AN60024

Switch Debouncer and Glitch Filter with PSoC® 3 and PSoC 5

Author: Mark Ainsworth
Associated Project: Yes
Associated Part Family: All PSoC® 3 and PSoC 5 parts
Software Version: PSoC Creator™ 1.0 and higher
Related Application Notes: None

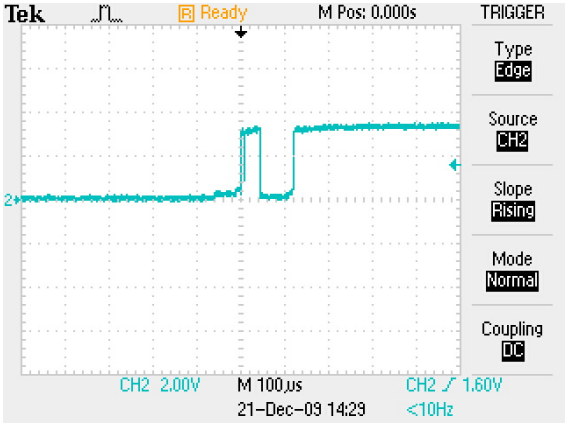
If you have a question, or need help with this application note, contact the author at mkea@cypress.com

Abstract

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

CapSense® and TrueTouch™ notwithstanding, many products still have mechanical buttons or switches. When a switch is pressed or released, its output can oscillate rapidly for a brief period, as shown in Figure 1. These oscillations can cause other parts of the system (such as the CPU) to falsely detect multiple press and release events, possibly leading to erratic and unexpected system behavior. Imagine for example what might happen if the switch was a 'Speed Up' button on a treadmill.

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



Filtering out these oscillations is known as switch debouncing. Switch debouncing can be done with a simple RC filter on the switch, but why pay for extra components when PSoC 3 or PSoC 5 devices can do the job easily and in many ways, using minimal device resources? And, an RC filter is fixed, while PSoC 3 or PSoC 5 devices can easily and dynamically adapt the characteristics of a switch debouncer to different switch or button characteristics.

This application note shows several ways to do switch debouncing in PSoC 3 or PSoC 5, using hardware, software, or both. Glitch filtering, which is very similar to switch debouncing, is also demonstrated.

Several PSoC Creator design projects are attached to this application note to illustrate the different methods. Some of the designs are included as components in a PSoC Creator library project, for easy reuse. For more information on how to create and use library projects, see PSoC Creator help articles "Library Component Project" and "Basic Hierarchical Design Tutorial".

Pin Setup

Before we get into switch debouncing, let us briefly look at how best to connect a switch or button to a PSoC 3 or PSoC 5 pin. A button, for example those used on the CY8CKIT-001 PSoC Development Kit board, typically shorts to ground when pressed and opens when released. Therefore, a pull-up resistor to a voltage level is usually required at the pin. Using PSoC Creator, a Pin component can be configured to provide that pull-up, as shown in Figure 2, Figure 3, and Figure 4 on page 2.

Figure 2. Block Diagram of a Switch Shorting an Input Pin to Ground

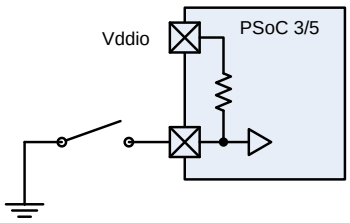
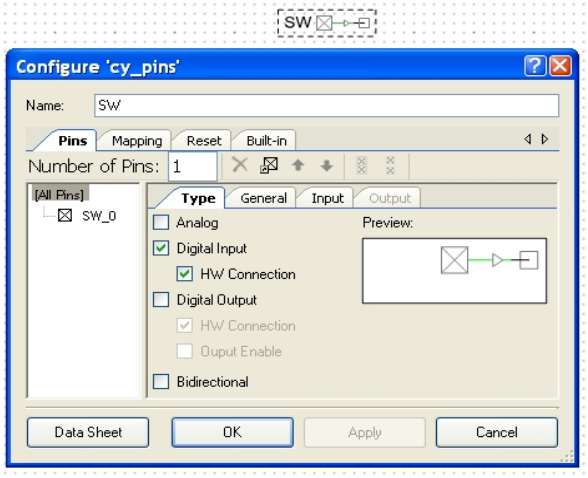
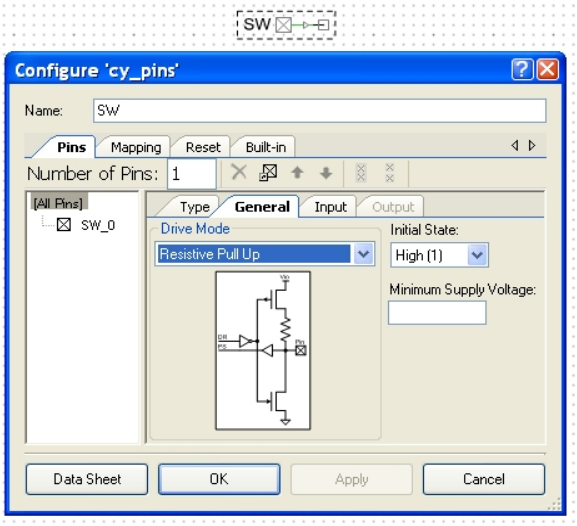


Figure 3. Type Configuration for an Input Pin Component



For this setup, the Digital Input box must be checked; the HW Connection box may or may not be checked depending on the circuit used. To enable the pin's internal pull-up resistor, go to the General tab and set the configuration as shown in Figure 4.

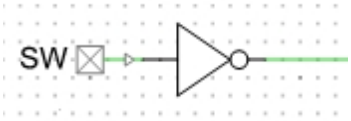
Figure 4. Configuration for an Input Pin Component



Set the Drive Mode to Resistive Pull Up, and the Initial State to High (1), to enable the pull-up resistor. The drive mode and state value are usually left unchanged after initialization.

Reading the pin value gives a logic '0' when the switch is pressed, and a logic '1' when released, which in most cases is acceptable. If you want the polarity reversed, either do the inversion in software, or in hardware just add an inverter as shown in Figure 5.

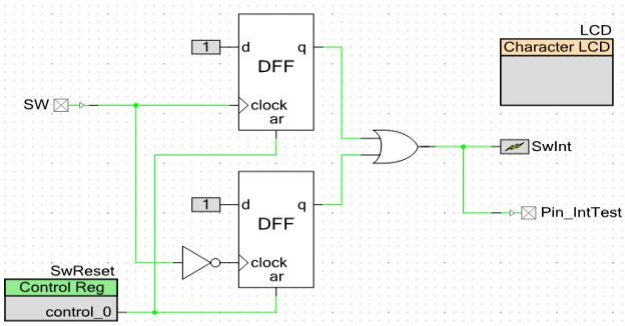
Figure 5. Schematic of an Input Pin with a Hardware Connection to an Inverter



A Switch Bounce Sensitive Design

The design shown in Figure 6 demonstrates a poor design—it is overly sensitive to switch bounce.

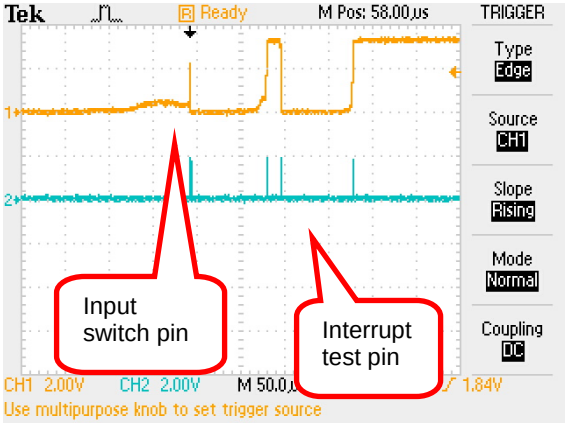
Figure 6. Schematic Showing a Design that is Overly Sensitive to Switch Bounce



The positive and negative edges from the input pin are captured by two digital flip flops (DFF) and trigger an interrupt. Code 1 shows how to handle the interrupt—the interrupt handler code simply resets the DFFs and increments a global 'count' variable. The background loop in main() continually displays the value of 'count' on the LCD display.

The display should increment with each button press and button release. However if you run the attached project 'A_DebounceNone' you will see that the display sometimes increments several times on a single button press or release. The reason becomes apparent if you connect a scope to both the input switch pin and the interrupt test pin; see Figure 7:

Figure 7. Scope Shot Showing Switch Bounce and Interrupt Response



The system is running at high speed; the project is configured to run at a top speed bus clock. This allows each interrupt to be processed in a very short time relative to the transitions rate, and thus 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 resolve this problem, you must use a switch debouncing technique. The remainder of this application note shows various ways to do so.

Note that in PSoC 3 and PSoC 5, the input 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 6 on page 2.

Using the PICU, pin transitions can wake up the PSoC 3 from a sleep or hibernate mode. 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 3 sleep and hibernate modes, see AN66083.

Code 1. Interrupt-based Response to Input Pin Transitions

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

CY_ISR(SwInt_ISR)
{
    /* Clear interrupt source. PSoC 3 interrupt handlers should not call
     * functions, so access the register directly.
     */
    SwReset_Control = 1;
    SwReset_Control = 0;

    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, 8); /* row, column */
        LCD_PrintHexUint8(temp);
    }
} /* end of main() */
```

Switch Debouncing by Sampling

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

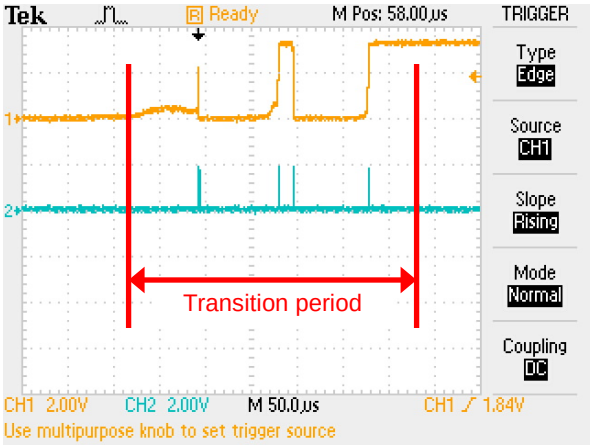
The sample period is set by the anticipated transition time of the signal, that is, the time it takes for the signal to stabilize at the new state when the switch is opened or closed. During the transition time the state of the signal is essentially unknown—at any time during that period it could be either 0 or 1. You should not sample more than once during that period or you may detect an extra transition. So the maximum sample rate is just the inverse of the maximum transition period, and do not forget that you may need to check both the low-to-high and high-to-low transition periods.

Note that you might still sample a pulse during the transition period and thus detect it early as the actual transition; see the Glitch Filtering section on page 8 for more information.

The example in Figure 8 shows that the transition from pressed to released takes ~300 µs, so in that case the maximum sampling rate should be 1 / 300 µs, or 3.3 k samples per second (sps).

In practice, you should set the sample rate to be much lower than the expected transition period, 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.

Figure 8. Scope Shot Showing Switch Transition Time



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, 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 just monitors one pin (in bit 0), so the members of the 'switches' array are compared in their entirety, assuming 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 3 or PSoC 5 you can monitor a pin in software as well as in hardware, at the same time. The attached project 'B_DebounceSwPoll' contains some code added to project 'A_DebounceNone'; to show both raw (unfiltered) and filtered counts.

Code 2. Periodic Sampling of an Input Pin

```
void main()
{
    uint8 filtered_count = 0; /* # of filtered transitions of input pin 'SW' */

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

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

    for(;;) /* do forever */
    {
        /* 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(0, 14); /* row, column */
        LCD_PrintHexUint8(filtered_count);
    }
} /* end of main() */
```

In code that must do many tasks it may be difficult to poll a pin at regular time intervals. In that case an alternative is to sample by using a periodic interrupt. The easiest way to do this is to connect a clock component directly to a sample interrupt component, as Figure 9 shows. 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 10.

Figure 9. Components for Periodic Interrupts

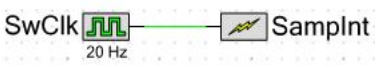


Figure 10. SampInt Configuration

In this design, as Code 3 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.

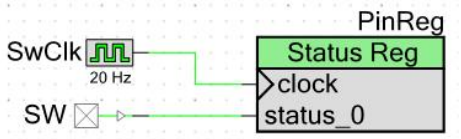
Code 3. Interrupt-based Sampling of an Input Pin

```
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 */

    /* Read the state of the 'SW' pin without calling a function */
    temp = ((SW_PS & SW_MASK) >> SW_SHIFT);

    /* 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
    { /* Set semaphore if a 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() */
```

Switch Debouncing Using Hardware

PSoC 3 or PSoC 5 makes it easy to do switch debouncing in hardware instead of software, which can reduce usage of the CPU. The easiest way to do this is to poll with a clock and a status register, as Figure 11 shows.

Figure 11. Schematic of a Design with Register-based Switch Debouncing



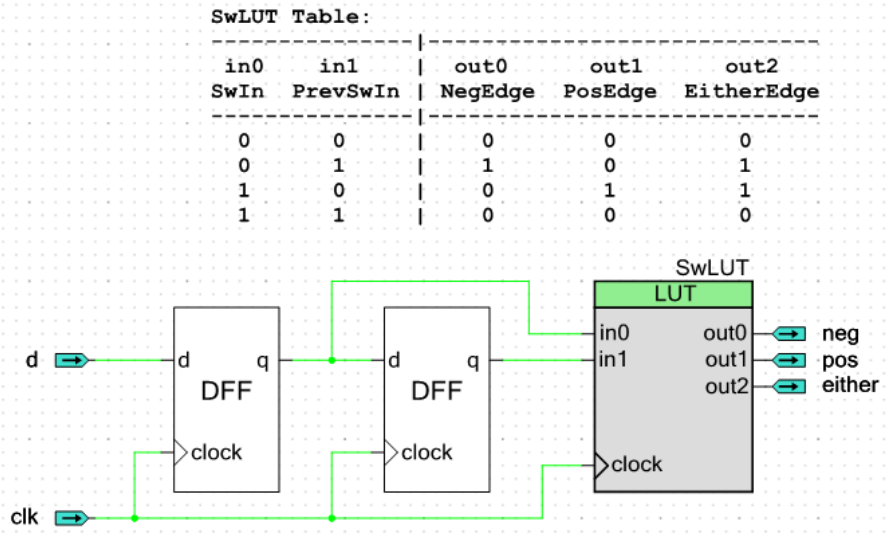
The code for this design is the same as in Code 2 except that you read from PinReg_Status instead of SW_Read(), and the CyDelay() function is removed. See attached project 'D_DebounceHwReg'.

You can further de-burden the CPU by moving both the input pin polling and comparison functions into hardware, as Figure 12 shows. The hardware design detects and reports transitions on the input pin.

In the design, the clk input is used to sample the switch input 'd'. As noted previously, a clock frequency of 10 to 200 samples per second is usually appropriate.

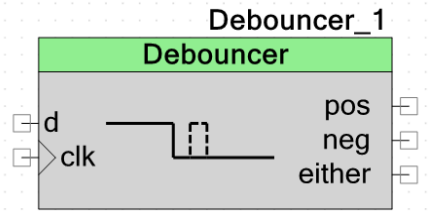
The first DFF is used to sample the switch input. The second DFF stores the previous sample, and the lookup table (LUT) component does edge detection by comparing the two samples. Note that the delay from an input transition to an edge detect at one of the LUT outputs is one period of clk.

Figure 12. Debouncer Component Schematic



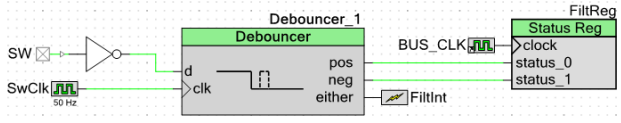
For easy reuse, this design is encapsulated as a component in a PSoC Creator library. The component symbol is shown in Figure 13.

Figure 13. Debouncer Component Symbol



A design using the component is shown in Figure 14. As shown previously in Figure 10 on page 5, make sure that the interrupt component is configured for RISING_EDGE mode.

Figure 14. Design Using a Debouncer Component



Note that 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 release event at initialization. An inverter on the input pin eliminates this problem.

This design also includes a status register 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.

It is not necessary to connect an output terminal to an Interrupt component. Depending on the application, you can connect it to other digital logic or to a DMA channel. Also, multiple instances of the component can be easily added to support multiple pins.

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_Status;

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

Glitch Filtering

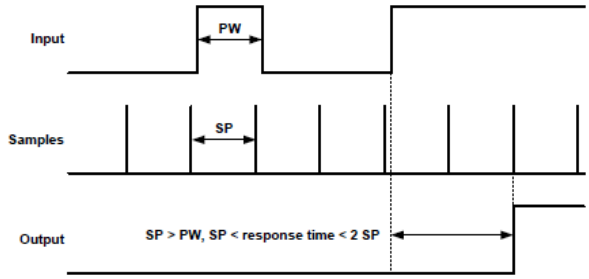
Glitch filtering is similar to switch debouncing but supports a slightly different application. In switch debouncing, you filter out all edges except the single one that you want. In glitch filtering, you remove unwanted pulses from a signal.

Note that glitches are not necessarily associated with switches; they can occur on lines carrying signals from other 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 is unchanged from its current value.

If N = 1 then the time between two successive samples must be greater than the maximum pulse width that can be filtered (MaxPW). It then follows that, since it will take at least two samples for the output to change, the response time, or filter delay, is between 1 and 2 sample periods. See Figure 15.

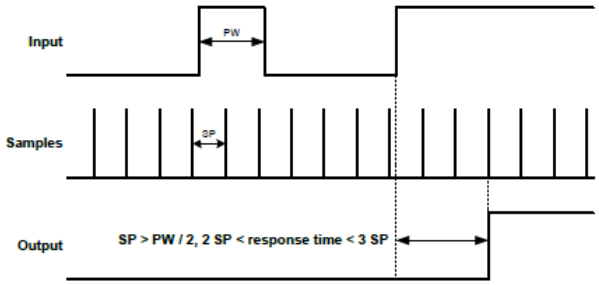
Figure 15. Glitch Filter Performance for N = 1



The response time can also be expressed in terms of the MaxPW. In the case for N = 1, since 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 then the response time is between 2 and 3 sample periods. But the sample period can be shorter—the MaxPW must be less than 2 sample periods—and then the response time can be expressed as 1 to 1.5 times the MaxPW. See Figure 16.

Figure 16. 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

$$N \cdot SP < response\ \ time < (N+1)\ SP$$

$$SP > \frac{MaxPW}{N+1}$$

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

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 resources to implement.

Figure 17 shows a schematic for a hardware-based design where N = 2. For easy reuse, the design is encapsulated as a component in a PSoC Creator library, and as a Concept component in PSoC Creator 2.0 or higher. The component symbol is shown in Figure 18.

Figure 17. Glitch Filter Component Schematic, Where N = 2

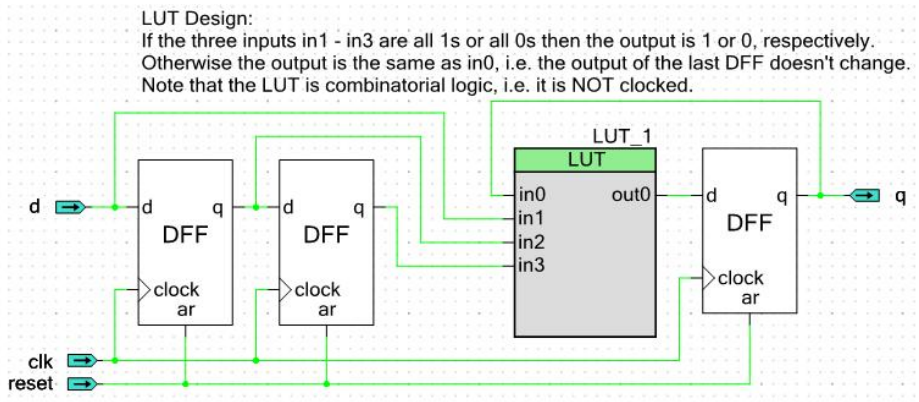
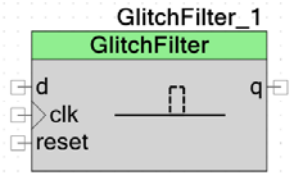


Figure 18. Glitch Filter Component Symbol

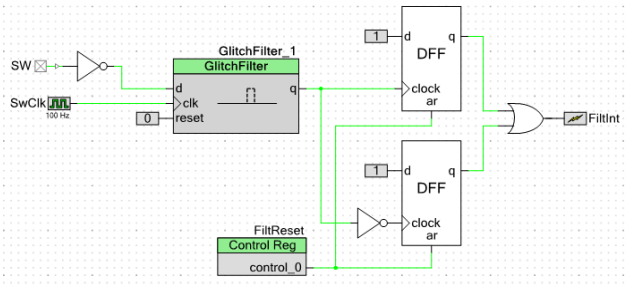


Although a switch debouncer is not a 100 percent effective glitch filter, a glitch filter can be an effective switch debouncer. However, glitch filters may be overkill for a basic switch debouncing application, in terms of resource use.

Summary

We can now come full circle back to the original design shown in Figure 6 on page 2, except that we add a Glitch Filter component between the pin and the DFFs, as shown in Figure 19. The design is now no longer sensitive to switch bounce.

Figure 19. Design from Figure 6 with a Glitch Filter Added



Similar to Figure 14, an inverter is added to eliminate the initial false reset event. Because the DFFs in the GlitchFilter component are reset to '0' at device reset, the reset input is not always needed and in this case is just connected to a Logic Low '0' component.

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 various combinations of hardware and software. The hardware-based designs are included as components in PSoC Creator library projects. For more information on how to create and use library projects, see PSoC Creator help articles "Library Component Project" and "Basic Hierarchical Design Tutorial".

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 only interested in one edge, then a design can be simplified in hardware or software. Also, to support multiple digital inputs you can scale the design either by using multiple bits in a register or variable or by using multiple instances of the Debouncer or GlitchFilter component.

Design Projects

The projects attached to this application note are organized as shown in Table 1 and Table 2.

Table 1. PSoC Creator 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	Test / demonstration project for the Debouncer component. See Table 2.
F_GlitchFilter	Test / demonstration project for the GlitchFilter component. See Table 2.

Table 2. Contents of the DebounceLib Folder

Library Project Name	Description
Debouncer	Debouncer hardware design
GlitchFilter	Glitch filter hardware design. Also available as a Concept Component in PSoC Creator 2.0.

About the Author

Name:	Mark Ainsworth
Title:	Applications Engineer Principal
Background:	Mark Ainsworth has a BS in Computer Engineering from Syracuse University and a MSEE from University of Washington.
Contact:	mkea@cypress.com

Document History

Document Title: Switch Debouncer and Glitch Filter with PSoC® 3 and PSoC 5 – AN60024

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

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
Optical Navigation Sensors	cypress.com/go/ons
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 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com