AN2165

PSoC® 1 Implementation of a Direct Sequence Spread Spectrum Baseband Transmitter

Author: Kristopher Young

Associated Project: Yes
Associated Part Family: CY8C27xxx
Software Version: PSoC Designer™
Related Application Notes: AN2094, AN2104

This applications note presents an example design for implementing the baseband functions of a Direct Sequence Spread Spectrum (DSSS) transmitter with the PSoC® 1 programmable system-on-chip. A detailed description of the implementation and a brief DSSS primer are also included.

Introduction

Spread spectrum communication has come a long way since it was first conceived in 1941 by Hollywood actress Hedy Lamarr and avant-garde pianist George Antheil (US Patent #2,292,387: Secret Communications System). These days, it is used in several areas including cellular phones, wireless LANs, satellite communication, and even gas and water meters. The capabilities of the digital blocks in the PSoC are well suited for many of the baseband functions required to transmit a spread spectrum signal, providing a low-cost and highly integrated solution with many additional features.

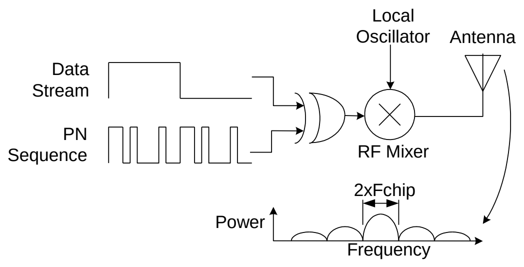
Spread Spectrum Primer

Spread spectrum RF transmission involves spreading out the power spectrum of an RF signal using a code. The same code is then used to receive the signal. This has tremendous benefits, including shared bandwidth between multiple users using multiple codes, reduction of interference experienced by other users of the spectrum, and resistance to external interference, jamming, multipath effects, and eavesdropping.

Spread spectrum usually involves changing the transmission frequency of a signal over time (frequency hopping) or combining a signal with a Pseudo-random Noise (PN) code before transmission (direct sequence or DS). Spread spectrum can also be implemented by a combination of the two. This discussion will be limited to implementing the transmission of DS spread spectra.

In DS Spread Spectrum (DSSS) transmission, a PN sequence is XOR combined with a data stream before it is mixed with RF and then transmitted. Figure 1 shows the basic block diagram of this process. The resulting signal contains the spectral characteristics of the PN sequence. The bits that make up a PN sequence are known as chips and the inverse of its period is known as chip rate. The null-to-null bandwidth of the main lobe of the resulting signal is two times the chipping rate, Fchip.

Figure 1. DS Spread Spectrum Transmitter - Basic Block Diagram



The PN sequence is a bit sequence with special characteristics involving cross correlation properties, which make the correlation between one code and a different code very low, and the correlation of a code with itself very high. The receiver uses these properties to recover data from a specific transmitter in the presence of noise and transmissions from other users. An 8-bit digital block in the PSoC can be configured as a linear feedback shift register (LFSR).

The LFSR can generate a PN sequence with a length of up to 255 chips with complete control over the defining polynomial and the seed (initial value). Multiple blocks can be chained together for a maximum LFSR of 32 bits, which can generate a PN sequence of length 4,294,967,295 chips.

This example design uses the PSoC to generate a synchronous PN chipping sequence and serial data stream, which can then be presented to another IC or circuit for combination, RF conversion, and transmission. PSoC contains all the pieces required to do this, and with minimal required processor overhead, even for relatively high data rates and long PN sequences.

Implementation Details

Figure 2 shows the logic diagram of the chip sequence and data stream generator as implemented in this example. The xmit_ena counter gates the clock for the chip sequence divider and the serial clock divider. This frames the packet, and is only high during the transmission of the packet. Because this gates the actual clocks for the counters, this gate is synchronous to the chip sequence and data stream outputs. The transmit gating signal is also used to gate the outputs to guarantee their states when the packet is not active. PRS8 (8-bit pseudo-random sequence) is the PN sequence generator, and the SPI Master is used as our serial data stream generator.

Figure 2. Logic Diagram for PSoC DSSS Generator

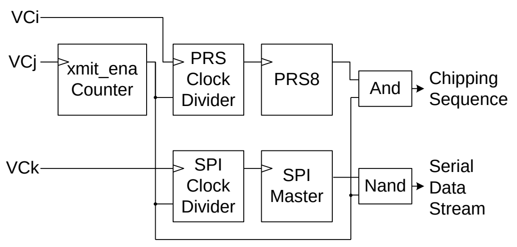
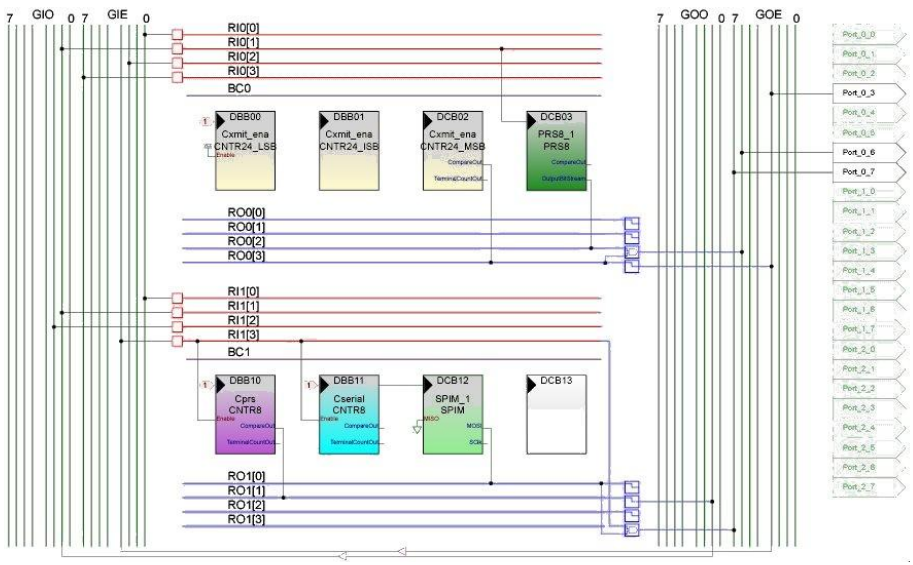


Figure 3 shows the implementation with the CY8C27xxx (in PSoC Designer 5.4), using seven of the eight digital blocks. The transmit gate generator, SPI clock divider, and PRS clock divider are named Cxmit_ena, Cserial, and Cprs, respectively. The Cxmit_ena output is looped back through the interconnect so it can be used in the second row for the counter gating. The output of Cprs is also looped back so it can be routed to the first row to provide the clock for the PRS8 block. The PRS output and SPI output are both combined with the Cxmit_ena signal via the row lookup table logic functions. The resulting chipping sequence appears on P0[6], the serial data stream appears on P0[7], and the gating signal appears on P0[3], which can be used as a scope trigger to see the signals. Interrupt code is called on the terminal count of Cxmit_ena to stop the counter at the end of the packet; thus, only one packet will appear. Sending additional packets will require re-initializing the blocks and restarting Cxmit_ena.

Although this design uses all but one digital block, remember that dynamic re-configuration allows us to load this configuration when transmitting and then unload it when complete. This leaves all the digital blocks free when not actively transmitting a packet. (Refer to Application Notes AN2104 and AN2094 for further explanation of dynamic re-configuration.) If more than one digital block is needed during transmission, it is also possible to gain another digital block by trading off sequence length and/or packet size. Cxmit_ena was chosen to be 24 bits to accommodate a packet size more than 256-byte at a PN sequence length of 63. If a smaller packet size or shorter PN sequence length is used, Cxmit_ena can be reduced to 16 bits, thus, increasing the spare digital blocks to 2.

Figure 3. Digital Block Layout for DS Generation

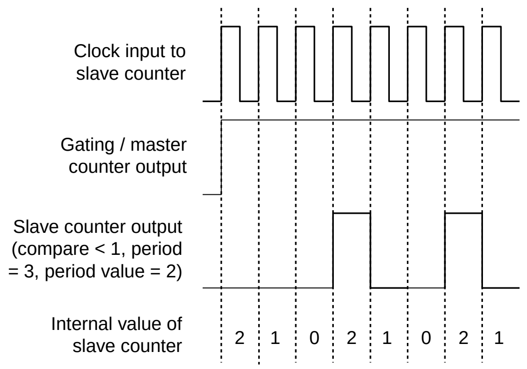


Clocking Details

The Cxmit_ena, Cprs, and Cserial can all be clocked with different input clock frequencies; however, the calculation of the periods needs to consider these frequencies. The input clock of the SPI Master is twice the output serial bit rate, while the input clock to the PRS is equal to the chipping rate. Also, all clocks need to be derived from the same source to maintain synchronicity between the chip sequence and the data stream.

DS spread spectrum requires that the data stream and chipping sequence start at the same time and remain synchronous. This is done here using Cxmit_ena as a master clock to gate Cprs and Cserial, which are essentially slave clocks. When PSoC counters are gated, they begin at their period value and begin counting down. To guarantee a full first period, the counters are set to compare less-than or equal-to one. As Figure 4 shows, this guarantees that the first period is equal to all additional periods with respect to the rising edge, which is when both the SPI Master and PRS transition. The assertion of the gating clock is a “virtual” rising edge, because that is when the first serial data bit and first chip appear on the outputs. Note that the timing of the compare output of the Counter User Module in PSoC has a latency of one, with respect to the internal count.

Figure 4. Counter Clocking for a Gating Counter and Slave Counter with a Period of 3


Clock input to slave counter
Gating / master counter output
Slave counter output (compare < 1, period = 3, period value = 2)
Internal value of slave counter
2
1
0
2
1
0
2
1

The SPI Master requires three clock cycles before the first data bit appears at the output of the shift register. Because we want the first bit present when we start our packet, this implementation has to “prime the pump” by giving the SPIM three clock transitions before the packet is started. We can manipulate the Cserial counter to give the clock pulses we need. If the Cserial counter is enabled with a compare value higher than the period value, the compare output will be high. When it is stopped, the compare output will always be low. So we set up the counter to be enabled by changing the enable input from Cxmit_ena to Vdd, setting the compare value higher than the period value, and then starting and stopping the module a few times to get our setup clocks.

After setting the counter, reset the period and compare values back, change the counter enable input back to Cxmit_ena, and restart the counter.

The counter compare output will now stay low, waiting for the assertion of Cxmit_ena to start counting.

If a precise frequency is needed that cannot be generated by dividing the PSoC’s internal 24-MHz clock, an external clock can be applied to P1[4]. This external clock can then be selected as the source for the internal SYSCLK. This will in turn be the source for all the VCN dividers, which can clock all the digital blocks.

Data Interface

The data packet contents for this example reside in the flash and are set up at the beginning of *main.c* (see the Appendix). The code handles a 16-bit packet length (in bytes), but the actual maximum packet length will depend on the length of the Cxmit_ena counter, the length of the chipping sequence, and the clocking relationships between the counters. These relationships are commented in the declaration section of *main.c*. The project, when configured for a sequence length of 63 chips, can support a packet length of 16 Kb as shipped.

The example project is set up to trigger an interrupt when the SPI transmit register is read empty. This happens when the first data bit appears on the output of the shift register. The processor has close to eight serial bit periods to process this interrupt, which loads the next data byte. The interrupt code in the example is dense assembly to ensure efficient operation at fast data rates (see the Appendix). At slower data rates, this code can be written in a C function for greater functionality and readability.

Output Gating

The SPI and PRS both present the first bit on their outputs before they are clocked. Because of this, we need to use Cxmit_ena to activate the outputs when the packet begins. This is done based on the sense required for the outputs.

In this example, the serial data stream signal is required to be an active low output, so it needs to be held high when the packet is not transmitting. Also, it needs to be low when the data bit is ‘1’ and vice versa. Therefore, the output is high whenever Cxmit_ena is low. If Cxmit_ena is high and the data bit ‘0’, the output is also high. In other words, the output is low only if both the data bit and Cxmit_ena are high. This is the NAND function, which is a logic function available in the LUTs in the PSoC.

The chipping sequence is an active high gated by the Cxmit_ena. The output should only be high if both Cxmit_ena is high and the PN chip is high; therefore, this is an AND function – also available in the PSoC. Drawing a two input logic table may be useful in determining what function is needed, depending on the required inputs and outputs.

Internal XOR

The implemented example provides both the chipping sequence and serial data stream, which must be combined in an external device before being converted to RF for broadcast. Some DSSS implementations may require that the chip sequence and data stream be pre-combined for direct RF modulation. This combination is commonly done with an XOR function. The XOR function is available as a LUT logic function in the PSoC device.

Figure 5 shows an example of a layout, which includes the XOR combination of the chipping sequence and serial data stream. The PRS output is routed back to input row 1[0], where it is combined with the SPIM output in an XOR LUT function. This combination is then brought out to P2[4], where it is connected externally to P2[2] so that it can be routed to input row 0[2]. In input row 0[2], the PRS output-SPIM output combination is combined with the Cxmit_ena gating signal with a logic AND function. Note also that this layout shows an alternative routing for the PRS8 input clock from Cprs using the broadcast row features.

Figure 5. Layout Including XOR Combination

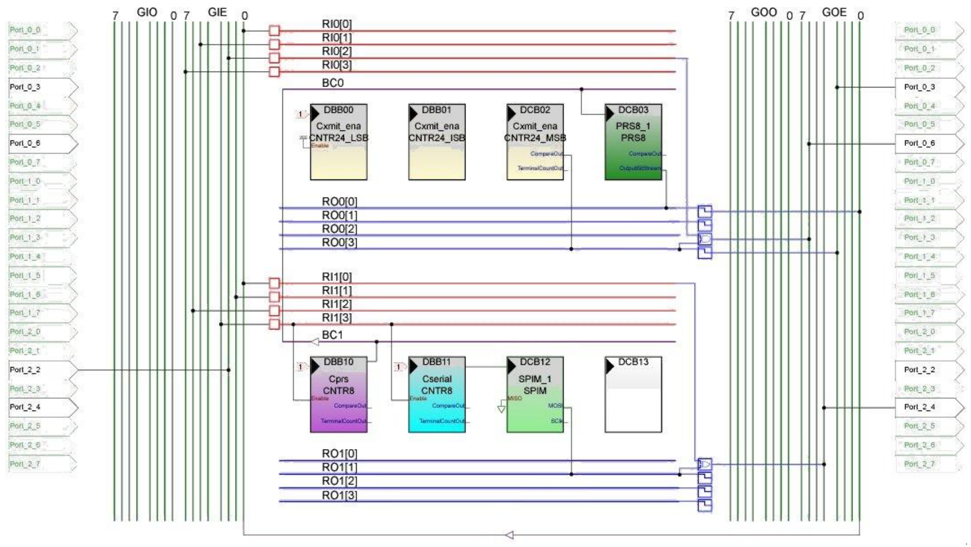


Figure 6. Actual Data Packet

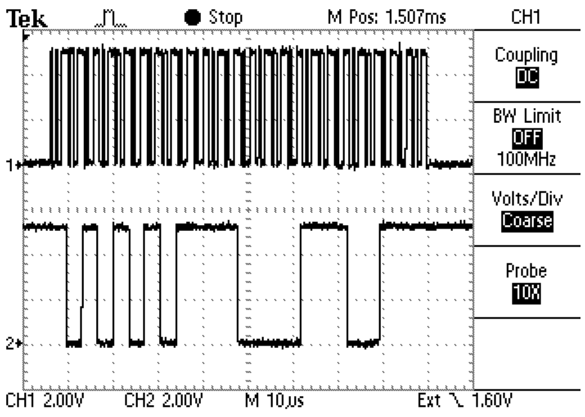
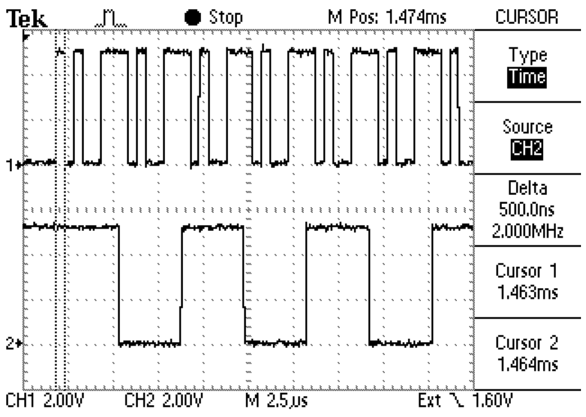


Figure 7. Beginning of Actual Data Packet



Results

Figure 6 shows a scope plot of a short data sequence being output on channel 2, consisting of [0x55, 0x0F, 0x18] MSB first, active low, at a serial bit rate of 286 Kbps. Channel 1 is a 7-bit chipping sequence [1,0,1,0,0,1,1] at a chip rate of 2 MHz. Figure 7 shows the beginning of the same packet, giving more detail of how the data bits line up with the chipping sequence. This is running with a 21 percent processor overhead at a CPU speed of 12 MHz.

A chipping rate as high as 12 MHz can be produced from the PSoC, limited by the I/O specifications. The data rate is limited by the interrupt processing for the data handling.

The assembly code in the example takes about 80 clock cycles to execute, which will yield a maximum data rate of 2.4 Mbps with 100 percent CPU utilization at 24 MHz.

Summary

Spread spectrum communication is becoming more and more popular, especially in industrial and data collection applications. Being able to implement the DSSS transmitter baseband functions in the PSoC allows such applications to have lower cost and greater integration. Dynamic re-configuration also means that all of the other features of the PSoC are available, adding more capability and functionality to these designs. Transmit away!

Appendix

Code Listing

Listing 1. Main.c

```
//-----------------------------------------------------------------
//MAIN.C
//-----------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

//-----------------------------------------------------------------
// PRS DEFINES
//-----------------------------------------------------------------
//#define PRS_LENGTH       63     //63 bits per serial data bit
//#define bPOLY     0x39   //Modular polynomial = [6,5,4,1], length of 63

#define   PRS_LENGTH       7      //7 bits per serial data bit
#define   bPOLY            0x06   // Modular Polynomial = [3,2], length of 7

//#define PRS_LENGTH       15     //15 bits per serial data bit
//#define bPOLY     0x0C          // Modular Polynomial = [4,3], length of 15

#define   bSEED     0xFF          // Seed value

//-----------------------------------------------------------------
// CLOCK RELATIONSHIPS
//-----------------------------------------------------------------
#define   CXMIT_ENA_FREQ_PRS 2    // FROM IDE:  Clock frequency input of Cxmit_ena counter is
              // this times the PRS frequency (actual chip rate)
              // NOTE:  Used to calculate wGATECOMPARE for
// Cxmit_ena.  Make sure counter width is adequate to
// accommodate the value.

#define   CSERIAL_FREQ_PRS_MUL2 1// FROM IDE:  Clock frequency input of Cserial counter is
// this times the PRS frequency (actual chip rate)
// times two. NOTE:  Used to calculate period of
// Cserial counter. Needs to be an integer, and
// wCSERIAL_PERVAL must fit into the counter width
// of Cserial.

//-----------------------------------------------------------------
// PACKET DATA AND COUNTER COMPARE/PERIOD CALCULATIONS
//-----------------------------------------------------------------
#define   PACKET_LENGTH_BYTES 3

// Now we'll calculate the compare value we need for our packet gating clock
#define    PACKET_LENGTH_BITS     (8 * PACKET_LENGTH_BYTES)
#define    wGATECOMPARE           (PRS_LENGTH * CXMIT_ENA_FREQ_PRS * PACKET_LENGTH_BITS)

// Calculate Cserial counter periodval (actual period minus 1)
#define   bCSERIAL_PERVAL         ((PRS_LENGTH * CSERIAL_FREQ_PRS_MUL2)-1)

const BYTE packetdata[] =
{
   0x55, 0x0f, 0x18
};

WORD wPacketIndex;
BYTE bDBINtemp, btemp;

void main()
```

```
{
        //PRS8 setup - PRS clock is gated by transmit enable, so go ahead and start it
        PRS8_1_WritePolynomial(bPOLY);  // load the PRS polynomial
        PRS8_1_WriteSeed(bSEED);         // load the PRS seed
        PRS8_1_Start();                  // start the PRS8
    Cprs_Start();                 //Start the PRS clock

        //SPIM setup - start, enable interrupts, load the first byte
        SPIM_1_Start(0x00);              // Start SPI Master in mode 0
        SPIM_1_EnableInt();
        wPacketIndex = 0;
        SPIM_1_SendTxData(packetdata[wPacketIndex++]); //Load the first byte in
// the packet
        M8C_EnableGInt;
        DCB12FN &= 0x0F;          //SPIM function register, upper nibble must
//  be 0 for interrupt on TX reg empty.



        //Prepare the SPIM for the beginning of the packet by providing it
    // three "setup clocks" from Cserial
        Cserial_WritePeriod(0x00);              //Set the period lower than compare
// value (compare always true)
        Cserial_WriteCompareValue(0xFF);
        bDBINtemp = DBB11IN;                    //Save the input mux settings for
// the Cserial counter
        btemp = bDBINtemp;               //Manually enable the counter by changing
// the enable input mux to VCC
        btemp |= 0x10;
        btemp &= 0x1F;
        DBB11IN = btemp;

        //Provide three pulses to initialize the SPIM
        Cserial_Start();                 //compare output high
        Cserial_Stop();                  //compare output low
        Cserial_Start();                 //compare output high
        Cserial_Stop();                  //compare output low
        Cserial_Start();                 //compare output high
        Cserial_Stop();                  //compare output low

        //Now, change the enable input back to our gate and restart the counter,
    //  which will wait for the gate
        DBB11IN = bDBINtemp;
        Cserial_WritePeriod(bCSERIAL_PERVAL);   //Set the period value for Cserial
        Cserial_WriteCompareValue(1);           //Compare less than 1 ("less than" is
// specified in IDE)
        Cserial_Start();

        //Set up our gating counter
        Cxmit_ena_WritePeriod(wGATECOMPARE + 256);    //Set the "off time" for the gate
//    clock (256 is arbitrary)
        Cxmit_ena_WriteCompareValue(wGATECOMPARE);    //Set the length of the gate clock
        Cxmit_ena_EnableInt();                  // ISR stops Cxmit_ena after
//   packet

        //Start our gating signal, which will kick everything off
        Cxmit_ena_Start();

       while(1)
       {
          //infinite loop
          asm("nop");
       }
}
```

Listing 2. User Code from SPIM Interrupt Handler (_SPIM_1_ISR in spim_1int.asm)

```
;;
;;Write out the next byte from _packetdata, increment our _wPacketIndex
;;
mov A, REG[SPIM_1_CONTROL_REG]    ;dummy read to SPI control register to re-enable interrupt
mov A,<_packetdata         ;LSB of packetdata address
adc A, [_wPacketIndex+1]          ;add LSB of index to LSB of packetdata addr
mov X, A             ;X is now our LSB, indexed to our
             ;    current byte position
mov A,>_packetdata         ;MSB of packetdata address
jnc nocarry          ;if previous add had a carry, we need to add it
             ;    in to the MSB index
inc A
    nocarry:
add A, [_wPacketIndex]            ;Now add in the MSB of the wPacketIndex
romx
call SPIM_1_SendTxData
inc [_wPacketIndex+1]             ;Increment our packet index (LSB first)
jnc finish
inc [_wPacketIndex]        ;Add our carry to the MSB of wPacketIndex
    finish:
```

Listing 3. User Code from Cxmit_ena Interrupt Handler (_Cxmit_ena_ISR in cxmit_enaint.asm)

```
;;
   ;;Stop the gating counter on terminal count
;;  so we'll just get one packet
   ;;
   call Cxmit_ena_Stop
```

Document History

Document Title: PSoC® 1 Implementation of a Direct Sequence Spread Spectrum Baseband Transmitter - AN2165

Document Number: 001-35345

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1513664	YIS	11/02/2007	Recataloged Application Note
*A	3154677	GIR	01/31/2011	Title updated. Abstract updated. Associated PSoC Designer project updated to PD5.1-sp1.
*B	4316783	RKRM	03/24/2014	Listed related application notes on page 1. Updated Spread Spectrum Primer. Updated Summary. Updated the associated PSoC Designer project to PD5.4. Updated the template.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

CYPRESS
PERFORM

Cypress Semiconductor	Phone	: 408-943-2600
198 Champion Court	Fax	: 408-943-4730
San Jose, CA 95134-1709	Website	: www.cypress.com