Lab 6: Flip-Flops and Glitch Filters

Objective

Demonstrate knowledge of flip-flops and glitch filters. Additionally, improve the stopwatch from Lab 2 by converting polling for events to responding to interrupts for events.

Note

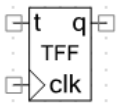
This lab is INCOMPLETE. It was originally part of Lab 3, but it was determined that it was too long. Modifying the stopwatch project to use interrupts instead of polling took too long to accomplish, and the purpose of using the glitch filter in the stopwatch may have been confusing. It is probably better to remove the stopwatch from Lab 2 and create a lab that is only about creating a stopwatch and doesn't introduce any new components, which allows the students to apply what they have learned so far to a larger application. This proposed stopwatch-only lab should be made such that the procedure isn't step-by-step so the student can figure out what to do based on information and hints given. Having such a lab would also prepare them better for the final project.

Background

B1: Flip-Flops

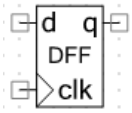
A flip-flop is a component that is used to store one bit of information. In PSoC Creator, a flip-flop component can be configured to hold multiple bits using the ArrayWidth parameter, but behind the scenes it creates one flip-flop per bit. The way a flip-flop stores a bit depends on the type of flip-flop used. All flip-flops do not have software interfaces.

Toggle Flip Flop



A toggle flip-flop stores a bit that toggles between low and high states depending on the input t, synchronized to the clock. When t is high at the rising edge of the clock, the output q is toggled. When the input is low, q stays the same. There are no configuration options other than ArrayWidth.

D Flip Flop



This flip-flop sets the output q to the input d at the rising edge of the clock. Until the rising edge of the clock, q does not change. One use for the d flip-flop is to convert an asynchronous signal to a synchronous signal sampled by the clock. This is similar to a debouncer, except there is no edge detection.

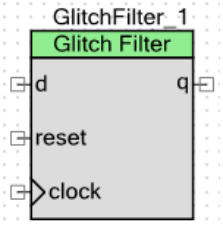
More options for the D Flip Flop and other flip-flops are detailed in the appendix.

You may now start Procedure Part A. Return here for background for Part B.

B2: Glitch Filter Component

Glitch filters are similar to debouncers, as they are used to remove oscillations in the input signal. However, they are more robust than the debouncer, as it takes multiple samples or counts how many times it has sampled a different value before making a transition. While a debouncer has the chance that it took a sample during an oscillation, the glitch filter will take at least two samples on its clock before changing its output. Additionally, the glitch filter has an option to allow one logic value to bypass the filter. For example, the depress action in a stopwatch button filter can be passed on immediately (when sampled) while the release is filtered so oscillations in the release do not cause another depress to be detected.

A glitch filter uses up more resources than a debouncer, so it should only be used when a debouncer is insufficient. Additionally, increasing the number of samples before changing state increases the number of resources used, up to 9. Setting the number between 1 and 8 uses an implementation that increases resources as the number increases and setting the number from 9 to 256 uses an implementation that uses the same amount of resources regardless of the number, but more resources that when it's set to 8.



Here are the inputs and outputs for the component:

- Inputs
  - d: the input that is sampled
  - reset: resets the filter when it is logic high for at least one clock cycle. May remain disconnected.
  - clock: the clock used to sample the input
- Outputs
  - q: the filtered output value

These are the configuration options:

- Signal width (bits): if multiple signals need to be filtered, use this to increase how many inputs and outputs there are. This utilizes buses (multiple signals per wire) for the inputs and outputs.

- Glitch length (samples): determines how many times a signal has to be sampled (1-256) in a new state before transitioning to it. If the new state is sampled this many times AND remains in that state on the following clock rising edge, the output changes to the new state on the following rising edge. This also shows the response time for the filter based on the current input clock.
- Bypass filter
    - None: both logic levels are filtered the same way
    - Logic zero: if the output is 1 and the input is sampled at 0, the output changes to 0 in that same clock cycle rather than going through the filtering process.
    - Logic one: if the output is 0 and the input is sampled at 1, the output changes to 1 in that same clock cycle rather than going through the filtering process.

This component has no programming interface; it exists purely in hardware.

Pushbuttons do not have to be inverted before being used with the glitch filter, but it is usually best practice to do so anyway since some components, such as interrupts, only look for rising edges.

Procedure Part A

Outcome

In this part of the lab, the stopwatch in Lab 2 will be revisited and modified to work using interrupts rather than polling. This will also fix the issue where the software thinks the stopwatch has started when the user holds down the start button while pushing the stop button.

In addition to using interrupts on the pushbuttons, another interrupt will be used along with a new timer to signal the CPU to refresh the display approximately 60 times per second while the stopwatch is running, instead of refreshing as fast as possible.

Part 1: Designing the System

1. Close all open tabs and collapse the Lab3 project in the workspace explorer.
2. Right click on the Lab2 project and select Copy (note: if it can't be copied and it's the active project, right click a different project and make that one active).
3. Right click on the root of the workspace (at the top of the tree) and select Paste.
4. The pasted project will be named "Lab2_Copy_01." Rename it to "Lab6" by right clicking on it and selecting Rename.
5. Open TopDesign.cysch
6. Delete StopwatchStart and the clock it's connected to.

Because the trigger input of Stopwatch is sampled on Stopwatch_clock, the interrupt for starting the stopwatch needs to also sample SW2_StopwatchStart at that frequency. Otherwise, very brief signals (such as oscillations) may be interpreted as starting the stopwatch when they actually don't. Ideally, the timer would generate an interrupt when it is triggered, but that is not an option in this component.

7. To solve the above problem, add a Digital>Logic>D Flip Flop to the schematic. Connect the inverted SW2_StopwatchStart to the d input and Stopwatch_clock to the clock input. This flip-flop resamples the input to the clock in the same way Stopwatch does with its inputs.
8. Add an interrupt component, name it StopwatchStartISR, configure it to be RISING_EDGE, and connect it to the q output of the flip-flop.
9. Reconfigure the Stopwatch timer to generate an interrupt signal on capture by checking the On Capture box next to Interrupts and keeping the number at 1 in the configuration options.
10. Add an interrupt component, name it StopwatchStopISR, and connect it to the interrupt output of Stopwatch (a capture signals when the stopwatch stops).
11. Create an 8-bit fixed function continuous timer named DisplayRefresh with a period of 16ms that's configured to generate an interrupt signal on TC (which is when the period expires). SecondTimer_clock may be reused for this purpose.
12. Add an interrupt component, name it DisplayRefreshISR, and connect it to the interrupt output of DisplayRefresh. This ISR will signal the CPU to update the display when the stopwatch is running. The 16ms period on the attached clock will cause this interrupt to run roughly 60 times per second.
13. Generate the application.

Since there are multiple interrupts, it may be desirable to change their priorities in the DWR file under the Interrupts tab. However, in this lab, all three interrupts will be using the same data resources, creating a potential race condition. If all interrupts remain at the same priority (which is default), interrupts will run to completion before allowing another interrupt to run, negating the race condition. Note that this does not eliminate possible race conditions between the main function and the interrupts, but the main function will not do anything after initialization in this application.
Additionally, there is no need to prioritize the start and stop interrupts over the display refresh interrupt. The Stopwatch timer handles starting and capturing at the moment those buttons are pressed, so a short delay before handling the interrupt is acceptable.

Part 2: Programming the Firmware

Outcome

In Lab 2, all of the code was placed in the main function. In this lab, all but the initialization code and global variables will be moved to the ISRs, which allow for quicker reaction times. The main loop will simply be doing nothing in an infinite loop.

Custom Header File

Since there will be a global variable used throughout multiple files, a header file that holds it and any constant values (such as the Stopwatch clock frequency) should be used. Under Lab3_partC in the workspace explorer, there is a folder named Header Files. Right click this folder and select Add>New Item…, choose Header File, name it globals.h, and click Ok.

The template for header files does not include the necessary boilerplate to protect against multiple includes of the same header file, so that needs to be added. Also, include <project.h> in this header, otherwise data types such as uint8 will be undefined. Place all references to global variables (using the

extern keyword) and any constants (for example, #define FALSE 0) in this file. If sprintf will be used, also include “stdio.h”. See the code below for what the header should look like.

```
#ifndef __GLOBALS_H__
#define __GLOBALS_H__

#include <project.h>
#include "stdio.h"

#define TRUE  1
#define FALSE 0
#define STOPWATCH_FREQ 10000

extern uint8 started_b;

#endif
```

Main

In main.c, declare the following global variable:

uint8 started_b;

This is a Boolean value that indicates whether the stopwatch is running. Zero means it is not running and non-zero means it is. Feel free to use the TRUE and FALSE definitions in globals for this.

Include <globals.h> in case anything from there needs to be used in main. Additionally, if the asm declaration to include printing of floating points needs to be used, ensure it remains in main.c.

In the main function, call all of the start functions on the ISRs, timers, and Display. Initialize started_b to 0, or FALSE if that’s defined in globals.h. Print the string “Stopped” on the first line of the display. Because of the power-on reset nature of the pushbutton pins, StopwatchStartISR and Stopwatch may have already been triggered, so clear the interrupt with its ClearPending function and reset the timer by writing 1 to StopwatchReset. Finally, enable interrupts with CyGlobalIntEnable.

Comment out all of the code in the infinite for loop. All of the code will be moved and/or reimplemented in the ISRs. Ensure the infinite for loop remains, since returning from main will stop the program.

StopwatchStartISR

Open StopwatchStartISR.c. Include <globals.h> in the editable (between the #START and #END comments) section at the top of this file.

In the ISR (StopwatchStartISR_Interrupt, between the #START and #END comments) check to see if started_b is zero. If it is, set started_b to non-zero, clear the display, and print “Started” to the top line of the display.

Since this ISR is connected directly to the inverted pushbutton, there is nothing to clear in the hardware to allow for future start interrupts.

While it is best practice to make ISRs run as quickly as possible so other ISRs can be handled, the Stopwatch timer runs relatively slow compared to the CPU (10 kHz vs. 24 MHz), so there is no rush. Additionally, since the timer also handles capturing the moment the stop button is pressed, there is no rush to handle the stop button ISR either. In the case where the speed of the ISR matters, it is best practice to unload moderate and heavy work such as printing to the display to the main function. This was demonstrated in Part A.

DisplayRefreshISR

Open DisplayRefreshISR.c. Include <globals.h> in the editable section at the top of this file.

In the ISR check to see if started_b is non-zero. If it is, read the counter of the timer and print the number of seconds to the second line of the display (reuse code from Lab 2).

Since this ISR is triggered by a timer, call DisplayRefresh_ReadStatusRegister to clear its interrupt output.

StopwatchStopISR

Open StopwatchStopISR.c. Include <globals.h> in the editable section at the top of this file.

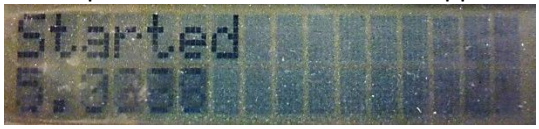
In the ISR check to see if started_b is non-zero. If it is, print "Stopped" to the top line of the display, read the timer's capture and print the number of seconds to the second line of the display, and set started_b to zero.

Since this ISR is triggered by a timer, call Stopwatch_ReadStatusRegister to clear its interrupt output. Additionally, reset the timer by writing 1 to StopwatchReset.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Test the program to make sure it works:
    - Disconnect and reconnect the USB cable from the computer to hard reset the devlopment kit.
    - Starting out, the display should only say Stopped, or equivalent. LED4 should still be blinking once per second.

    - Press SW2. The display should say Started, or equivalent, and should be continually updated with the elapsed time. Ensure that time appears to start at zero.



    - Press SW3. The display should say Stopped and show a time close to what was displayed right before pressing SW3.

- Press SW2. The display should say Started and should continually show the elapsed time, starting back over at zero. Ensure there are no excess digits left over from the last run; if there are, the display wasn't cleared before starting.

3. Note that while the stopwatch is running, pressing SW2 has no effect. Pressing SW3 should stop the stopwatch as usual. But unlike in lab 2, if SW2 is held down while pressing SW3, the software will not think the timer is running since the interrupt is filtering the SW2 input on its rising edge, sampled at 10 kHz. Releasing SW2 while the stopwatch is stopped may trigger it to start, however, since it is not debounced. SW3 is also affected by this issue. This problem can be fixed using glitch filters, which is described in the next section.

Procedure Part B

Outcome

In this part of the lab, the issue where releasing a pushbutton may trigger the timer will be fixed using glitch filters. Unlike a debouncer, the glitch filter can be configured to let one transition bypass the filtering. It is also more robust than the debouncer, as it observes multiple samples of the input before changing the output, rather than just sampling the input and producing that as the output.

Part 1: Designing the System

1. Reopen TopDesign.cysch. The final schematic for this part is depicted in Figure 1.
2. Delete the wires leading from the NOT gates attached to the pushbutton inputs to the Stopwatch timer and (in SW2's case) the D flip-flop connected to StopwatchStartISR.
3. Add a Digital>Utility>Glitch Filter to the schematic and name it StopwatchStartFilter.
4. Connect Stopwatch_clock to the clock input and the inverted SW2_StopwatchStart to the d input.
5. Configure StopwatchStartFilter to have 200 samples (20 ms) and set Bypass filter to Logic one. Setting Bypass filter to Logic one will allow the depress action to take effect on the next rising edge of the clock, rather than after 200 samples (as long as the pushbutton is still pressed). Having logic zero filtered means the signal won't go low until at least 20 ms after the pushbutton is released, removing any oscillations caused by the release, which eliminates the issue described at the end of the last part of the lab.

Using the bypass filter option in the glitch filter is insufficient to start the timer immediately after the pushbutton is pressed. Since the bypass is synchronous, the change in the output only happens at the rising edge of the clock. Since the Stopwatch timer checks its inputs at the rising edge of the clock, any inputs that change right after that time will not be read until the next rising edge. Running the glitch filter on a faster clock wouldn't completely fix the problem either, since the faster clock could still line up with Stopwatch_clock, or a change an input could happen after the faster clock samples, then Stopwatch_clock samples before the faster clock samples again, causing a miss in

triggering the timer. To ensure that the timer is triggered when it should, there also needs to be an asynchronous logic one bypass to the glitch filter.

6. Add a Digital>Logic>Or gate to the schematic. Connect the q output of StopwatchStartFilter and the inverted SW2_StopwatchStart to the inputs of the OR gate. This way, when the button is pressed, the output of the OR gate immediately goes high. When the glitch filter samples the button, it also goes high and will continue to be so until the pushbutton is released for 20ms.
7. Connect the output of the OR gate to the trigger input of Stopwatch and the d input of the D flip-flop. If the glitch filter was used directly without the OR gate, its output only goes high at the rising edge of the clock, at the same time the timer samples its trigger input. Therefore, the response to the button press would be delayed by one cycle.
8. Repeat steps 3-8, except name the glitch filter StopwatchStopFilter, use SW3_StopwatchStop, and connect the OR gate to the capture input of Stopwatch (and not the flip-flop).

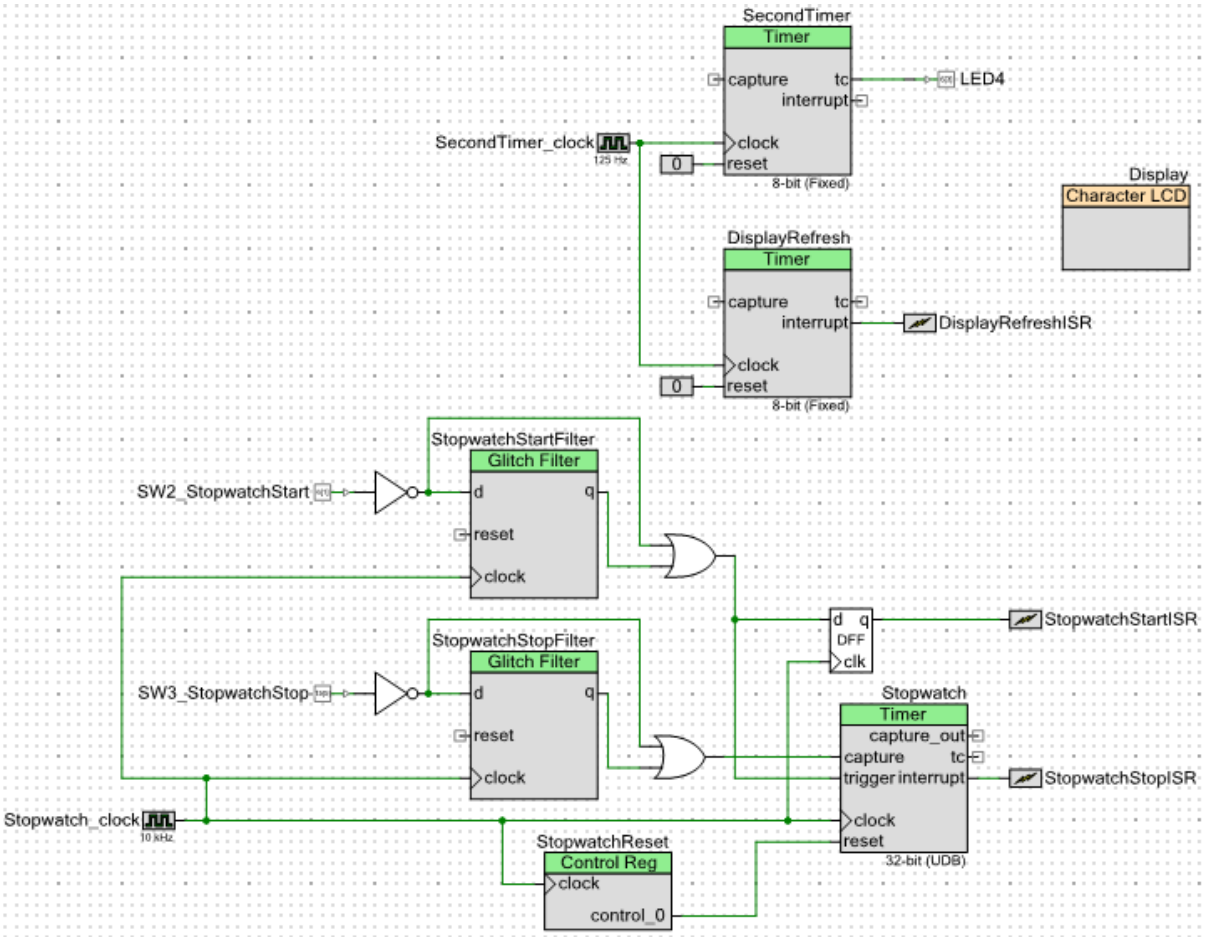


Figure 1: Schematic for Procedure Part D

Part 2: Programming the Firmware

There is nothing to change in the programming, as the glitch filter has no programming interface and does not need to be initialized.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Ensure the glitch filters are working as anticipated.
    - Press and hold SW3 and press and release SW2. The stopwatch should start.
    - Release SW3. The stopwatch should continue to tick.
    - Press and hold SW2 and press and release SW3. The stopwatch should stop.
    - Release SW2. The stopwatch should remain stopped.
3. Temporarily reduce the frequency of Stopwatch_clock to see the effect of the glitch filters:
    - Depressing a pushbutton the first time should have an immediate effect. Releasing and pressing the pushbutton again should not, unless the time between presses exceeds the time specified next to Glitch length (samples) in the configuration of the glitch filter.
    - The delay between releasing the pushbutton and having that action take effect removes any oscillations caused by the release of the pushbutton, and thus eliminates triggering the Stopwatch timer or interrupts when they shouldn't.

Appendix

A1: Other Flip-Flop Types

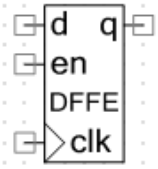
Additional Options for the D Flip Flop

The D flip-flop has additional input terminals that are hidden by default and shown by changing the configuration. Only one of these is visible at any time, except for sr and sp, which can be shown at the same time.

- ar – Asynchronous Reset (PresetOrReset set to Asynchronous Reset): while this input is high, the value of the flip-flop is forced to be low.
- ap – Asynchronous Preset (PresetOrReset set to Asynchronous Preset): same as ar, except the output is forced to be high.
- sr – Synchronous Reset (PresetOrReset set to Synchronous Reset or Synchronous Preset & Reset): when this input is high on the rising edge of the clock, the value of the flip-flop is set to low, regardless of d.
- sp - Synchronous Preset (PresetOrReset set to Synchronous Preset or Synchronous Preset & Reset): same as sr except the value is set to high.

If both reset and preset are triggered, the value is set to low.

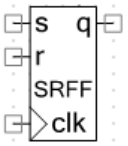
D Flip Flop with Enable



This is the same as the D Flip Flop, except it has an enable input. When enable is low, d does not affect q. When enable is high, q becomes d at the rising edge of the clock, just like with a D Flip Flop.

This does not have any of the hidden reset or preset pins, unlike the other D Flip Flop, and thus has no configuration options other than ArrayWidth.

SR Flip Flop



This flip-flop sets q at the rising edge of the clock to high if s is high, to low if r is low, or makes no change if both s and r are low. If both s and r are high, r takes precedence and q is set to low. The two inputs stand for set and reset. There are no configuration options other than ArrayWidth.