The second level of reset is a reset of the SPI hardware registers. Use the *io_set_clock( )* function with the same parameters that were used when the SPI object was declared, for example:|

io_set_clock (spiSlave, 0, clockedge(+), invert);

Another consideration is that the *io_out_ready()* and *io_in_ready()* functions are not interchangeable. These functions may seem more or less identical in purpose due to the fact that the input and output operations are synchronized. However, in reality the *io_out_ready( )* function will return true before *io_in_ready( )* returns a count that is equal to the expected count. This, again, is due to the fact that the transmit data and shift registers hold two bytes of data and complete the transmit process before the receive process has completed.

This last consideration comes into play depending on what you do in the Neuron C application immediately following the completion of the transfer. Since both transmit and receive are synchronized in SPI, it is best to use *io_in_ready( )* to qualify the completion of the SPI transfer.

The following diagram shows the interaction between the SPI hardware registers (transmit and receive) with the interrupt service routine.

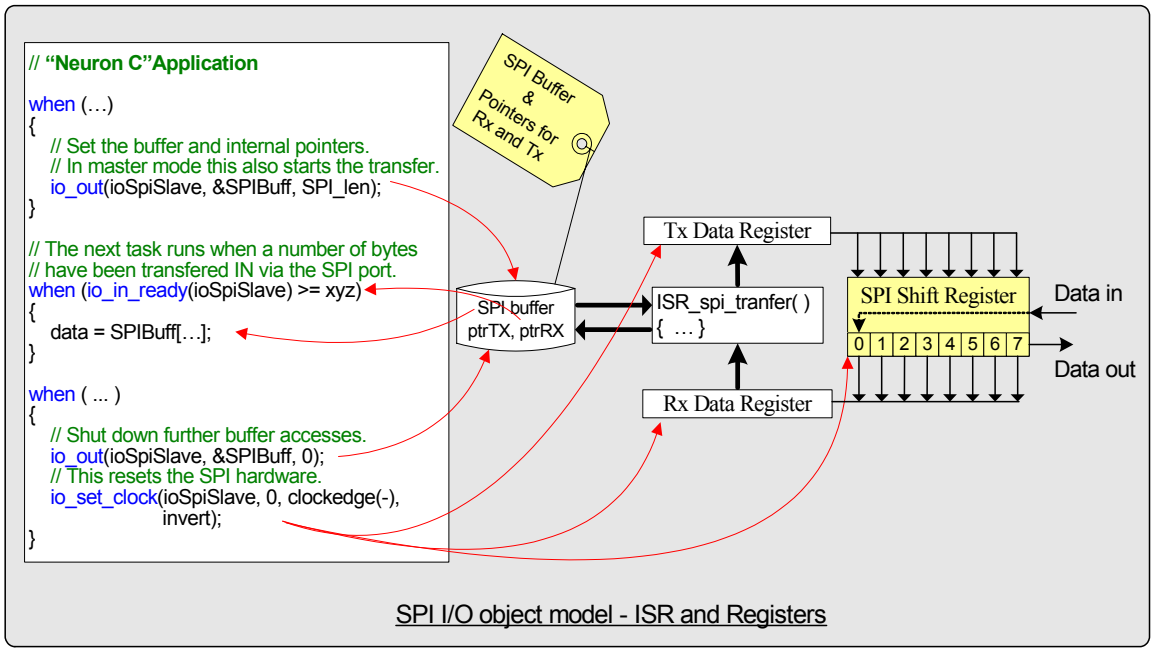

Figure 4. Internal SPI interactions

Using the SPI Interfaces for Communications with Other Microcontrollers

Most Smart Transceivers and Neuron Chip-based designs that use one of the SPI interfaces (Neurowire or SPI I/O object models) are meant to communicate with slave peripherals such as A/D converters, serial EEPROM memory, real time clocks, and etcetera. These kinds of applications are simple and straight forward; the Smart Transceiver or Neuron Chip is always the SPI master (driving the clock signal) and the different peripherals are the SPI slaves. The master device normally initiates the SPI transfers and the number of bytes to be transferred is predetermined by the technical specifications of the peripherals, as well as the clock polarity and phase.

However, when interfacing the Smart Transceiver or Neuron Chip to another microcontroller, implementation might prove to be complex. For this task, the following requirements should be addressed:

- polarity and phase of the clocks need to be agreed to by both devices;
- bi-directional communication may be required whereby the slave device can initiate an SPI transfer;
- data amounts may vary with every transfer; and
- SPI communication is just one of the many tasks both microcontrollers need to execute.

To create an SPI interface that satisfies these requirements, a carefully-defined communications protocol must be used.

Defining an SPI Protocol

In order to define an SPI Protocol, you need to understand the different requirements of the SPI interface and determine the appropriate resources needed to satisfy them.

The example provided demonstrates communications between a Neuron Chip-based device using the Neurowire I/O object model (the SPI master) and a Power Line Smart Transceiver device that uses an SPI I/O object model (the SPI slave). This example can easily be ported to other microcontrollers that support the SPI interface in either master or slave mode.

a) **Clock Polarity and Phase.** As noted in the *SPI overview*, it is crucial for correct operation of the SPI interface to have both master and slave use the same clock polarity and phase.

   In the example protocol, the clock phase needs to be set to CPHA=1 because it is the only phase supported by the Neurowire I/O object model. Polarity comes next. For this example, clock polarity CPOL=0 was arbitrarily chosen.

b) **Size of the data to be transferred is not fixed.** As the information to be transferred varies, so will its size. A mechanism is needed so that both master and slave share knowledge of the expected data amount.

   One common way to do this is to send the size of the packet with the first byte of the stream, and then have the devices count the rest of the bytes as they arrive. While this approach seems logical, it presents problems because it assumes that the devices have enough time to process the first byte and to keep counting the rest of the bytes as they arrive. This could be possible in some cases, but overlooks the other tasks, besides the SPI interface, that the devices need to perform. Therefore, it is recommended that a defined fixed packet size should always be transferred, even if the actual size of the data is smaller. The packet can then be examined to determine the actual size of the data.

   The total amount of data for a transfer, however, may also exceed the fixed packet size. There are two ways to deal with this problem. The first and obvious one is to make the fixed size larger so that any type of data would fit in a single packet. This solution is not scalable (e.g., the maximum number of bits for a Neurowire transfer is 255 bits, the fixed size could not be larger than 31 bytes) and can waste processing time if the size of the data is oftentimes smaller than the fixed size packet. A second approach is to split the transmission into two or more SPI

transfers. In this approach, the first transfer would always have a fixed size, but it would carry information about the total length of the data. Based on this information, a second, third or more SPI transfers can be performed to complete the data exchange.

This example SPI protocol uses the term **Transaction** to refer to the entire process where one or more **SPI transfers** take place. Each transaction is limited to a maximum of two SPI transfers (this could be customized to allow for more than 2 transfers) where the first one has a fixed size but the second one has a variable size determined by information embedded in the first transfer. Both SPI transfers are limited in size to 31 bytes because Neurowire transfers (used by our master) can only contain a maximum of 255 bits.

Another way to deal with data transfers of unknown size is by monitoring the slave select signal (if one is used) for falling and then rising edges. To do this, *level detect* and *bit input* I/O object models can be used in conjunction. This example SPI protocol will use the fixed size data transfer scheme explained above.

c) **The Slave should be able to initiate an SPI transaction**. In simple SPI implementations, the master device is the one that starts all SPI transfers. Even if the slave has some information to share, it will wait until the SPI master polls it for this data.

This example protocol will incorporate a special signal called HREQ (Host request) used by the slave to prompt the master whenever it has information to share.

d) **SPI is just one of the tasks a microcontroller executes.** Microcontrollers have a broad range of peripherals other than the SPI interface; those peripherals and other control algorithms also need to be handled and executed by the same CPU in a coordinated manner so that proper attention is paid to each of them.
For example, a Smart Transceiver may be executing a routine that requires precise timing and nothing should interrupt it, not even the SPI interface.

This requirement translates to a term that you may be familiar with, “Handshaking.”

This example SPI protocol will include two extra signals for handshaking, SREQ (Slave Request) and SRDY (Slave Ready). SREQ is driven by the master to signal the slave that a new SPI transfer is going to start. The slave will then sense SREQ and, once it sees it asserted, (the handshake signals will be asserted in low level) it will get ready to receive the stream of data and will assert SRDY to let the master know that the transfer can start. Similarly, when the slave needs to relay information to the master, it will assert the HREQ signal and the master will drive the transaction from that point.

Note that the handshaking happens for every SPI transfer; if the entire SPI transaction requires two SPI transfers, the handshaking will happen once for each transfer. For reference, some timing diagrams (created using a logic analyzer) are included in this document.

e) **Protocol glossary.** Here are the other terms used in the implementation of this just-defined SPI communications protocol.

Signal Assertion. Used when a handshake signal becomes active, in this case, assertion will happen when the signal goes from high to low level.

Downlink message. A message sent by the master to the slave.

Uplink message. A message sent by the slave to the master.

Half Duplex transaction. When there is only a downlink or an uplink message during a single transaction.

Full Duplex transaction. When there are both downlink and an uplink messages in a single transaction. This is supported by the SPI interface and is also fully supported in this example protocol.

f) **Master and Slave state diagrams.** The following state diagrams (Petri Nets) show how our SPI protocol works on both the master and slave sides.

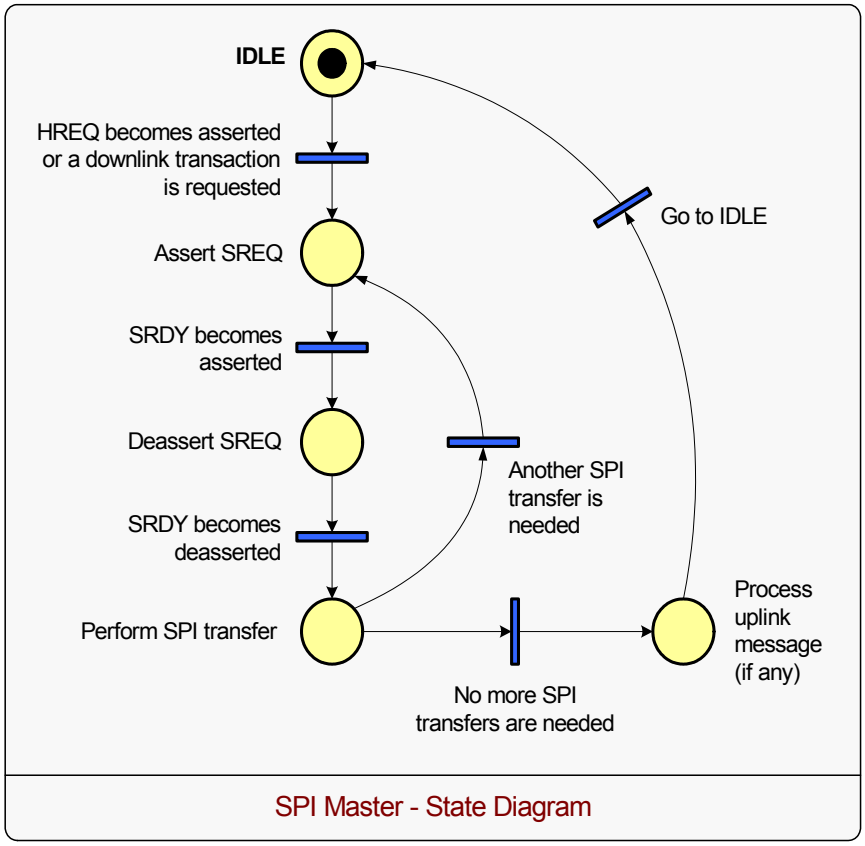

Figure 5. SPI Master - State Diagram

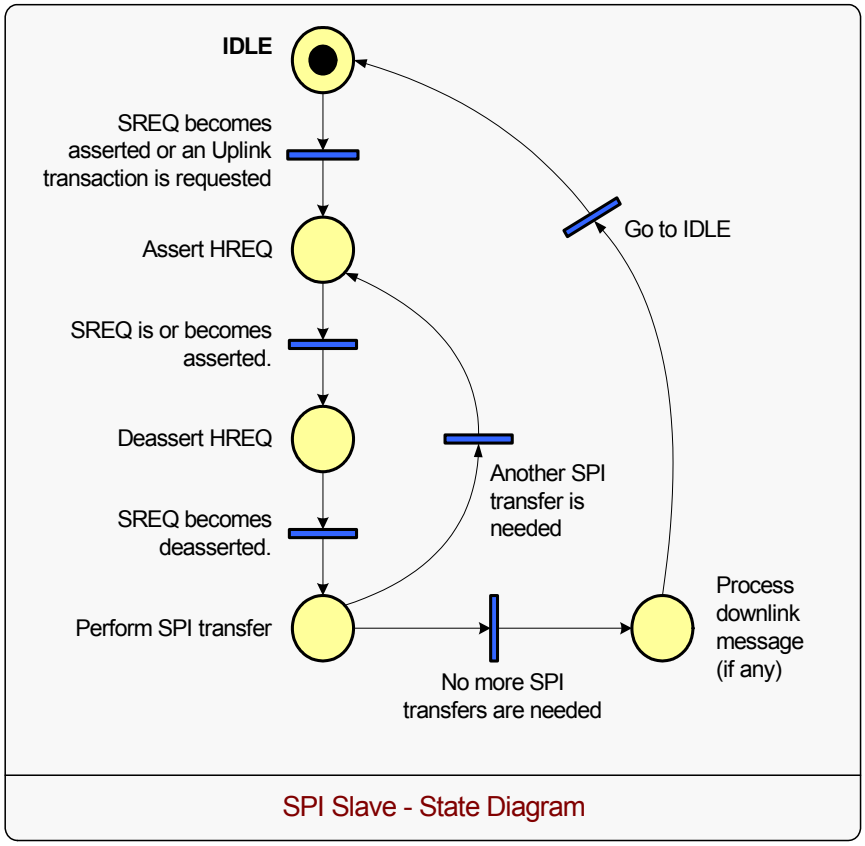

Figure 6. SPI Slave – State Diagram

g) **How does the protocol look on a Logic Analyzer?** The following are screen captures from a Logic Analyzer. You can use these when troubleshooting your implementation of the protocol.

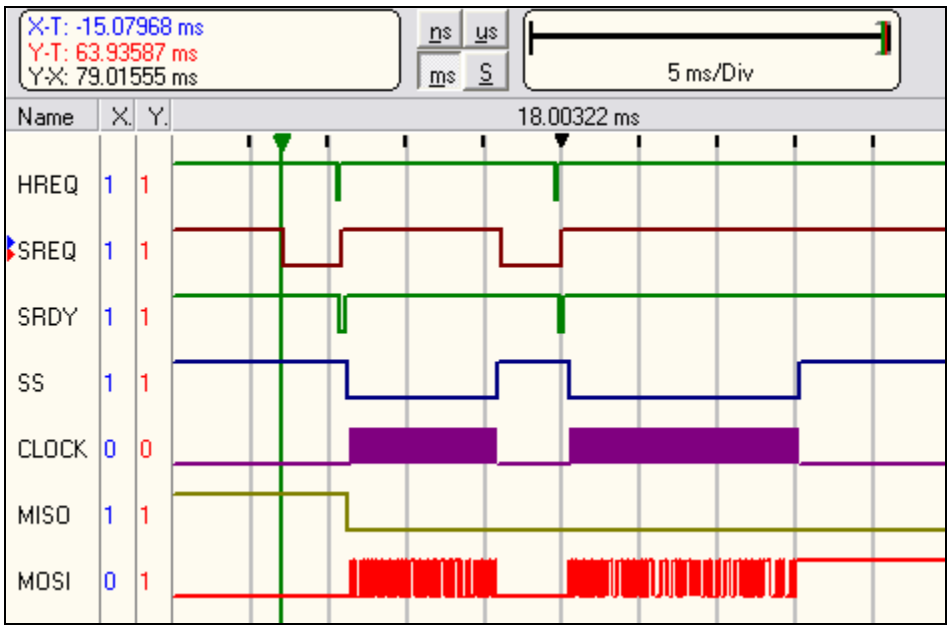

Figure 7. SPI Transaction (two SPI transfers) Initiated by the Master

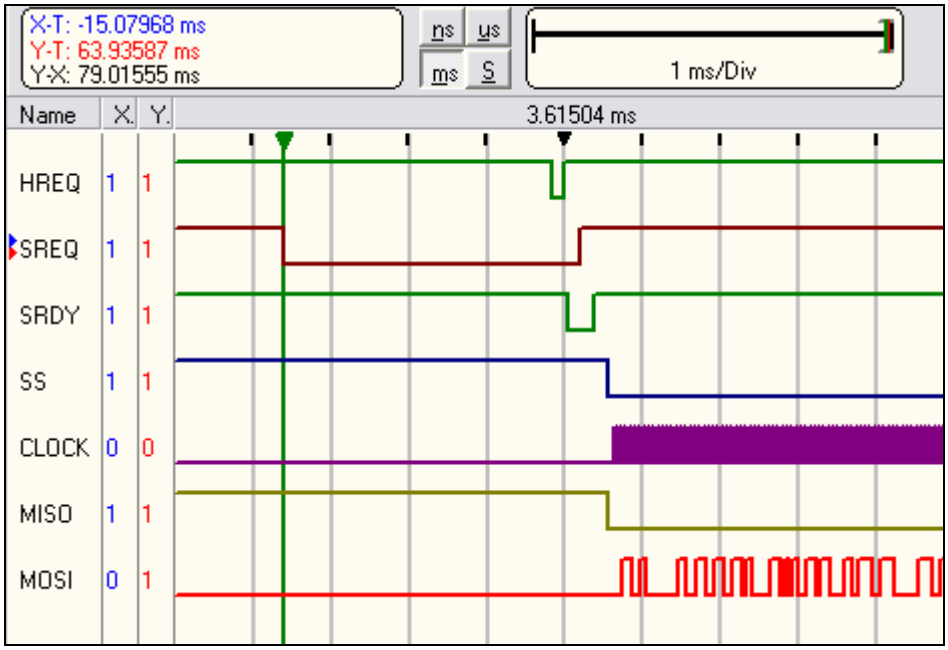

Figure 8. Handshake Signals (for both First and Second SPI Transfers)

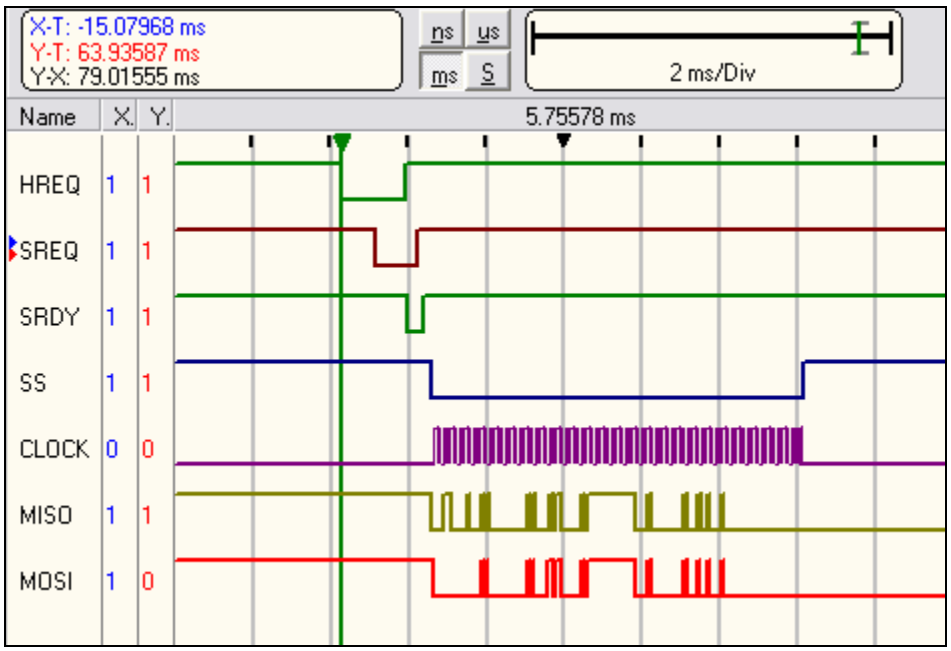

Figure 9. SPI Transaction (Single SPI transfer) Initiated by the Slave

Neuron C Implementation of the Just Defined SPI-based Communications Protocol

The following implementation can also be downloaded from Echelon's website at www.echelon.com/downloads. Note that you will need NodeBuilder 3.1 or newer to build the project.

The hardware used in this project is:

- An LTM-10A device (model 65150-2xx available from Echelon) as the SPI master, containing a Neuron 3150 Chip and plenty of FLASH and RAM memory for development purposes.
- A development board based on the PL 3150 Power Line Smart Transceiver as the SPI slave.

Note that the Neuron 3150 Chip used in the LTM-10A device can only implement the Neurowire I/O object model while the PL 3150 Power Line Smart Transceiver also supports an improved SPI interface called the SPI I/O object model; this improved interface is used in the example implementation.

Both devices are connected via the I/O pins as shown in the following diagram.

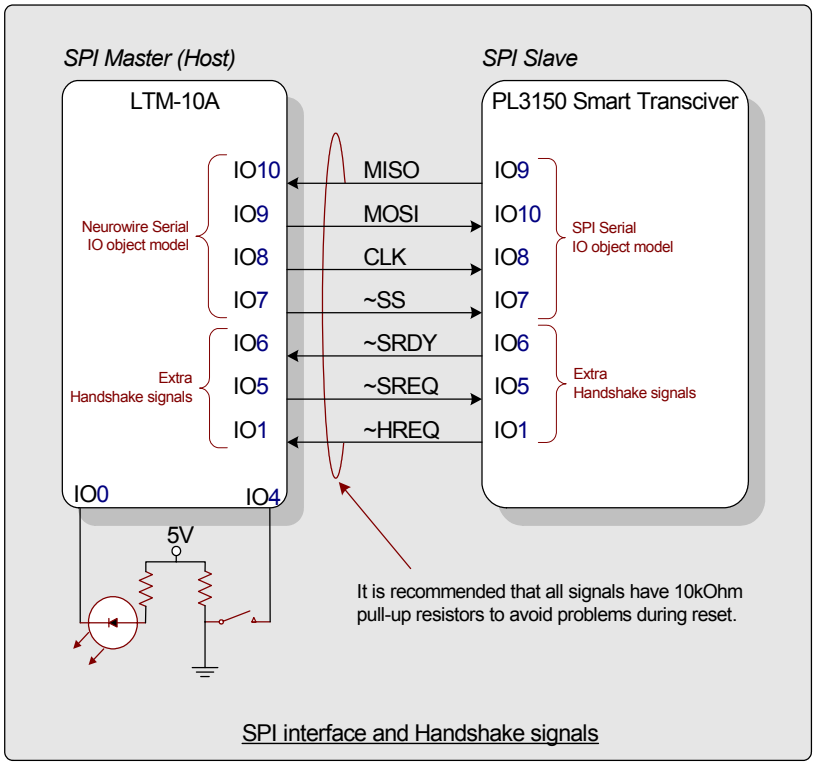

Figure 10. LMT-10A and PL 3150 Smart Transceiver Connections

A backup copy of the NodeBuilder 3.1 project called "SPI Eng Bulletin.zip" can be downloaded from www.echelon.com/downloads.

Part Number 005-0165-01 Rev. A
Echelon Corporation
www.echelon.com