Switch Debouncer and Glitch Filter with PSoC® 3, PSoC 4, and PSoC 5LP

Author: Mark Ainsworth

Associated Project: Yes

Associated Part Family: CY8C3xxx, CY8C42xx, CY8C5xxxLP

Software Version: PSoC Creator™ 3.0 and higher

Related Application Notes: For a complete list of the application notes, click here.

AN60024 introduces the concepts of switch debouncing and glitch filtering for digital input signals, and shows how to create several debounce and filter solutions for PSoC® 3, PSoC 4, and PSoC 5LP, using PSoC Creator™.

Contents

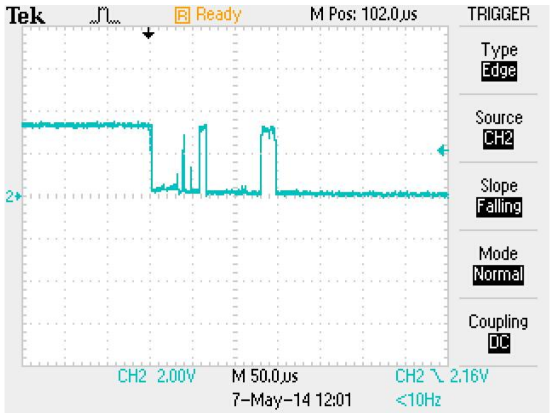
1 Introduction

CapSense® and TrueTouch® notwithstanding, many products still have mechanical buttons or switches. When a switch is pressed or released, its output can oscillate for a brief period, as Figure 1 shows.

In some applications, these oscillations may cause other parts of the system, such as the CPU, to falsely detect multiple press and release events. This in turn can cause erratic and unexpected system behavior. Imagine what might happen if, for example, the switch is the 'Speed Up' button on a treadmill.

Filtering out these oscillations is known as switch debouncing. You could use a simple RC filter, but why pay for extra components when PSoC devices can do the job easily and in many ways, using minimal device resources? And, an RC filter is fixed, while with PSoC you can easily and dynamically adapt to different switch or button characteristics.

Figure 1. Scope Shot Showing Switch Bounce on Transition from High to Low Voltage



This application note shows several ways to take advantage of the flexibility of PSoC. With PSoC, switch debouncing can be done using hardware, software, or both. Glitch filtering, which is similar to switch debouncing, is also demonstrated.

You will also see how fast and easy it is to use PSoC Creator to produce many different designs. Several PSoC Creator demonstration projects are attached to this application note. They are designed to work with the CY8CKIT-001 PSoC Development Kit, but can be easily adapted to work with most other kits.

This application note assumes that you have some basic familiarity with PSoC 3, PSoC 4 or PSoC 5LP devices and the PSoC Creator IDE. If you are new to PSoC, you can find introductions in the following application notes:

- AN54181, Getting Started with PSoC 3
- AN79953, Getting Started with PSoC 4
- AN77759, Getting Started with PSoC 5LP

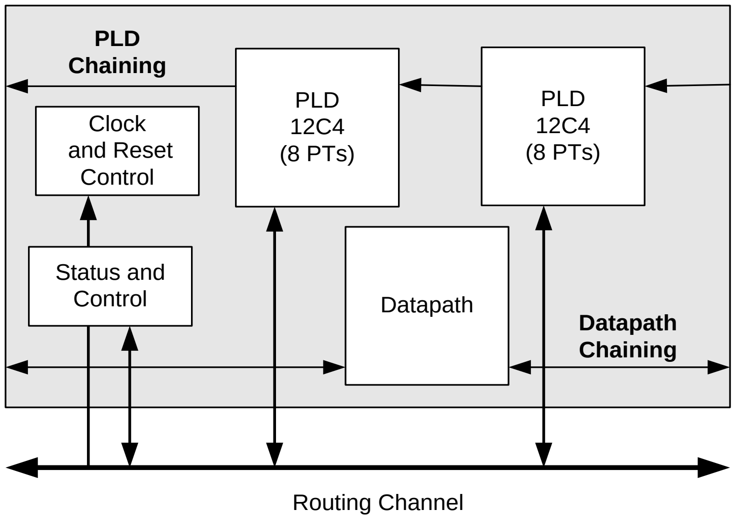
If you are new to PSoC Creator, see the PSoC Creator home page.

1.1 Universal Digital Blocks (UDBs)

Many of the designs shown in this application note make use of the Universal Digital Blocks (UDBs) available in PSoC.
A PSoC device has as many as 24 UDBs. Each UDB contains:

- two small programmable logic devices (PLDs),
- a datapath module containing a programmable 8-bit ALU,
- and other registers and functions, as Figure 2 shows.

Figure 2. UDB Block Diagram



UDBs, as well as PSoC's Digital System Interconnect (DSI) routing fabric, give you the flexibility to design complex custom peripheral functions. This allows you to optimize your design at the system level as well as significantly reduce CPU usage. For more information on UDBs and PSoC digital design in general, see the following application notes:

- AN81623, PSoC Digital Design Best Practices
- AN82250, PSoC Implementing Programmable Designs with Verilog
- AN82156, PSoC Designing PSoC Creator Components with UDB Datapaths

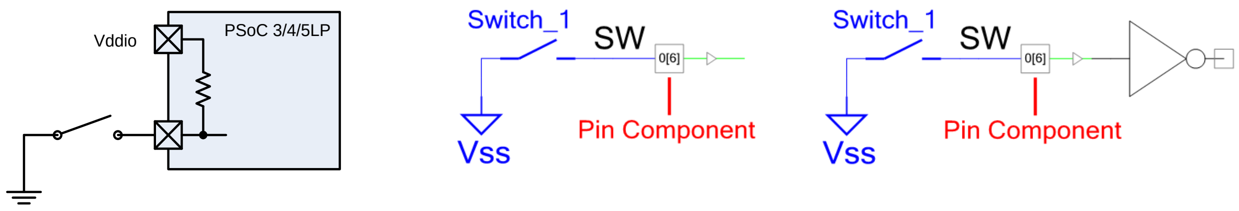
The PSoC Creator IDE offers an extensive library of preprogrammed peripherals. It also enables you to create your own custom designs.

2 I/O Pin Setup

Before we look at switch debouncing, it may be helpful to review how best to connect a switch or button to a PSoC I/O pin. A button typically shorts to ground when pressed and opens when released. To get valid digital high voltages, a pull-up resistor is required, as the leftmost diagram in Figure 3 shows.

Each PSoC I/O pin has an internal pull-up resistor. With PSoC Creator, you can use the Digital Input Pin Component to enable the pull-up; Figure 3 also shows some example PSoC Creator schematics.

Figure 3. Grounded Switch Circuit and Corresponding Example PSoC Creator Schematics

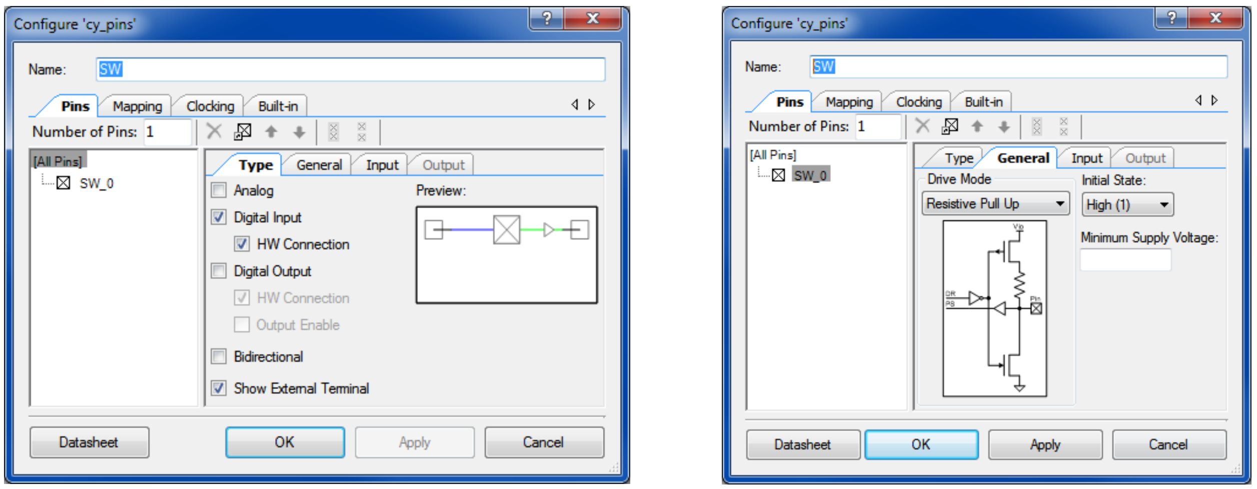


You can read the pin value using firmware or hardware. The pin value is logic ‘0’ when the switch is pressed, and logic ‘1’ when released. If you want to reverse the polarity, you can do the inversion in firmware or in hardware. In hardware just add an inverter, as the rightmost schematic in Figure 3 shows.

After placing the Pin Component on your PSoC Creator project schematic, double-click the Component to configure it. A configuration dialog is displayed, as Figure 4 shows.

Enter a name for the Pin; in this case it is “SW”, for “switch”. Select the options on the **Type** and **General** tabs. Checking the **HW Connection** and **Show External Terminal** boxes is optional, depending on your application. To enable the pull-up resistor, set the **Drive Mode** to **Resistive Pull Up**, and the **Initial State** to **High (1)**. For more information, see the Pins Component datasheet.

Figure 4. Digital Input Pin Component Configuration



Name: SW
Number of Pins: 1
[All Pins]
SW_0

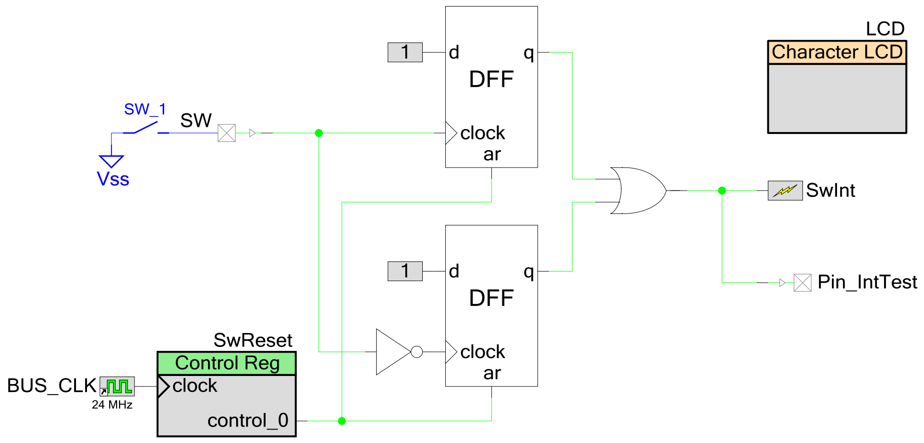
- [ ] Analog
- [x] Digital Input
  - [x] HW Connection
- [ ] Digital Output
  - [x] HW Connection
  - [ ] Output Enable
- [ ] Bidirectional
- [x] Show External Terminal

Name: SW
Number of Pins: 1
[All Pins]
SW_0
Drive Mode: Resistive Pull Up
Initial State: High (1)
Minimum Supply Voltage:

3 A Switch Bounce-Sensitive Design

Figure 5 shows a poor design – it is sensitive to switch bounce.

Figure 5. Schematic Showing a Design that is Overly Sensitive to Switch Bounce



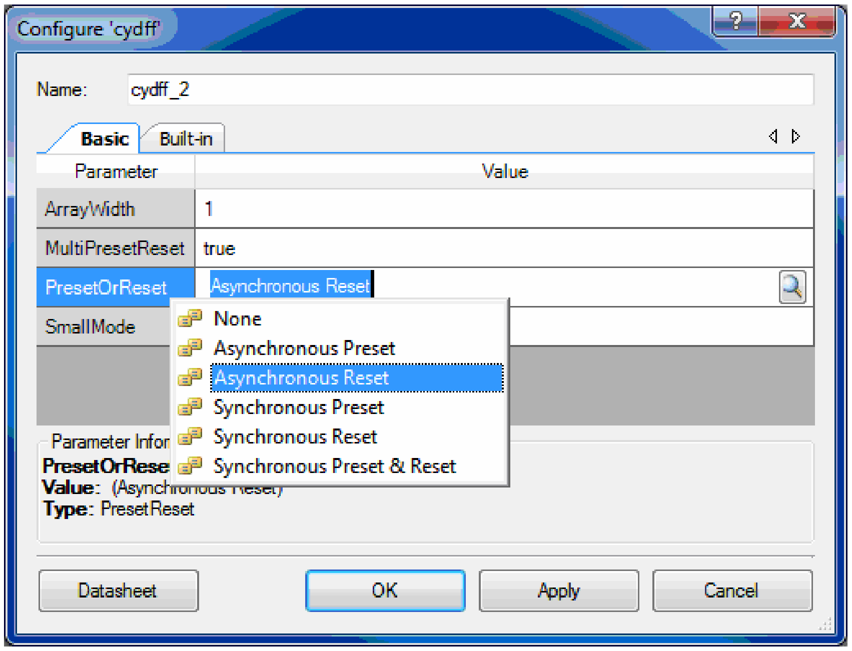
The positive and negative edges from the input pin are captured by two digital flip flops (DFF) and trigger an interrupt. Code 1 on page 5 shows how to handle the interrupt. The interrupt handler code does the following:

- Resets the DFFs by writing to the Control Reg Component ‘SwReset’. The Component is configured in pulse mode; writing a ‘1’ to it generates a single pulse that resets the DFFs.
- Increments a global ‘count’ variable

The loop in main() continually displays the value of ‘count’ on the LCD.

Note: The DFF Component by default does not show the preset or reset terminals (‘ar’ in Figure 5). To enable these terminals, configure the Component as Figure 6 shows.

Figure 6. DFF Component Configuration



Configure 'cydff'

Name: cydff_2

Parameter	Value
ArrayWidth	1
MultiPresetReset	true
PresetOrReset	Asynchronous Reset
SmallMode	

Parameter Infor
PresetOrRese
Value: (Asynchronous Reset)
Type: PresetReset

Code 1. Interrupt-Based Response to Input Pin Transitions

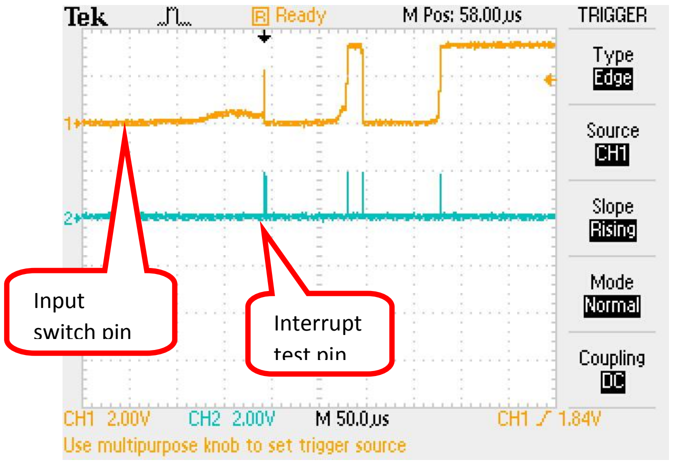
```
uint8 count; /* # of transitions of input pin 'SW' */

CY_ISR(SwInt_ISR)
{
    SwReset_Write(1); /* clear interrupt source */
    count++;
} /* end of SwInt_ISR() */

void main()
{
    uint8 temp; /* local copy of count variable */

    /* Initialization code */
    . . .

    for(;;) /* do forever */
    {
        /* Grab a copy of the shared count variable, and display the copy.
         * This ensures the interrupt handler will not change the count
         * variable while it is being displayed.
         */
        CYGlobalIntDisable; /* macro */
        temp = count;
        CYGlobalIntEnable;  /* macro */
        LCD_Position(0, 14); /* row, column */
        LCD_PrintHexUint8(temp);
    }
} /* end of main() */
```

The display increments with each button press and button release. However if you run the attached project ‘A_DebounceNone’, you see that the display sometimes increments several times on a single button press or release. The reason becomes apparent if you connect an oscilloscope to both the input switch pin and the interrupt test pin; see Figure 7.

The system is running at high speed. This allows each interrupt to be processed in a very short time relative to the transitions rate. Thus, instead of a single edge, every edge is detected and processed. (If you look closely at Figure 7, you can see that even the very narrow first pulse generates two interrupts.)

To solve this problem, you must use a switch debouncing technique. The remainder of this application note shows various ways to do so.

Figure 7. Scope Shot Showing Switch Bounce and Interrupt Response



Note that PSoC pins are capable of directly generating an edge-triggered interrupt through the port interrupt control unit (PICU). However, the PICU cannot be used in this application because the interrupts occur on pin transition edges and not at regular sample intervals. Thus, a PICU-based design is subject to the same switch bounce problems noted in Figure 5.

The PICU can also cause pin transitions to wake up the PSoC from a low-power mode, such as Sleep or Hibernate. In this case, you can create a design in which a debounce operation is suspended when entering a low-power mode and restarted after wakeup. For more information on PSoC power modes, see application note AN77900 for PSoC 3 or PSoC 5LP, or AN86233 for PSoC 4.

4 Switch Debouncing by Sampling

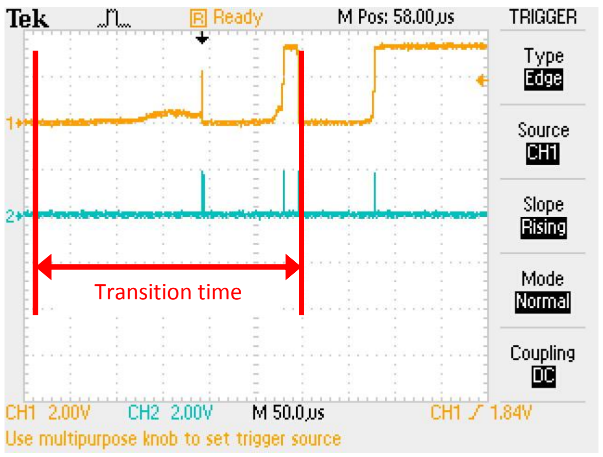
There are many ways to do switch debouncing, but all of them involve sampling the input pin at a periodic rate.

Set the sample period to be greater than the anticipated transition time of the signal, that is, the time it takes for the signal to stabilize at the new state when the switch is opened or closed. During the transition time, the state of the signal is essentially unknown – at any time it could be either 0 or 1. You should not sample more than once during that period or you may detect an extra transition.

The maximum sample rate is the inverse of the maximum transition time; remember that you may need to check both the low-to-high and high-to-low transition times.

The example in Figure 8 shows that the maximum transition time from pressed to released is ~300 µs. In this case, the sampling rate should not exceed 1 / (300 µs), or 3.3k samples per second (sps).

Figure 8. Scope Shot Showing Switch Transition Time



In practice, set the sample rate to be much lower than the expected transition time, but fast enough that the system is responsive when the switch is opened or closed. A rate of 10 to 200 sps is usually appropriate.

5 Switch Debouncing Using Software

The simplest way to sample a switch is to poll the input pin, that is, program the CPU to read the pin's input value at regular time intervals. In Code 2 on page 7, the pin is sampled at an interval that is controlled by the CyDelay() function provided by PSoC Creator.

The easiest way to detect a transition on the input is to use two variables, for current and previous values of the pin, and compare them for transition events.

The example in Code 2 monitors just one pin, in bit 0 of each element of the 'switches' array. In this case, we compare the array elements in their entirety, with the assumption that bits 1 to 7 are always zero. If you need to monitor multiple pins then you can either:

- use a separate pair of variables for each pin, or
- use a single pair of variables, with a bit defined for each pin

There are tradeoffs in code size, execution speed, and RAM memory usage, for each method.

With PSoC, you can monitor a pin in software and hardware at the same time. The attached project 'B_DebounceSwPoll' adds to project 'A_DebounceNone' to show both raw (unfiltered) and filtered counts.

Code 2. Periodic Sampling of an Input Pin

```c
void main()
{
    /* Initialization code */
    . . .

    /* Init switch variables */
    uint8 switches[2] = {0, 0}; /* [0] = current, [1] = previous */
    switches[0] = switches[1] = SW_Read(); /* 0 = pressed, 1 = not pressed */

    /* Init display */
    LCD_Start();
    LCD_Position(0, 0); /* row, column */
    LCD_PrintString("Raw Count   = ");
    LCD_Position(1, 0); /* row, column */
    LCD_PrintString("Filt. Count = ");

    for(;;) /* do forever */
    {
        /* Place your application code here. */
        /* Grab a copy of the shared count variable, and display the copy.
         * This is so that the interrupt handler won't change the count
         * variable while it's being displayed.
         */
        CyGlobalIntDisable; /* macro */
        temp = count;
        CyGlobalIntEnable;  /* macro */
        LCD_Position(0, 14); /* row, column */
        LCD_PrintHexUint8(temp);

        /* Periodically sample the input pin, and display the filtered count */
        CyDelay(50); /* msec */

        /* Update the current and previous switch read values */
        switches[1] = switches[0];
        switches[0] = SW_Read();

        /* Increment counter if a switch transitions either way */
        if (switches[0] != switches[1])
        {
            filtered_count++;
        }

        /* Display the current value in filtered count variable */
        LCD_Position(1, 14); /* row, column */
        LCD_PrintHexUint8(filtered_count);
    }
} /* end of main() */
```

With code that must do many tasks, it may be difficult to poll a pin at regular intervals. In that case, an alternative is to sample by using a periodic interrupt. The easiest way to do this, for PSoC 3 and PSoC 5LP, is to connect a Clock Component directly to an Interrupt Component, as Figure 9 shows.

PSoC 4 does not support routing a clock directly to an interrupt. The easiest workaround is to double the clock frequency and route it through a TFF Component, as Figure 10 shows. The TFF Component output is half the frequency of the input clock.

Figure 9. Components for Periodic Interrupts

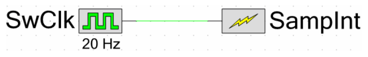
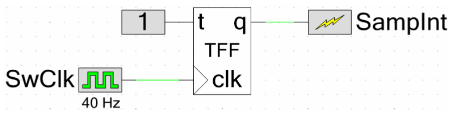
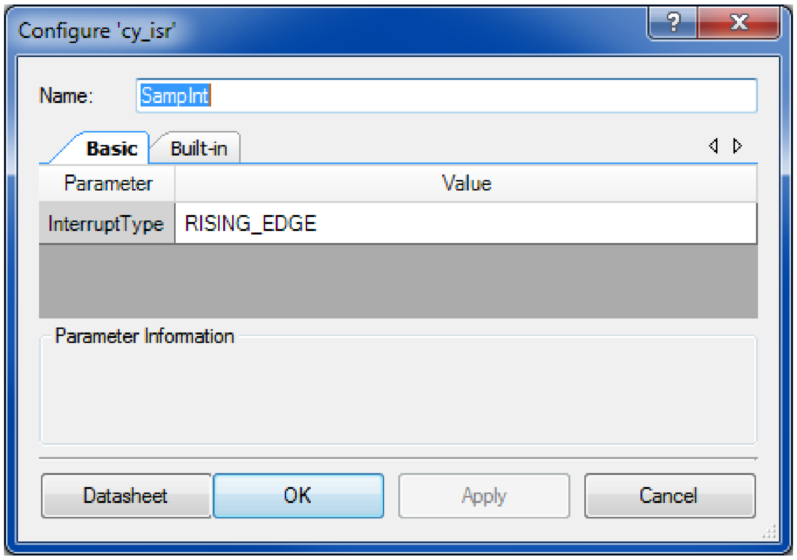


Figure 10. PSoC 4 Components for Periodic Interrupts



In both cases, the Interrupt Component 'SampInt' must be set as a rising edge triggered interrupt, so that the interrupt is triggered only once per clock period – see Figure 11.

Figure 11. SampInt Configuration



Name: SampInt

Parameter	Value
InterruptType	RISING_EDGE

In this design, as Code 3 on page 9 shows, the sample interrupt handler does the sampling and edge detection. This makes the main() function much simpler. The attached project 'C_DebounceSwInt' contains similar code, to show both raw (unfiltered) and filtered counts

For more information on interrupts in PSoC, see AN54460, PSoC Interrupts.

Code 3. Interrupt-Based Sampling of an Input Pin

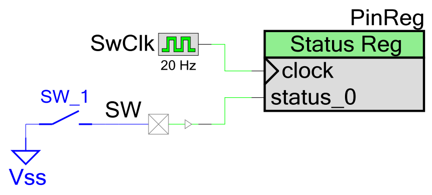
```
uint8 filtered_count; /* # of filtered transitions of input pin 'SW' */

CY_ISR(SampInt_ISR)
{
    static uint8 init = 0;
    static uint8 switches[2]; /* [0] = current, [1] = previous */
    uint8 temp;

    /* interrupting from clock, no interrupt source to clear */

    temp = SW_Read(); /* read the pin value */

    /* Switch variable initialization should be done only once, the first
     * time this function is called.
     */
    if (!init)
    {
        init = 1;
        switches[0] = switches[1] = temp;
    }
    else
    { /* Increment the count if the switch transitions either way. */
        switches[1] = switches[0];
        switches[0] = temp;
        if (switches[0] != switches[1])
        {
            filtered_count++;
        }
    }
} /* end of SampInt_ISR */


void main()
{
    uint8 temp; /* local copy of shared count variable */

    /* Initialization code */
    . . .

    for(;;) /* do forever */
    {
        . . .

        /* Grab a copy of the shared filtered count variable, and display the
         * copy. This is so that the interrupt handler won't change the count
         * variable while it's being displayed.
         */
        CYGlobalIntDisable /* macro */
        temp = filtered_count;
        CYGlobalIntEnable  /* macro */
        LCD_Position(1, 14); /* row, column */
        LCD_PrintHexUint8(temp);
    }
} /* end of main() */
```

6 Switch Debouncing Using Hardware

PSoC makes it easy to do switch debouncing in hardware instead of software, which can reduce CPU usage. The simplest way to do this is to poll with a clock and a status register, as Figure 12 shows.

The code for this design is the same as in Code 2 except that you read from PinReg_Status instead of SW_Read(), and the CyDelay() function is removed. See attached project ‘D_DebounceHwReg’.

Figure 12. Schematic of a Design with Register-based Switch Debouncing

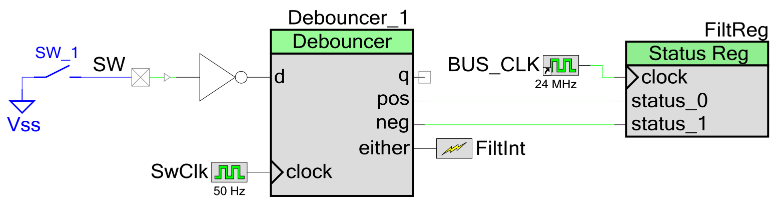


6.1 Debouncer Component

You can reduce CPU usage even more by moving both the input pin polling and the comparison functions into hardware. The PSoC Creator Component library includes a Debouncer Component that is designed for this purpose.

Figure 13 shows a design using the Debouncer Component. As shown previously in Figure 11 on page 8, make sure that the FiltInt Component is configured for RISING_EDGE mode.

Figure 13. Design Using a Debouncer Component



Note: The output of the Debouncer is reset to ‘0’ at device reset, and the input switch is read as ‘1’ when it is open, which is the typical case. This results in a single false switch release event at initialization. An inverter on the input pin solves this problem.

This design also includes a Status Reg Component, so that the CPU can read the type of edge that caused the interrupt. The Status Reg Component is configured for “sticky 1” mode. In this mode, when a bit is set to ‘1’, it stays that way until the register is read by the CPU or DMA, at which point it is reset to ‘0’.

Most of the debounce functions are now performed in hardware instead of software, so the interrupt handler is simplified; see Code 4. The code in main() remains unchanged except for some initialization.

Code 4. Interrupt-based Response to Hardware Debouncer Component

```
uint8 filtered_count; /* # of filtered transitions of input pin 'SW' */

CY_ISR(FiltInt_ISR)
{
    /* No need to clear any interrupt source; interrupt component should be
     * configured for RISING_EDGE mode.
     */
    /* Read the debouncer status reg just to clear it, no need to check its
     * contents in this application.
     */
    FiltReg_Read();

    filtered_count++;
} /* end of FiltInt_ISR() */
```

It is not necessary to connect a Debouncer Component output terminal to an Interrupt Component. Depending on the application, you can connect any of the terminals to other digital logic or to a DMA channel.

You can easily add support for multiple pins by changing the Component’s signal width parameter.

7 Glitch Filtering

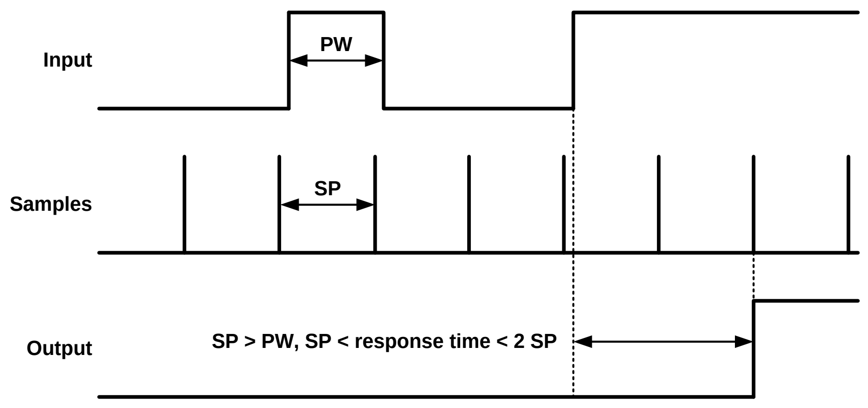
Glitch filtering is similar to switch debouncing but supports a slightly different application. In switch debouncing, you filter out all edges except the single one that you want. In glitch filtering, you remove unwanted glitch pulses from a signal.

Note that glitches are not necessarily associated with switches; they can occur on lines carrying signals from sources such as RF receivers. Electrical or in some cases even mechanical interference can trigger an unwanted glitch pulse from a receiver.

The switch debouncing methods shown previously “mostly” work for glitch filtering but never 100 percent, because you might sample just when a glitch is occurring. Instead, the glitch filter algorithm outputs a ‘1’ only when the current and previous N samples are ‘1’, and a ‘0’ only when the current and previous N samples are ‘0’. Otherwise, the output remains unchanged from its current value.

If N = 1 then the time between two successive samples (SP in Figure 14) must be greater than the maximum pulse width that can be filtered (MaxPW). Because it takes at least two samples for the output to change, the response time, or filter delay, is between 1 and 2 sample periods.

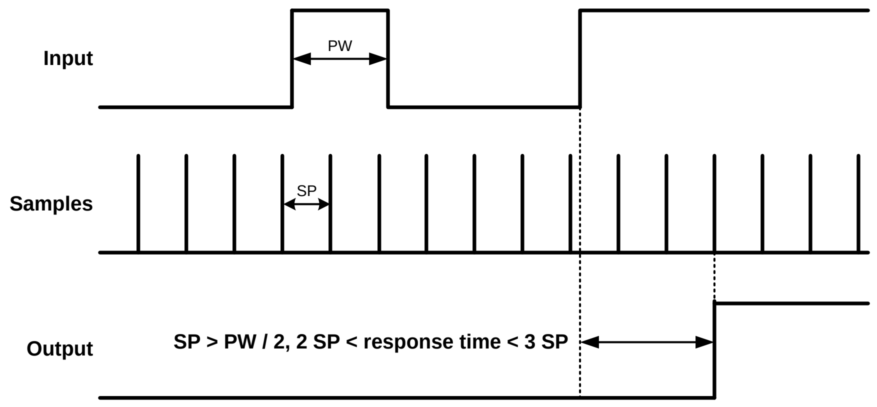
Figure 14. Glitch Filter Performance for N = 1



The response time can also be expressed in terms of the MaxPW. In the case for N = 1, because the sample period can be as low as the maximum filtered pulse width, the response time can range from 1 to 2 times the MaxPW.

You can use larger values of N to get more deterministic and possibly shorter response times. For example, if N = 2, the response time is between 2 and 3 sample periods. However the sample period can be shorter – the MaxPW must be less than 2 sample periods – and thus the response time can be expressed as 1 to 1.5 times the MaxPW. See Figure 15.

Figure 15. Glitch Filter Performance for N = 2



If N = 3 then the response time can range from 1 to 1.333 times the MaxPW, and so on. The general response time equations are:

Equation 1. Glitch Filter Response Time

$$\mathrm{N} \cdot SP \;<\; response\ time \;<\; (\mathrm{N}+1)\ SP$$

$$SP \;>\; \frac{MaxPW}{\mathrm{N}}$$

$$MaxPW < \; response\ time \;<\; \left(1+\frac{1}{\mathrm{N}}\right) MaxPW$$

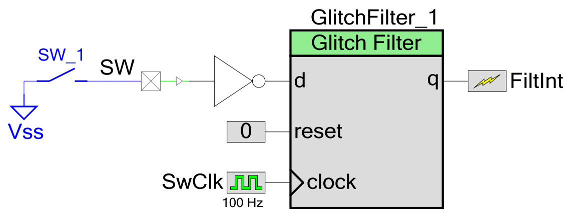
where SP = sample period, and MaxPW = maximum filtered pulse width.

Most glitch filters are designed with N = 1, 2, or 3. Note that in a hardware-based design, larger values of N require more PSoC digital resources to implement.

Figure 16 shows a design using a PSoC Creator Glitch Filter Component. The clock frequency is the sample period. As shown previously in Figure 11 on page 8, make sure that the FiltInt Component is configured for RISING_EDGE mode.

Note: The output of the Glitch Filter is reset to ‘0’ at device reset, and the input switch is read as ‘1’ when it is open, which is the typical case. This results in a single false release event at initialization. An inverter on the input pin solves this problem.

Figure 16. Design Using a Glitch Filter Component



It is not necessary to connect the Glitch Filter Component’s output terminal to an Interrupt Component. Depending on the application, you can connect it to other digital logic or to a DMA channel.

You can easily add support for multiple pins by changing the Component’s signal width parameter.

Although a switch debouncer is not a 100 percent effective glitch filter, a glitch filter can be an effective switch debouncer. However, glitch filters may be overkill for a basic switch debouncing application. The Glitch Filter Component uses approximately 20% more PSoC UDB PLD resources than the Debouncer Component.

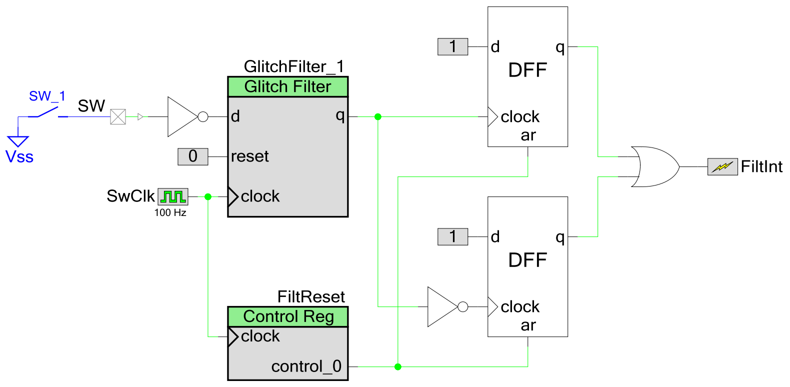
8 Summary

We can now come full circle back to the original design shown in Figure 5 on page 4, except that we add a Glitch Filter component between the pin and the DFFs, as Figure 17 shows. The design is no longer sensitive to switch bounce.

Similar to Figure 13 and Figure 16, we add an inverter to eliminate the initial false reset event.

The Glitch Filter clock also clocks the FiltReset Control Reg. This ensures a reliable reset when writing to the Control Reg.

Figure 17. Design from Figure 5 with a Glitch Filter Added



Switch debouncing and glitch filtering are an essential part of processing digital input signals. This application note has shown several ways to implement them, using different combinations of hardware and software.

There are many opportunities to optimize these designs for specific applications. For example:

- All of the designs capture both positive and negative edges. If you are interested in only one edge, you can simplify the design in hardware or software.
- To support multiple digital inputs, you can scale the design either by using multiple bits in a register or variable or by setting the signal width parameter of the Debouncer or GlitchFilter Component.

9 Design Projects

The projects attached to this application note are organized as shown in Table 1.

Table 1. Debounce Projects

Design Project Name	Description
A_DebounceNone	Demonstrates extra counts recorded in the absence of switch debouncing – how NOT to do the design.
B_DebounceSwPoll	Demonstrates switch debouncing by software sampling of an input pin at periodic intervals.
C_DebounceSwInt	Demonstrates switch debouncing using an interrupt driven by a low-frequency clock.
D_DebounceHwReg	Demonstrates switch debouncing by connecting the input pin to a status register with a low-frequency clock.
E_DebounceHw	Demonstration project for the Debouncer Component.
F_GlitchFilter	Demonstration project for the Glitch Filter Component.

10 Related Application Notes

- AN54181, Getting Started with PSoC 3

- AN79953, Getting Started with PSoC 4
- AN77759, Getting Started with PSoC 5LP
- AN81623, PSoC Digital Design Best Practices
- AN82250, PSoC Implementing Programmable Designs with Verilog
- AN82156, PSoC Designing PSoC Creator Components with UDB Datapaths
- AN77900, PSoC 3 and PSoC 5LP Power Reduction Techniques
- AN86233, PSoC 4 Power Reduction Techniques
- AN54460, PSoC Interrupts

About the Author

Name:	Mark Ainsworth
Title:	Applications Engineer Principal
Background:	Mark Ainsworth has a BS in Computer Engineering from Syracuse University and an MSEE from University of Washington, as well as many years experience designing and building embedded systems.

Document History

Document Title: AN60024 - Switch Debouncer and Glitch Filter with PSoC® 3, PSoC 4, and PSoC 5LP

Document Number: 001-60024

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2895472	FSU	03/18/2010	New application note
*A	2901010	FSU	03/31/2010	Attached associated project to the application note
*B	2991584	SRIH	07/22/2010	Fixed branding discrepancies
*C	3010675	XKJ	08/18/2010	Updated projects to Beta 5
*D	3068548	KLMZ	10/22/2010	Updated D_DebounceSwInt project and the corresponding content in the application note to use an edge triggered interrupt instead of a control register.
*E	3137066	MKEA	01/13/2011	Updated project files to work with PSoC Creator 1.0 FCS
*F	3370231	MKEA	09/14/2011	Expanded content and improved project based on customer feedback
*G	3393004	MKEA	10/12/2011	Project file renamed properly and attached
*H	3432788	MKEA	11/08/2011	Updated template and project file
*I	3441058	MKEA	12/16/2011	Simplified Debouncer component
*J	3538104	MKEA	02/29/2012	Added references for more information on PSoC Creator library projects and updated template
*K	3718792	MKEA	08/23/2012	Added Table of Contents.
*L	3819424	MKEA	11/22/2012	Updated for PSoC 5LP and PSoC Creator 2.1 SP1
*M	4088648	MKEA	08/08/2013	Updated for PSoC 4 and PSoC Creator 2.2 SP1. Projects modified to use PSoC Creator standard Debouncer and Glitch Filter Components.
*N	4258095	MKEA	01/22/2014	Corrected Code listings to match attached projects. Corrected some reference links. Miscellaneous minor edits.
*O	4364538	MKEA	04/28/2014	Updated for PSoC Creator 3.0. Added a section on UDBs. Minor edits and formatting changes.
*P	5726423	BENV	05/04/2017	Updated logo and copyright

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

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709