

Stud Finder

By: Chris and Vincent Paiano

Associated Project: Yes

Associated Part Family: CY8C22xxx, CY8C24xxx, CY8C27xxx

PSoC Designer Version: 4.1

Abstract

The PSoC™ device is configured to react to the presence of studs through drywall to allow the user to easily locate a sturdy point to anchor or hang up an object. The sensor presents the studs via five LEDs. By holding the Stud Finder against a wall and moving it along horizontally, the different LEDs will light up as a stud is sensed.

Introduction

When hanging a heavy object, it is advisable to find a piece of your home's supporting wood beams within your wall on which to anchor the object or your drywall may rip apart. Using a PSoC Stud Finder, one can simply pass it along a wall and look at the LEDs to find these studs without penetrating the drywall. This is very similar to the operation of COTS (Commercial-Off-The-Shelf) units, available in hardware stores everywhere.

Software/PSoC Implementation

The PSoC implementation of the Stud Finder involves a resonant circuit based on two inverters' propagation delays, which generates an oscillation. This oscillation clocks an 8-bit pre-scale counter to divide the frequency down, and that output is then fed to a 16-bit counter clock input to count the frequency as it comes in. An additional 8-bit counter is set up as a sample time generator; every time it reaches terminal count, the frequency counter's count is retrieved. The difference between the current count and the last count is read as the frequency's value. Based on the constant NumOfReadings, a number of readings is taken and then averaged. This average controls which LEDs light up at any given time, based on the constants LEDxThreshold.

A TX8 block, clocked at 8x the desired bit rate of 115.2 kbps, was also included to provide a debugging function. By connecting this block's output to the serial port on your computer and then opening a standard serial terminal program (such as Hyperterminal® in a Windows®-based environment) and then setting the debug constant to 1, the frequency average as well as the active LED can be viewed. This allowed us to empirically obtain the values for the five LEDxThreshold constants. This was an invaluable debugging method, as the emulator does not allow real-time viewing of variables – one must stop the execution or set breakpoints to view a snapshot of the desired variables. Not to mention the fact that our LCD had inexplicably stopped working, so we needed a way to look at the variables outside of the emulator.

A sample Hyperterminal connection file is included with this project, named *LocalToCom1.ht* – which should work for most standard PCs to connect to the first serial port, COM1. The settings can easily be altered in the Hyperterminal program itself to connect to another serial port.

The PSoC code, shown in Code 1, demonstrates how all this might be achieved in practice. A flowchart of the code is shown in Figure 1, and the configurations of the PSoC resources appear in Figures 2, 3, 4 and 5.

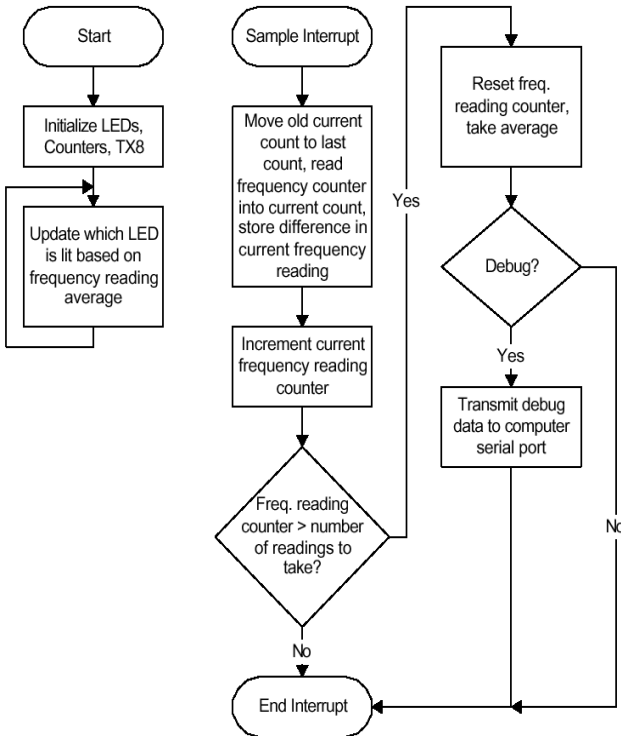


Figure 1. Program Flowchart

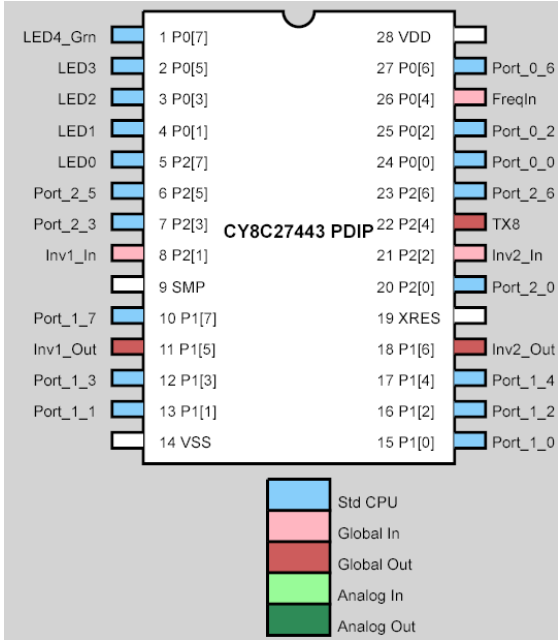


Figure 2. PSoC Pin Configuration

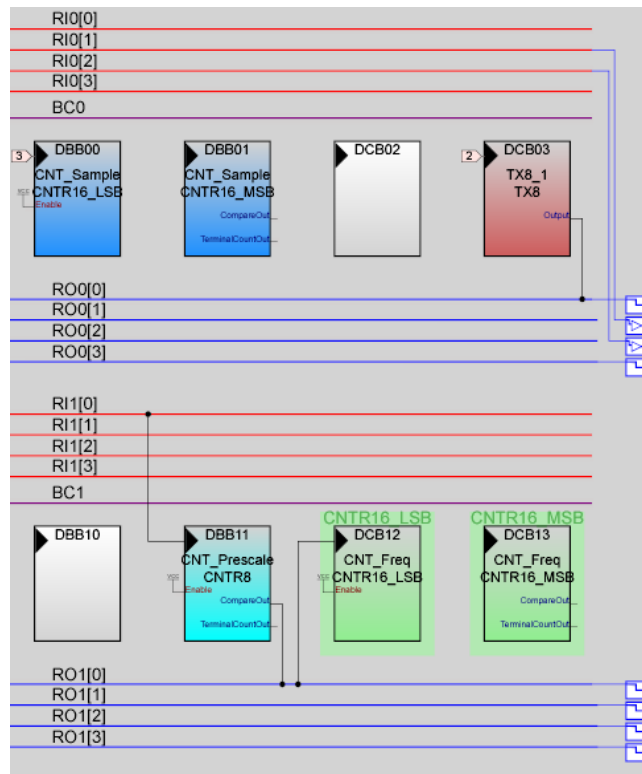


Figure 3. PSoc User Module Configuration GUI

CNT_Prescale		CNT_Sample	
User Module Parameters		User Module Parameters	
Clock	Row_1_Input_0	Clock	VC3
Enable	High	Enable	High
CompareOut	Row_1_Output_0	CompareOut	None
TerminalCountOut	None	TerminalCountOut	None
Period	32	Period	5000
CompareValue	16	CompareValue	2500
CompareType	Less Than Or Equal	CompareType	Less Than Or Equal
InterruptType	Terminal Count	InterruptType	Terminal Count
ClockSync	Unsynchronized	ClockSync	Unsynchronized
InvertEnable	Normal	InvertEnable	Normal

TX8_1	
User Module Parameters	
Clock	VC2
Output	Row_0_Output_0
Parity	None

Figure 4. PSoc User Module Parameter Settings

Global Resources		Name	Port	Select	Drive	Interrupt
CPU_Clock	24_MHz (SysClk/1)	Port_0_0	P0[0]	StdCPU	High Z Ar	DisableInt
32K_Select	Internal	LED1	P0[1]	StdCPU	Strong	DisableInt
PLL_Mode	Disable	Port_0_2	P0[2]	StdCPU	High Z Ar	DisableInt
Sleep_Timer	512_Hz	LED2	P0[3]	StdCPU	Strong	DisableInt
VC1= SysClk/N	13	FreqIn	P0[4]	GlobalInE	High Z	DisableInt
VC2= VC1/N	2	LED3	P0[5]	StdCPU	Strong	DisableInt
VC3 Source	SysClk/1	Port_0_6	P0[6]	StdCPU	High Z Ar	DisableInt
VC3 Divider	256	LED4_Gm	P0[7]	StdCPU	Strong	DisableInt
SysClk Source	Internal 24_MHz	Port_1_0	P1[0]	StdCPU	High Z Ar	DisableInt
SysClk*2 Disable	No	Port_1_1	P1[1]	StdCPU	High Z Ar	DisableInt
Analog Power	All Off	Port_1_2	P1[2]	StdCPU	High Z Ar	DisableInt
Ref Mux	(Vdd/2)+/-BandGap	Port_1_3	P1[3]	StdCPU	High Z Ar	DisableInt
AGndBypass	Disable	Port_1_4	P1[4]	StdCPU	High Z Ar	DisableInt
Op-Amp Bias	Low	Inv1_Out	P1[5]	GlobalOut	Strong	DisableInt
A_Buff_Power	Low	Inv2_Out	P1[6]	GlobalOut	Strong	DisableInt
SwitchModePump	OFF	Port_1_7	P1[7]	StdCPU	High Z Ar	DisableInt
Trip Voltage [LVD (SMP)]	4.64V (5.00V)	Port_2_0	P2[0]	StdCPU	High Z Ar	DisableInt
LVDThrottleBack	Disable	Inv1_In	P2[1]	GlobalInE	High Z	DisableInt
Supply Voltage	5.0V	Inv2_In	P2[2]	GlobalInE	High Z	DisableInt
Watchdog Enable	Disable	Port_2_3	P2[3]	StdCPU	High Z Ar	DisableInt
		TX8	P2[4]	GlobalOut	Strong	DisableInt
		Port_2_5	P2[5]	StdCPU	High Z Ar	DisableInt
		Port_2_6	P2[6]	StdCPU	High Z Ar	DisableInt
		LED0	P2[7]	StdCPU	Strong	DisableInt

Figure 5. PSoC Global Resource Settings

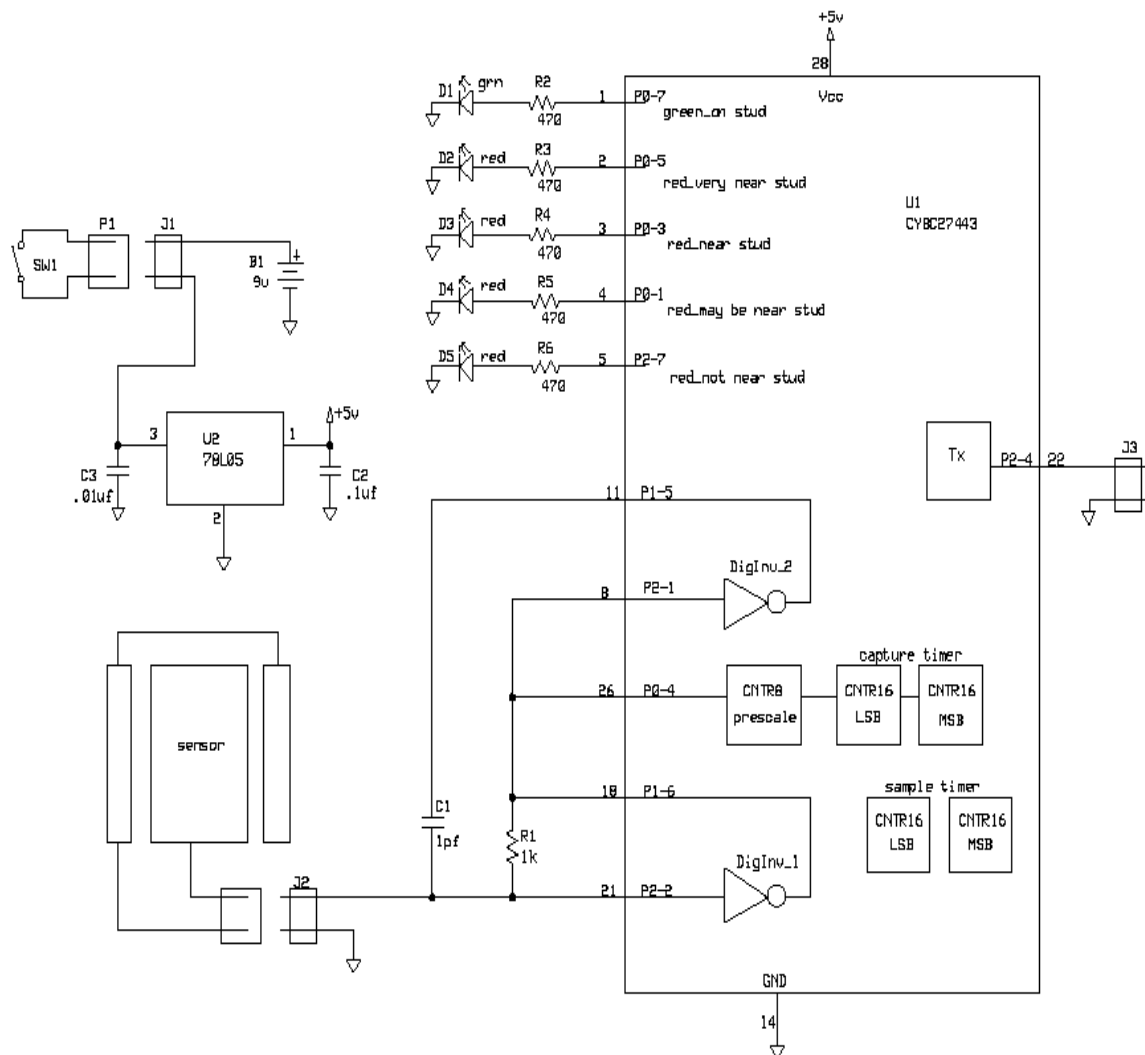


Figure 6. Schematic

Hardware Implementation

The hardware implementation of the LED Stud Finder, shown above in Figure 6, consists of a pair of inverters on the digital interconnects, configured with R_1 , C_1 , and the antenna to form an oscillator running at approx 9 MHz. In the prototype, aluminum tape was cut into a 1.5-inch square (the width of a 2-by-4) for the active portion, and two 1.5 x 0.5-inch sections for the passive.

Capacitive coupling to ground of the first inverter input forms a variable frequency proximity sensor, the output clocking an 8-bit counter to pre-scale the signal before it is inserted into the 16-bit capture timer.

The 5-volt regulator provides a stable voltage for consistent readings. The active high current for the LED configuration is easily sourced through the PSoC device.

Photographs of the prototype front and back are shown in Figure 7.

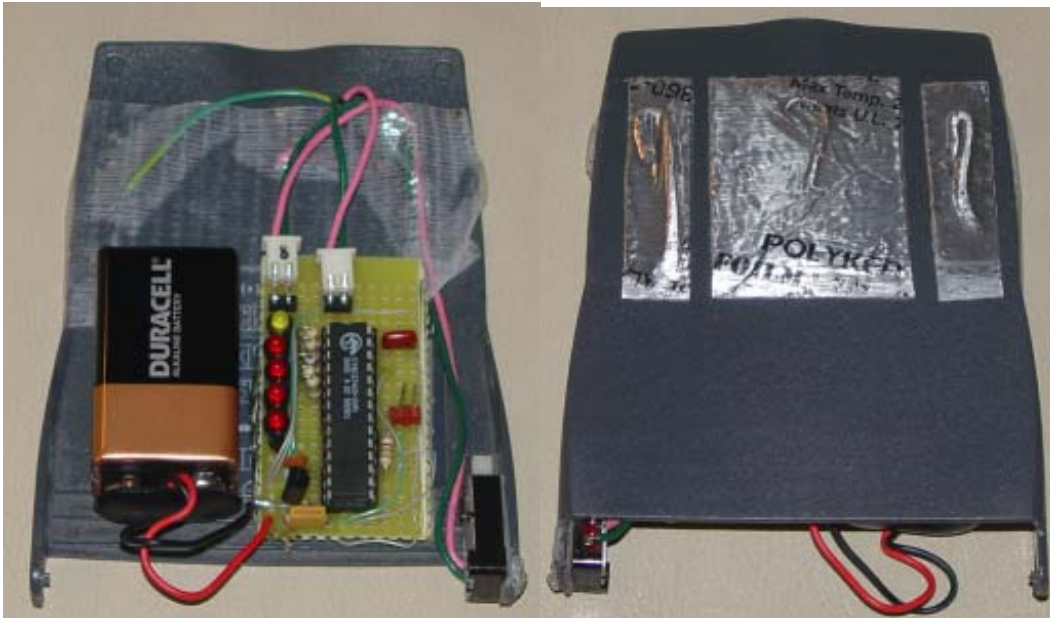


Figure 7. Stud Finder Prototype Front (Left) and Back (Right)

```

main.c:
#include <m8c.h>          //Part-specific constants and macros
#include "PSOCAPI.h"     //PSOC API definitions for all User Modules
#include "chris_utils.h"
#define NumReadings 1
#define LED0Threshold 32000 //Red - not near stud
#define LED1Threshold 30600 //Red - may be near stud
#define LED2Threshold 30575 //Red - near stud
#define LED3Threshold 30550 //Red - very near stud
#define LED4Threshold 30500 //Yellow=On Stud
#define LED0(b) (PRT2DR = (b==0) ? (PRT2DR&0x7F) : (PRT2DR|0x80)) //P2_7
#define LED1(b) (PRT0DR = (b==0) ? (PRT0DR&0xFD) : (PRT0DR|0x02)) //P0_1
#define LED2(b) (PRT0DR = (b==0) ? (PRT0DR&0xF7) : (PRT0DR|0x08)) //P0_3
#define LED3(b) (PRT0DR = (b==0) ? (PRT0DR&0xDF) : (PRT0DR|0x20)) //P0_5
#define LED4(b) (PRT0DR = (b==0) ? (PRT0DR&0x7F) : (PRT0DR|0x80)) //P0_7
#define LEDUpdateDelay 1
unsigned int FreqCurrentCount=CNT_Freq_PERIOD;
unsigned int FreqLastCount;
unsigned int FreqReading[NumReadings];
unsigned int FreqAvg;
unsigned char ReadingCounter=0;
unsigned char LEDUpdateCounter=0;
unsigned char FreqIntCounter=0;
unsigned char LEDActive=255;
void CalcFreqAvg(void);
#define Debug 0 //Make this 1 to activate the TX8
#define RS232BurstDelay 2 //How many times function is called before burst is
sent
int RS232BurstCounter = 0;
void SendRS232Burst(void);
void main()
{
    CNT_Sample_EnableInt();
    CNT_Sample_Start();
    CNT_Prescale_Start();
    CNT_Freq_EnableInt();
    CNT_Freq_Start();

```

```

TX8_1_Start(TX8_1_PARITY_NONE);
M8C_EnableGInt;
SendRS232(13); //carriage return/newline - reset/init Hyperterminal display
while(1)
{
    if (++LEDUpdateCounter>LEDUpdateDelay)
    {
        LEDUpdateCounter=0;
        if (FreqAvg<LED4Threshold) {LED0(0); LED1(0); LED2(0); LED3(0);
LED4(1); LEDActive=4;}
        else if (FreqAvg<LED3Threshold) {LED0(0); LED1(0); LED2(0);
LED4(0); LED3(1); LEDActive=3;}
        else if (FreqAvg<LED2Threshold) {LED0(0); LED1(0); LED3(0);
LED4(0); LED2(1); LEDActive=2;}
        else if (FreqAvg<LED1Threshold) {LED0(0); LED2(0); LED3(0);
LED4(0); LED1(1); LEDActive=1;}
        else if (FreqAvg<LED0Threshold) {LED1(0); LED2(0); LED3(0);
LED4(0); LED0(1); LEDActive=0;}
        else {LED0(0); LED1(0); LED2(0); LED3(0); LED4(0); LEDActive=255;}
    }
}
#pragma interrupt_handler SampleINT
void SampleINT()
{
    //Time to read frequency measurement
    FreqLastCount=FreqCurrentCount;
    FreqCurrentCount=CNT_Freq_wReadCounter();
    if (FreqIntCounter>1) //overflow!
        FreqReading[ReadingCounter]=CNT_Freq_PERIOD;
    else if (FreqIntCounter) //single period reset occurred
        FreqReading[ReadingCounter]=FreqLastCount+(CNT_Freq_PERIOD-
FreqCurrentCount);
    else //readings within timer period
        FreqReading[ReadingCounter]=FreqLastCount-FreqCurrentCount;
    if (++ReadingCounter>=NumReadings) CalcFreqAvg();
    FreqIntCounter=0;

    /**
    FreqReading[ReadingCounter]=CNT_Freq_PERIOD-CNT_Freq_wReadCounter();
    if (++ReadingCounter>=NumReadings) CalcFreqAvg();
    CNT_Freq_Stop();
    CNT_Freq_WritePeriod(CNT_Freq_PERIOD);
    CNT_Freq_WriteCompareValue(CNT_Freq_COMPARE_VALUE);
    CNT_Freq_Start();
    /**/
}
#pragma interrupt_handler FreqINT
void FreqINT()
{
    //Timer period reset occurring
    FreqIntCounter++;
}
void CalcFreqAvg()
{
    FreqAvg=0;
    for (ReadingCounter=0;ReadingCounter<NumReadings;ReadingCounter++)
        FreqAvg+=FreqReading[ReadingCounter];
    FreqAvg/=NumReadings;
    SendRS232Burst();
    ReadingCounter=0;
}

//TX8 RS232 Debug
void SendRS232Burst()
{
    if (Debug)
    {
        if (++RS232BurstCounter>RS232BurstDelay)
        {
            RS232BurstCounter=0;
            TxString("Av: ");
        }
    }
}

```

```

        IntToZTS(FreqAvg);
        TxZeroTerminatedRamString(ZTS);
        TxString(", LED:");
        CharToZTS(LEDActive);
        TxZeroTerminatedRamString(ZTS);
        SendRS232(13); //carriage return/newline
    }
}

chris_utils.h:
#include "stdlib.h"
#include "globalparams.h"
#define Bit(bitNumber) ( 1 << (bitNumber) )
char ZTS[21];
void TxZeroTerminatedRamString(BYTE *);
void TxString(const char *);
void CharToZTS(unsigned char);
void IntToZTS(unsigned int);
void LIntToZTS(unsigned long int);
#define ProcessorSpeedInMHz 24
#define WaitMSCorrectionFactor 5*ProcessorSpeedInMHz //To calibrate Wait function for MS
#define RS232ByteDelayInMS 3
#define RS232ByteDelay RS232ByteDelayInMS*WaitMSCorrectionFactor
long int Counter = 0;
void Wait(long int);
void SendRS232(char);
void Sleep(void);

void Sleep()
{
    //    ARF_CR=0;        //Analog power off
    M8C_EnableIntMask(INT_MSK0,INT_MSK0_SLEEP); //Enable the sleep interrupts
    M8C_ClearWDTAndSleep; //reset the sleep timer
    M8C_Sleep;
    asm("nop");        //fetched before actual sleep
    M8C_DisableIntMask(INT_MSK0,INT_MSK0_SLEEP); //Disable the sleep interrupts
    //    ARF_CR=ANALOG_POWER; //Analog power back on
}
void Wait(long int ToWait){ for (Counter = 0; Counter < ToWait; Counter++); }
void SendRS232(char ToSend)
{
    TX8_1_SendData(ToSend);
    while( !( bTX8_1_ReadTxStatus() & TX8_TX_BUFFER_EMPTY ) ); //tx started
    while( !( bTX8_1_ReadTxStatus() & TX8_TX_COMPLETE ) ); //tx... complete
    Wait(RS232ByteDelay);
}
void CharToZTS(unsigned char ToConvert)
{
    if (ToConvert >= 100) itoa(&ZTS[0], ToConvert, 10);
    else if (ToConvert >= 10){ZTS[0]='0'; itoa(&ZTS[1], ToConvert, 10);}
    else{ZTS[0]='0'; ZTS[1]='0'; itoa(&ZTS[2], ToConvert, 10);}
    ZTS[3] = 0; //for ZTS
}
void IntToZTS(unsigned int ToConvert)
{
    if (ToConvert >= 10000) itoa(&ZTS[0], ToConvert, 10);
    else if (ToConvert >= 1000){ZTS[0]='0'; itoa(&ZTS[1], ToConvert, 10);}
    else if (ToConvert >= 100){ZTS[0]='0'; ZTS[1]='0'; itoa(&ZTS[2], ToConvert, 10);}
    else if (ToConvert >= 10){ZTS[0]='0'; ZTS[1]='0'; ZTS[2]='0'; itoa(&ZTS[3],
ToConvert, 10);}
    else{ZTS[0]='0'; ZTS[1]='0'; ZTS[2]='0'; ZTS[3]='0'; itoa(&ZTS[4], ToConvert,
10);}
    ZTS[5] = 0; //for ZTS
}
void LIntToZTS(unsigned long int ToConvert)
{
    if (ToConvert> 10000000) ltoa(ZTS, ToConvert, 10);
    else if (ToConvert> 1000000) {ltoa(&ZTS[1], ToConvert, 10); ZTS[0]='0'; }
}

```

```

        else if (ToConvert > 100000) {ltoa(&ZTS[2], ToConvert, 10); ZTS[0]='0';
ZTS[1]='0'; }
        else if (ToConvert > 10000) {ltoa(&ZTS[3], ToConvert, 10); ZTS[0]='0';
ZTS[1]='0';\
                                ZTS[2]='0'; }
        else if (ToConvert > 1000) {ltoa(&ZTS[4], ToConvert, 10); ZTS[0]='0';
ZTS[1]='0';\
                                ZTS[2]='0'; ZTS[3]='0'; }
        else if (ToConvert > 100) {ltoa(&ZTS[5], ToConvert, 10); ZTS[0]='0';
ZTS[1]='0';\
                                ZTS[2]='0'; ZTS[3]='0';
ZTS[4]='0'; }
        else if (ToConvert > 10) {ltoa(&ZTS[6], ToConvert, 10); ZTS[0]='0'; ZTS[1]='0';\
                                ZTS[2]='0'; ZTS[3]='0';
ZTS[4]='0'; ZTS[5]='0'; }
        else {ltoa(&ZTS[7], ToConvert, 10); ZTS[0]='0'; ZTS[1]='0'; ZTS[2]='0';
ZTS[3]='0';\
                                ZTS[4]='0'; ZTS[5]='0';
ZTS[6]='0'; }
        ZTS[8]=0; //null
}
void TxZeroTerminatedRamString( BYTE * pbStrPtr )
{
    /* check for the end condition, before sending the next byte */
    while( *pbStrPtr != 0 )
    {
        SendRS232(*pbStrPtr);
        pbStrPtr++;
    }
}
void TxString( const char * pbStrPtr )
{
    /* check for the end condition, before sending the next byte */
    while( *pbStrPtr != 0 )
    {
        SendRS232(*pbStrPtr);
        pbStrPtr++;
    }
}

/*
Macros for turning bits on and off

PortX_Y(b) : PortNumber=X[7..0],
              Bit=Y[7..0], b[1,0]

PortX(b) : PortNumber=X[7..0], b[1,0]

*/

#define Port0(b) (PRT0DR = (b==0) ? 0x00 : 0xFF)
#define Port0_0(b) (PRT0DR = (b==0) ? (PRT0DR&0xFE) : (PRT0DR|0x01))
#define Port0_1(b) (PRT0DR = (b==0) ? (PRT0DR&0xFD) : (PRT0DR|0x02))
#define Port0_2(b) (PRT0DR = (b==0) ? (PRT0DR&0xFB) : (PRT0DR|0x04))
#define Port0_3(b) (PRT0DR = (b==0) ? (PRT0DR&0xF7) : (PRT0DR|0x08))
#define Port0_4(b) (PRT0DR = (b==0) ? (PRT0DR&0xEF) : (PRT0DR|0x10))
#define Port0_5(b) (PRT0DR = (b==0) ? (PRT0DR&0xDF) : (PRT0DR|0x20))
#define Port0_6(b) (PRT0DR = (b==0) ? (PRT0DR&0xBF) : (PRT0DR|0x40))
#define Port0_7(b) (PRT0DR = (b==0) ? (PRT0DR&0x7F) : (PRT0DR|0x80))
#define Port1(b) (PRT1DR = (b==0) ? 0x00 : 0xFF)
#define Port1_0(b) (PRT1DR = (b==0) ? (PRT1DR&0xFE) : (PRT1DR|0x01))
#define Port1_1(b) (PRT1DR = (b==0) ? (PRT1DR&0xFD) : (PRT1DR|0x02))
#define Port1_2(b) (PRT1DR = (b==0) ? (PRT1DR&0xFB) : (PRT1DR|0x04))
#define Port1_3(b) (PRT1DR = (b==0) ? (PRT1DR&0xF7) : (PRT1DR|0x08))
#define Port1_4(b) (PRT1DR = (b==0) ? (PRT1DR&0xEF) : (PRT1DR|0x10))
#define Port1_5(b) (PRT1DR = (b==0) ? (PRT1DR&0xDF) : (PRT1DR|0x20))
#define Port1_6(b) (PRT1DR = (b==0) ? (PRT1DR&0xBF) : (PRT1DR|0x40))
#define Port1_7(b) (PRT1DR = (b==0) ? (PRT1DR&0x7F) : (PRT1DR|0x80))

```

```

#define Port2(b) (PRT2DR = (b==0) ? 0x00 : 0xFF)
#define Port2_0(b) (PRT2DR = (b==0) ? (PRT2DR&0xFE) : (PRT2DR|0x01))
#define Port2_1(b) (PRT2DR = (b==0) ? (PRT2DR&0xFD) : (PRT2DR|0x02))
#define Port2_2(b) (PRT2DR = (b==0) ? (PRT2DR&0xFB) : (PRT2DR|0x04))
#define Port2_3(b) (PRT2DR = (b==0) ? (PRT2DR&0xF7) : (PRT2DR|0x08))
#define Port2_4(b) (PRT2DR = (b==0) ? (PRT2DR&0xEF) : (PRT2DR|0x10))
#define Port2_5(b) (PRT2DR = (b==0) ? (PRT2DR&0xDF) : (PRT2DR|0x20))
#define Port2_6(b) (PRT2DR = (b==0) ? (PRT2DR&0xBF) : (PRT2DR|0x40))
#define Port2_7(b) (PRT2DR = (b==0) ? (PRT2DR&0x7F) : (PRT2DR|0x80))
/**
#define Port3(b) (PRT3DR = (b==0) ? 0x00 : 0xFF)
#define Port3_0(b) (PRT3DR = (b==0) ? (PRT3DR&0xFE) : (PRT3DR|0x01))
#define Port3_1(b) (PRT3DR = (b==0) ? (PRT3DR&0xFD) : (PRT3DR|0x02))
#define Port3_2(b) (PRT3DR = (b==0) ? (PRT3DR&0xFB) : (PRT3DR|0x04))
#define Port3_3(b) (PRT3DR = (b==0) ? (PRT3DR&0xF7) : (PRT3DR|0x08))
#define Port3_4(b) (PRT3DR = (b==0) ? (PRT3DR&0xEF) : (PRT3DR|0x10))
#define Port3_5(b) (PRT3DR = (b==0) ? (PRT3DR&0xDF) : (PRT3DR|0x20))
#define Port3_6(b) (PRT3DR = (b==0) ? (PRT3DR&0xBF) : (PRT3DR|0x40))
#define Port3_7(b) (PRT3DR = (b==0) ? (PRT3DR&0x7F) : (PRT3DR|0x80))
#define Port4(b) (PRT4DR = (b==0) ? 0x00 : 0xFF)
#define Port4_0(b) (PRT4DR = (b==0) ? (PRT4DR&0xFE) : (PRT4DR|0x01))
#define Port4_1(b) (PRT4DR = (b==0) ? (PRT4DR&0xFD) : (PRT4DR|0x02))
#define Port4_2(b) (PRT4DR = (b==0) ? (PRT4DR&0xFB) : (PRT4DR|0x04))
#define Port4_3(b) (PRT4DR = (b==0) ? (PRT4DR&0xF7) : (PRT4DR|0x08))
#define Port4_4(b) (PRT4DR = (b==0) ? (PRT4DR&0xEF) : (PRT4DR|0x10))
#define Port4_5(b) (PRT4DR = (b==0) ? (PRT4DR&0xDF) : (PRT4DR|0x20))
#define Port4_6(b) (PRT4DR = (b==0) ? (PRT4DR&0xBF) : (PRT4DR|0x40))
#define Port4_7(b) (PRT4DR = (b==0) ? (PRT4DR&0x7F) : (PRT4DR|0x80))
#define Port5(b) (PRT5DR = (b==0) ? 0x00 : 0xFF)
#define Port5_0(b) (PRT5DR = (b==0) ? (PRT5DR&0xFE) : (PRT5DR|0x01))
#define Port5_1(b) (PRT5DR = (b==0) ? (PRT5DR&0xFD) : (PRT5DR|0x02))
#define Port5_2(b) (PRT5DR = (b==0) ? (PRT5DR&0xFB) : (PRT5DR|0x04))
#define Port5_3(b) (PRT5DR = (b==0) ? (PRT5DR&0xF7) : (PRT5DR|0x08))
#define Port5_4(b) (PRT5DR = (b==0) ? (PRT5DR&0xEF) : (PRT5DR|0x10))
#define Port5_5(b) (PRT5DR = (b==0) ? (PRT5DR&0xDF) : (PRT5DR|0x20))
#define Port5_6(b) (PRT5DR = (b==0) ? (PRT5DR&0xBF) : (PRT5DR|0x40))
#define Port5_7(b) (PRT5DR = (b==0) ? (PRT5DR&0x7F) : (PRT5DR|0x80))
**/

boot.tpl: (locate and alter the matching blocks as shown below)
    org 24h                ;PSoC Block DBB01 Interrupt Vector
;   `@INTERRUPT_9`
    ljmp _SampleINT
    reti

    org 34h                ;PSoC Block DBB11 Interrupt Vector
;   `@INTERRUPT_13`
    ljmp _FreqINT
    reti

```

Code 1. PSoC C Code

About the Authors

Name: Chris and Vincent Paiano

Title: B.S., Computer Engineer and
Electronic Engineer

Background: 22+ years programming/computer
experience.

40+ years electronics/design and
troubleshooting experience.

Contact: psoc@chrispaiano.com
engineering@chrispaiano.com

Cypress MicroSystems, Inc.
2700 162nd Street SW, Building D
Lynnwood, WA 98037
Phone: 800.669.0557
Fax: 425.787.4641

<http://www.cypress.com/> / <http://www.cypress.com/support/mysupport.cfm>

Copyright © 2004 Cypress MicroSystems, Inc. All rights reserved.

PSoC™, Programmable System-on-Chip™, and PSoC Designer™ are trademarks of Cypress MicroSystems, Inc.

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

The information contained herein is subject to change without notice. Made in the U.S.A.