



Application Note

AN2374

Wide Dynamic Range Programmable PSoC[®] Counter Timer

Author: J. Jayapandian
Associated Project: Yes
Associated Part Family: CY8C27443
Software Version: PSoC Designer 4.2
Associated Application Notes: None

Abstract

Pulse counting is an essential application requirement in industry and R&D laboratories. Counting applications count pulses for a preset time period. Counter and timer units are crucial components of most embedded systems and are included in many microcontrollers as one or two counters or timers. In traditional microcontrollers, the built-in modular functions such as counters, timers, ADCs, and DACs are fixed components, but all are programmable. In a PSoC device, the placement of the user module functions is a design choice that you can change and reconfigure; not frozen hardware configurations like a microcontroller. This Application Note details a PSoC Designer™ project that provides pulse counts for periods of a few milliseconds to several minutes. With the addition of a timer interrupt, it can go up to several hours. It uses a virtual instrument program so that you can control the program and display it on a personal computer.

Introduction

Counters and timers are the core of digital electronics and form the basic circuit in several new applications. The microprocessor central processing unit (CPU) is based on counters and timers. For systems ranging from simple to complex, counters and timers are a basic need. The advent of programmable intelligent controllers brings about innovative embedded component designs that use counters and timers as the main modules.

Microcontrollers available in the market with different configurations enable designers to provide embedded solutions for design automation. But single-chip implementations such as ADCs, DACs, counters, and timers are frozen in design. It is not possible to modify the the number of bits, resolution, clock, and input/output pin connectivity. Modules available on the device must be used as is. This limitation causes manufacturers to produce a wide variety of devices. The variety available forces designers to spend a great deal of time examining specifications to find the right part for their design.

Programmable System-on-Chip™ (PSoC) devices and PSoC Designer provide a more flexible solution for creating the required functionality on a single device.

This Application Note describes how a 16-bit Counter User Module and a 16-bit Timer User Module, with its interrupt program routine, are used to create a counter with a wide dynamic range. A standalone 16-bit counter can only count up to 65,536 and a 16-bit timer is similarly limited in the range of time it can measure. With PSoC, you can choose to add an interrupt upon terminal count to reload the counter and start counting again, which increases the range of the counter. The design tools play a vital role in sourcing user module functionality and providing interaction with the built-in M8C microcontroller. A fixed microcontroller does not have the flexibility to dynamically set the baud clock for UART communication. With PSoC devices, you can choose the baud rate to fit your design specifications.

In this Application Note, PSoC Designer uses all eight digital blocks for a counter, timer, and serial communication (UART) for interfacing with the PC as shown in Figure 1. Figure 2 shows the pin configuration of the CY8C27443 28-pin PSoC device. You can set a 16-bit number to preset the timer. This gives you a wide timer range from milliseconds to several minutes. You can extend the timer to several hours using the timer interrupt service routine. Similarly, you can extend a 16-bit counter capable of counting to 64K up to several million using the counter interrupt service routine, which registers the overflow as a carry bit. The counter register is reloaded whenever the counter reaches the terminal count.

Figure 1. PSoC Designer Project User Module Placement

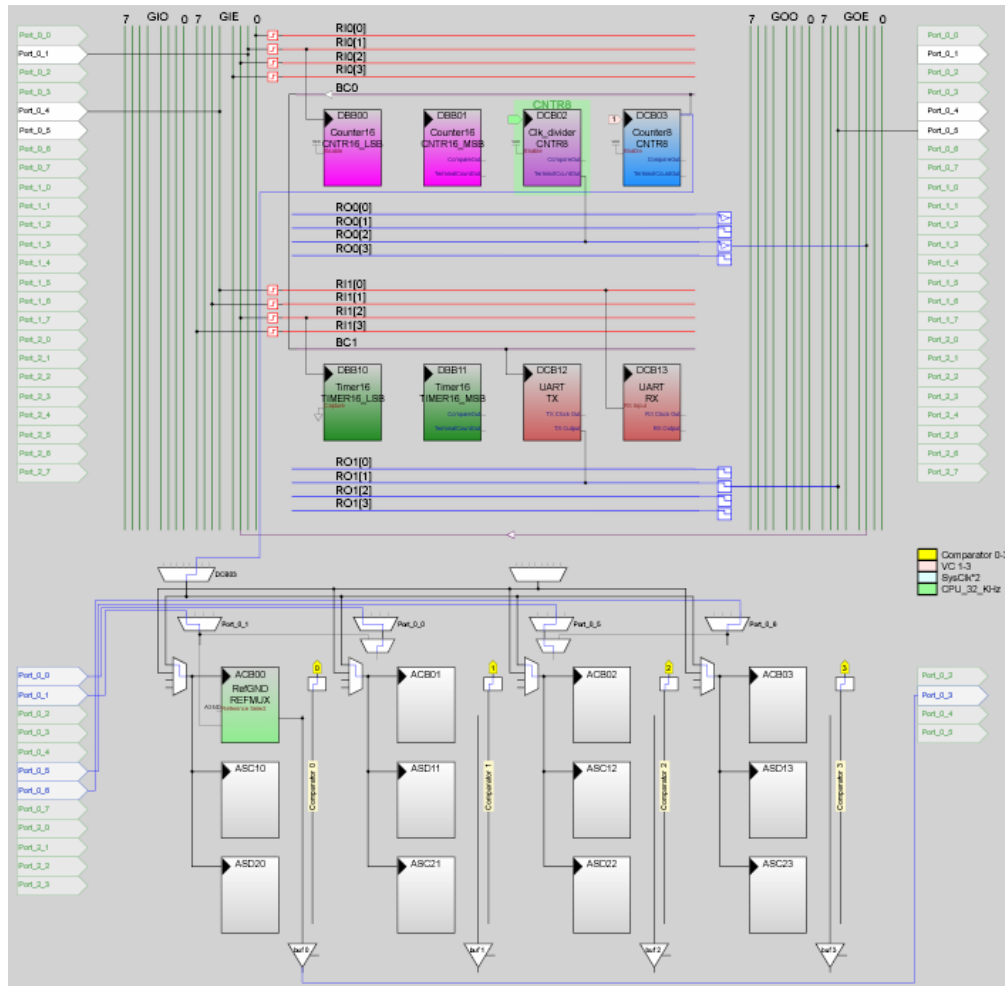
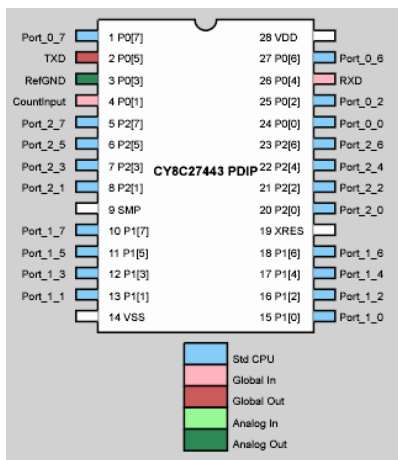


Figure 2. CY8C27443 28-Pin PSoC Device Pinout



The processor clock is divided by an 8-bit counter (placed in DCB03) to generate a clock for the serial communication UART. The baud rate is set to 115,200 in this design application. The clock divider user module, which occupies one digital block (DCB02), is another 8-bit counter used for basic clock generation. Dividing the CPU_32KHz clock by 250 provides a 128 Hz clock for the Timer16 User Module, which occupies two digital blocks (DEB10 LSB and DEB11 MSB). You can set the Timer16 to get wide dynamic range of time from milliseconds to several seconds. The interrupt of the Timer16 is used to set the carry bit (the variable `Ticker` in `main.c` in Appendix A: C Source on page 4) whenever Timer16 overflows.

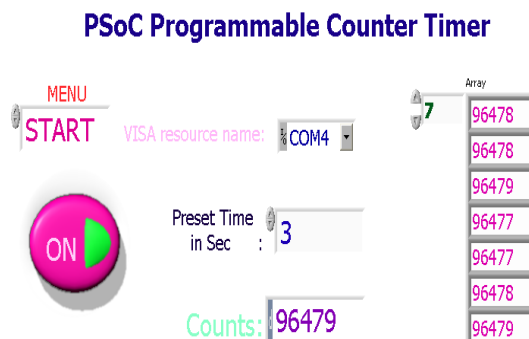
To count events, the Counter16 User Module is placed in another two digital blocks (DBB00 for LSB and DBB01 for MSB) in the project. Counter16 is capable of counting up to 65,535 (64K). If you invoke the interrupt whenever the counter reaches terminal count and register it as a carry bit (the variable `Of_bit` in `main.c`) you extend the counting capacity beyond the 64K limit. The remaining two digital blocks are used for serial communication with the PC through the UART. The transmit block (TX) is placed in DCB12 and the receive block (RX) is placed in DCB13 for serial communication. In this application, the UART baud rate is set to 115,200.

A single analog block, ACB00, is used for reference ground. Pin configurations for the PSoC device in this application are shown in Figure 2. Port0_1 is assigned to the input count pulse with reference to the `ref_ground` on Port0_3. The serial communication transmits through Port0_5 and receives through Port0_4. For testing purposes, you can assign the `CPU_32_kHz` clock to the clock input on Port0_1 connected to Counter16. To do so, right click the clock input in the Device Editor Interconnect View and select the `CPU_32_kHz` clock. It provides a count of 32 kHz for one second.

The PSoC device interfaces with the PC through the serial port at the baud rate of 115k using a virtual instrument (VI) program designed and built with LabVIEW™ Express 7.1, a graphical language package. Figure 3 shows the front panel diagram of the virtual instrument control program. You connect the PSoC design to the PC's serial port and run the VI control program on the PC. The VI program allows you to set counter and timer values, start the counter and timer modules, and see the count values on the screen. You can also view an XY plot of count over time if you wish.

The Measurement & Automation Explorer in LabVIEW detects the serial port and is called by the VI program. In this example, the serial port COM4 is derived from a USB-to-serial connector.

Figure 3. Virtual Instrument Program Front Panel Diagram



The ON/OFF button is provided to terminate the program. The Array keeps the counted value in its registers. You can save these values with the File >> Save command. As a simple test, you can run PSoC Designer and use Microsoft® HyperTerminal to visualize the function.

Conclusion

This single-chip embedded design application explains how the digital blocks and communication blocks are used in the PSoC device. You can use this design in any industrial or lab application that needs a flexible timer or counter.

Appendix A: C Source

```

/* The main C program for the PSoC counter timer. This application design
 * provides pulse counts over a wide range of preset time, from a few milliseconds to several
 * minutes. You can use the carry output of Timer16 to extend this range to several hours */

#include <m8c.h>
#include "PSoCAPI.h"
#include "stdlib.h"
#include "uart.h"
#include "Timer16.h"
#include "Clk_divider.h"
#include "Ports.h"

#define CR 0x0D
#define LF 0x0A
#define HEXBASE 16

char cIndex;
long int Countdata, Countvalue;
char Count[4];
char tmrw[4];
char tmrh[4];
BYTE bData;
unsigned int value, Ticker;
int Of_bit;

void Set_timer(void);
void Ascii2Hex(void);
void Counter_read(void);
void DoTask(void);
void Timer16_ISR(void);
void Counter16_ISR(void);

#pragma interrupt_handler Timer16_ISR // Timer16_ISR
#pragma interrupt_handler Counter16_ISR // Counter16_ISR

void Initialize (void)
{
    UART_IntCntl(UART_ENABLE_RX_INT);
    Counter8_WritePeriod(26); // 26 Sets up baud rate generator
    Counter8_WriteCompareValue(13); // 13 Turns on baud rate generator
    Counter8_EnableInt();
    Counter8_Start();
    UART_Start(UART_PARITY_NONE);
    RefGND_Start(RefGND_HIGHPOWER); // Turn on power to the CT block
    RefGND_RefSelect(RefGND_AGND); // Apply AGND to ABUS2.
    Clk_divider_WritePeriod(0xFA); /* Clk_divider divides the 32kHz clock by 250
    * and provides 128 Hz clock to Timer16. */

    Clk_divider_WriteCompareValue(0x7D);
    Clk_divider_EnableInt(); // Ensures the interrupt is enabled
    Timer16_WriteCompareValue(00001);
    Count[0] = 0;
    Counter16_WritePeriod(0xFFFF); // Reloads Counter16 when it overflows
    Counter16_EnableInt(); // Counter16 interrupt enabled
    Timer16_EnableInt(); // Timer16 interrupt enabled
    M8C_EnableGInt(); // Turn on interrupts
}

```

```

signed char ExtractHex(unsigned char Val)
{
    if(Val < '0')                // If the value is invalid return error
        return -1;
    else if(Val <= '9')          // If the value is between ASCII 0 and 9
    { Val -= '0';                // then return binary 0-9
      return Val;
    }
    Val &= 0xdf;
    if (Val < 'A')               // Return an error if the value is
        return -1;              // between ASCII 9 and A
    else if (Val < 'G')         // If the value is between ASCII A and F
    { Val -= 'A';               // then return binary 10 - 15
      Val += 10;
      return Val;
    }
}

/* Timer16 interrupt service increments the variable Ticker for every overflow during
 * terminal count. The overflow is recorded as a carry bit by this interrupt service routine.
 * For every terminal count, Timer16 is loaded with 0xFFFF again and starts count. This way
 * you can extended the time set period to several hours. Note: the carry bit count has not
 * been used in this application. The preset time maximum in this application is
 * 8.5 seconds. */

void Timer16_ISR(void)
{
    ++Ticker;
    Timer16_Stop();
    Counter16_Stop();
    Clk_divider_Stop();
    Counter16_DisableInt();
    Counter_read();
    Counter16_WritePeriod(0xFFFF);
}

/* The variable Of_bit is the carry bit used by the Counter16 interrupt service. If the value
 * of Of_bit is greater than 0 then the count value is larger than 65535. This means the
 * actual count is Of_bit * 65535 + Counter_read(). The interrupt service routine increments
 * the variable Of_bit for every overflow and thus extends the range of the 16-bit counter
 * into the millions. */

void Counter16_ISR(void)
{
    ++Of_bit;
}

void Counter_read(void)
{
    UART_PutSHexInt(Of_bit);      /* Sets the carry bit. The value of Of_bit
 * is sent to the PC via the serial port
 * along with the existing count. */

    Countdata=Counter16_wReadCounter();
    Countdata &=0xFFFFF;
    Countvalue = ~ Countdata;
    itoa( Count, Countvalue, HEXBASE);
    cIndex = 0;
}

```

```

/* Sends the value of Count and Of_bit to the serial port */

while (Count[ cIndex])
{
    while (!(bUART_ReadTxStatus() &
        UART_TX_BUFFER_EMPTY) ) {}
    UART_SendData( Count[ cIndex]);
    cIndex++;
}

if ( Count[0] != 0)
{
    while (!(bUART_ReadTxStatus() & UART_TX_BUFFER_EMPTY)) {}
    UART_SendData( LF);
    while (!(bUART_ReadTxStatus() & UART_TX_BUFFER_EMPTY)) {}
    UART_SendData( CR);
    Count[0] = 0;
    cIndex = 0;
}
}

void Set_Timer(void)
{
    int i;
    unsigned int val1, val2, val3, val4;
    i = 0;

    UART_CPutString("Set Time in Sec:");
    UART_IntCnt1(UART_DISABLE_RX_INT);
    while(i < 4)
    {
        tmrw[i]= UART_cGetChar();
        tmrh[i]= ExtractHex(tmrw[i]);
        i++;
    }
    val1 = tmrh[0]<<12;
    val2 = tmrh[1]<<8;
    val3 = tmrh[2]<<4;
    val4 = tmrh[3];
    value = val1|val2|val3|val4;
    UART_PutSHexInt(value);
    Timer16_WritePeriod(value);
    Counter16_WritePeriod(0xFFFF);
}

void main()
{
    BYTE bData;
    unsigned int value;
    UART_CmdReset();
    Initialize();

    while (1)
    {
        bData = UART_bReadRxData();
        bData &= 0xdf;
    }
}

```

```

if ((bData >= 'G'))
{
    switch (bData)
    {
        case 'T'      :   Set_Timer();
                        UART_CPutString("\r\nTime Set OK\r\n");
                        break;

        case 'R'      :   Counter16_Stop();           // Stop Counter16, Timer16, Clk_divider
                        Timer16_Stop();
                        Clk_divider_Stop();
                        Of_bit = 0;
                        Ticker = 0;
                        break;

        case 'S'      :   Clk_divider_Start();       // Start Counter_Timer
                        Timer16_Start();
                        Counter16_EnableInt();
                        Counter16_Start();
                        break;

        default       :   UART_CPutString("\r\nWelcome to PSoC Counter_Timer V1.1\r\n");
                        break;
    }
    UART_CmdReset();
}
}
}

```

About the Author

Name: J. Jayapandian
Title: Scientific Officer – F
 Materials Science Division
 Indira Gandhi Centre for Atomic Research
 Kalpakkam – 603 102.
 Tamil Nadu, India.

Background: Specializes in analog, digital and embedded designs. Several design projects were carried out for various experimental automation requirements using the virtual instrument programming technique.

Contact: jjpandian@gmail.com
jjp@igcar.gov.in

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

© Cypress Semiconductor Corporation, 2006. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.