



# Interfacing Neuron<sup>®</sup> and PSoC<sup>™</sup> Devices Using SPI

## Introduction

Neuron<sup>®</sup> processors offer a variety of SW emulated interface options that can be used to communicate with other processors. This application note covers the issues related to configuring the Neuron as a SPI master connected to one or more PSoC<sup>™</sup> SPI slaves. The assumption is made that the reader is somewhat familiar with developing in both environments.

For more information regarding Neuron devices, visit the Cypress Semiconductor web site at <http://www.cypress.com>. Information regarding PSoC can be found at the Cypress MicroSystems web site at <http://www.cypressmicro.com>.

## Configuring the Neuron SPI Master Device

One of the emulated serial interfaces supported by Neuron is the neurowire I/O type. A neurowire I/O type is compatible with SPI mode 2 by default and supports full duplex communication with up to four slaves. The interface consists of the following I/O signals.

Signal	In/Out	IO Pin	Definition
SCLK	Out	IO8	Bit Clock
MOSI	Out	IO9	Serial Data Out
MISO	In	IO10	Serial Data In
~SSEL	Out	IO0-7	Slave Select

The following statements are added to the body of your code to instantiate the neurowire I/O type.

```
IO_8 neurowire master select (IO_0) PSOC1Select [1,5,10];
IO_0 output bit PSOC1_sel = 1;
```

The first line defines that you are selecting the neurowire Master I/O type and that IO\_0 is used as the active low slave select signal ~SSEL. Optionally, the bit rate can be specified as 1,5, or 10 Kbps for a 10-MHz crystal. The second line defines IO\_0 as an output bit and initializes it to 1. This serves to reserve IO\_0 for use by the neurowire function. The function calls to transmit and receive are as follows.

```
io_out(PSOC1Sel, &TX_Data, #bits);
io_in(PSOC1Sel, &RX_Data, #bits);
```

The neurowire interface is full duplex and there is no difference between the io\_out and io\_in functions; either one can be used. The first parameter in the function defines which slave select I/O to use. The second parameter is a pointer to your data structure. As TX data is shifted out, RX data will be shifted into the same data structure. The third parameter defines the total number of bits, up to 255, that will be shifted out. The slave select line ~SSEL will be asserted low during

the entire transfer. If your application requires that slave select toggle between bytes then you must issue an io\_out call for each byte.

## Configuring PSoC as an SPI Slave

With the PSoC development tool, select and place the SPIS module. In this example, digital communications block DCA04 was chosen, as shown in *Figure 1*. The SPIS module can be placed in any of the four digital communications blocks.

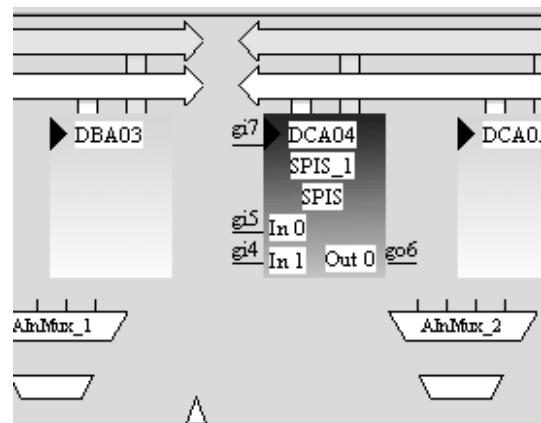


Figure 1.

The SPIS user module parameters are assigned to the global signals as shown in *Figure 2*.

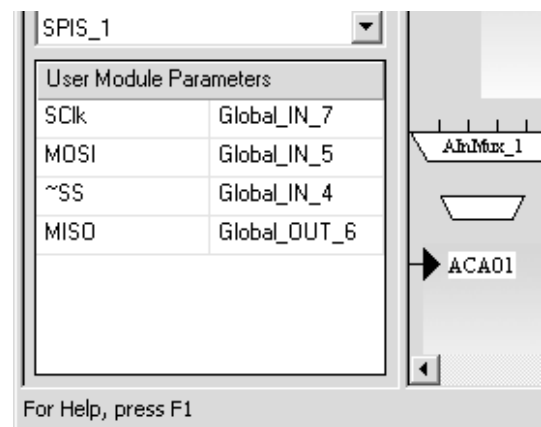


Figure 2.

Finally the global signals need to be connected to the device I/O pins (see *Figure 3*).

Name	Port	Select	Drive	Interrupt
Port_1_0	P1[0]	StdCPU	Pull Down	DisableInt
Port_1_1	P1[1]	StdCPU	Pull Down	DisableInt
Port_1_2	P1[2]	StdCPU	Pull Down	DisableInt
Port_1_3	P1[3]	StdCPU	Pull Down	DisableInt
SPISel	P1[4]	Global_IN_4	High Z	DisableInt
SDI	P1[5]	Global_IN_5	High Z	DisableInt
SDO	P1[6]	Global_OUT_6(S	Strong	DisableInt
SCLK	P1[7]	Global_IN_7	High Z	DisableInt
Port_2_0	P2[0]	StdCPU	Pull Down	DisableInt

For Help, press F1

**Figure 3.**

The SPIS needs to be initialized in order to make it operational. The following C statements must be executed before the module can be used.

```
SPIS_1_Start(SPIS_MODE_2);    // Set mode 2
SPIS_1_EnableInt();          // Enable SPIS ints
M8C_EnableGInt;              // Enable global ints
```

The SPI module consists of a Shift register, Transmit register, and Receive register. In SPIS mode 2, the contents of the TX register will be transferred into the shift register on the leading edge of the first clock. If your application requires slave data to be transmitted on the first byte then you will need to ensure that the TX register is loaded prior to the first bit clock. As the eighth bit is transferred, the contents of the shift register will be transferred into the RX register and a transmit interrupt will be generated. SPI interrupt code must read the contents of the RX register and place the next byte in the TX register. Care must be taken to ensure that the next byte is loaded into the TX register before the first clock of the next byte. The code must also read the status register in order to prime the hardware for the next SPI transfer. If your application does not have any data to transmit then you can write 0x00 into the transmit register, otherwise the last byte received will be transmitted.

If multiple SPI slaves are being used then care must be taken to three-state the MISO signal so that only the slave being addressed by ~SSEL drives the data line.

If your preference is to code in C then the following steps need to be completed:

```
#pragma interrupt_handler SPIInterrupt .
```

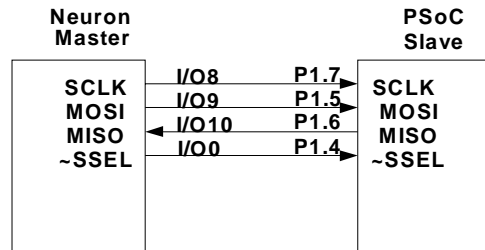
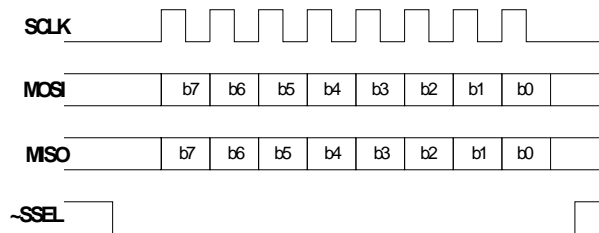
This #pragma statement must be placed just before the function definition. Also, boot.tpl must be modified so that the function name is placed in the vector location for the corresponding digital block as follows:

```
org 18h    ;PSoC Block DCA04 Interrupt Vector
`ljmp _SPIInterrupt`
reti .
```

Note that 18h is the interrupt vector location for DCA04.

## Putting It All Together

The SPI interface is now configured as shown in *Figure 4* and will generate the single byte transfer as shown in *Figure 5*.


**Figure 4.**

**Figure 5.**

As a general guideline, care should be taken when considering the use of SCLK to ensure that there are no false edges, which can cause erratic behavior in the slave.

## Code Examples

### Neuron code showing how to send and receive SPI data

```
1
IO_8 neurowire master select (IO_0) PSOC1Select;
IO_0 output bit PSOC1_sel = 1;
{
    unsigned char Data[3] = {0x55,0xAA,0x12};

// Do SPI transfer
    io_out(PSOC1Select,Data,24);

// Now process received data, Data[] contains
// any bytes received from the slave
    if(Data[0] == 13)    // as an example
}
}
```

### On the PSoC side, using circular buffers to Receive and send data:

```
1 #define SPI_BUF_SIZE0x0F

// Define variables
unsigned char    u8SPIRxBuffer[SPI_BUF_SIZE +1];
unsigned char    u8SPIRxHead;
unsigned char    u8SPITxBuffer[SPI_BUF_SIZE +1];
unsigned char    u8SPITxTail;

void SPIInterrupt( void )                // SPI interrupt service routine
{
    unsigned char u8x,u8y;

    u8x = bSPIS_1_ReadRxData();           // Get the received data and rearm the interrupt
    u8y = SPIS_1_CONTROL_REG;             // Reading control reg. will enable an interrupt when the
                                           // next byte is received.

// If there is a byte to send, put it in the tx reg prior to the next byte

if (gu8SPITxHead != u8SPITxTail)
{
    SPIS_1_TX_BUFFER_REG = u8SPITxBuffer[gu8SPITxTail];           // Write to the register
    u8SPITxTail++;
    u8SPITxTail &= SPI_BUF_SIZE;                                   // Wrap the pointer (32 bytes
deep)
}
else
{
    // Just put 0 in the TX reg
    SPIS_1_TX_BUFFER_REG = 0;
}

// Put the data just received in the Rx buffer
u8SPIRxBuffer[gu8SPIRxHead] = u8x;
u8SPIRxHead++;
u8SPIRxHead &= SPI_BUF_SIZE;           // Wrap the pointer
}
}
```

Neuron is a registered trademark of Echelon Corporation. Programmable System-on-Chip and PSoC are trademarks of Cypress MicroSystems. All product and company names mentioned in this document are the trademarks of their respective holders.

Approved AN025 5/14/03 kkv