INTRODUCTION

The physical terminals of a switch often bounce when opened or closed. Because of this, the voltage waveform of any electrical connection made to a bouncing switch often contains some high frequency components. Under these conditions, such a high frequency component is also generally referred to as a *bounce*. Additionally, during a period of time in which there exist bounces within a voltage waveform, the waveform is (unsurprisingly) said to be *bouncing*. An example of some bouncing in a voltage waveform is shown in Figure 1, where the diagram is meant to be representing the voltage waveform of a digital switch circuit created with a pull-down resistor.

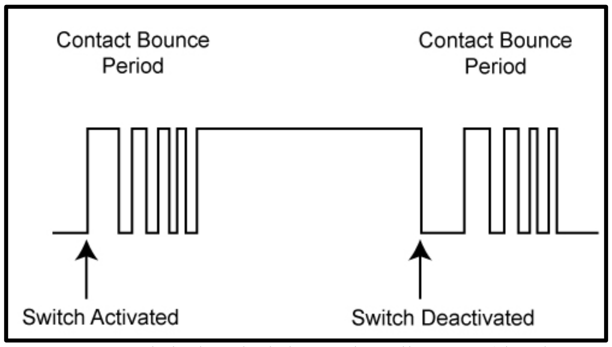

Figure 1: Digital switch bouncing diagram (thanks to Epec's Blog)

When software is written to depend on the digital value of a switch, switch bouncing can cause unintended and indeterminate results. For example, if the switch depicted in Figure 1 was utilized to increment some register value whenever the switch was determined to be closed, the bouncing shown implies that the relevant register value, meant to only be incremented once, could be incremented anywhere from one to nine times! Thus, it is often desirable to remove or ignore any bounces within a voltage waveform caused by a bouncing switch. In general, it is said that to do so is to *debounce the switch*, or, to perform *switch debouncing*.

Both hardware-based and software-based solutions exist for switch debouncing. Although designing hardware circuitry to debounce a switch is relatively simple and cheap, it can be costly in certain circumstances. In comparison, debouncing a switch through the use of software is almost always free and sufficient.

NOTE: In this document, the term *software* is meant to represent any form of program code, including machine and assembly code. Thus, it follows that there is also an implicit assumption that some computer architecture capable of supporting software is available to implement any relevant switch debouncing software designs.

To effectively debounce a switch through software, it is essentially only necessary to perform the following:

1. Outside of software, i.e., in the physical world, determine through measurements an estimate of the upper-bound for the duration of switch bouncing. The physical stress that the switch is put under, through varying forces applied to the switch, as well as the two separate cases of the switch being opened or closed, should all be considered when making an estimate.
2. Within software, whenever it is determined that some relevant switch is in its active state, instead of immediately performing the operation(s) meant to occur upon the switch being in this state, create some time delay, otherwise known as a *debounce delay*, greater than the estimated upper-bound for the duration of switch bouncing. With the assumption that the debounce delay is greater than this upper-bound, it can be assumed that all switch bouncing has ceased following such a delay. At such a time after an appropriate delay has completely elapsed, determine the state of the switch; if the switch is still in its active state, then it should be able to be safely assumed that the relevant operation(s) will be performed only once for some switch press (i.e., the intended operation[s] will not be performed due to the switch bouncing).

It should be straightforward for one to convince themselves of the above rationale: to debounce some switch, determine an estimate of the maximum amount of time that the switch could bounce, and only perform some operation(s) upon on the detection of the switch being in its active state if, after a delay greater than that of the relevant estimate, the switch is still in its active state.

NOTE: An estimate of the upper-bound for the duration of bouncing will likely be heuristic; only many measurements or careful analysis would allow for a very high probability that the switch will always cease to bounce after such an estimated amount of time. However, from empirical evidence involving common switch components, between five and twenty milliseconds is often a sufficient upper-bound estimate. (This range of time should almost assuredly be sufficient when debouncing any switches utilized for this course.)

While the general procedure for debouncing a switch through software has been made clear, an exact implementation has not been described. For the remainder of this document, three generalized implementation strategies that should be viable for most modern computer architectures are presented. Each strategy has benefits and drawbacks. The first technique, debouncing with a *software delay*, could be appropriate for simple applications, though should generally be avoided due to a strong restriction on program flow. The second, using a timer/counter (TC) system along with a relevant timer/counter hardware flag (where a hardware flag is defined here to be a signal that represents the occurrence of some predefined event), could also be appropriate for simple applications, though, like the first strategy, should generally be avoided due to a clear restriction on program flow. The third and final strategy described in this document, using a timer/counter along with a timer/counter interrupt, is by far the most desirable, since it has little to no restriction on program flow; however, this strategy is also generally the most complex to implement.

USING A SOFTWARE DELAY

The first strategy described here involves tasking the computer processor with a calculated number of (meaningless) instructions, otherwise known as a *software delay*, to keep the processor "busy" for a duration of time at least equivalent to that of the expected switch bouncing.

For example, suppose that it is desired that some switch be debounced through software. Further, suppose that all bouncing for the relevant switch should cease following a one-millisecond debounce delay. Then, to debounce the switch, one could simply create a software delay of one millisecond. After such a delay, the program would then check the appropriate pin level and perform any necessary function(s).

Overall, although software delays have the potential to be very precise, they prevent a microprocessor from executing other instructions, and ultimately cause CPU time to be wasted. Software delays are also extremely non-modular, as they cannot easily be used at other processor clock speeds, and if designed in an assembly language, cannot be directly ported to most other processors.

NOTE: Although the above is sometimes plausible when continually polling (i.e., continually reading) a switch outside of an interrupt service routine, software delays (or really any delay for that matter) should *almost never* be implemented within an **i**nterrupt **s**ervice **r**outine (**ISR**). Designing a program to explicitly delay within an ISR is typically a ghastly practice because this prevents a microprocessor from being able to service other interrupts during the delay, assuming that ISRs have not been allowed to nest/preempt each other, which is also normally discouraged. In general, interrupt service routines should be as short as possible.

For the following two debouncing strategies, it is assumed that hardware timer/counters would be utilized to create an appropriate debounce delay. In the first of these strategies, it is assumed that a hardware flag is used to identify when an overflow, compare match, or something else applicable occurs. (Note that, in this document, only overflow or compare match flags are explicitly considered.) Moreover, each of the below techniques are described in the context of an application that intends to respond asynchronously to a switch; in other words, it will be assumed that the following strategies begin (but do not carry out) the appropriate debounce delay within an interrupt handler for an I/O interrupt, upon the to-be-debounced switch changing to an appropriate state. Similar techniques could be implemented for applications that poll (i.e., respond synchronously to) a switch.

NOTE: If debouncing a tactile switch, it is probable that an unintended I/O interrupt will still occur upon a release of the switch. This is the only unintended interrupt that should occur. In any event, recall that the debouncing technique should always prevent such an erroneous interrupt from performing unintended functionality by validating the digital value of the relevant pin after the appropriate debounce delay completely elapses.

USING A HARDWARE FLAG

The first of these next two strategies enables a timer/counter within an I/O interrupt (for the purpose of creating a debounce delay) and polls an appropriate hardware flag within a main routine of the program, to determine when the relevant timer/counter has overflowed or when a compare match has occurred. A description of this strategy is provided below, and an example flowchart of the strategy is given in Figure 2. (Note that the exact implementation of the code could vary from application to application.)

First, the relevant program should configure, *but not enable*, an [1] I/O interrupt for an appropriate pin and [2] a timer/counter system. After these initial configurations, interrupts should be configured globally. (Global interrupt configurations could also come further on, but it is likely most reasonable to do them at this point.)

Next, whenever it is desired to use the relevant switch within a main routine, the I/O interrupt should be enabled and a polling loop should commence for the appropriate timer/counter hardware flag. (The timer/counter should not have yet been enabled to count, so the relevant timer/counter flag should not yet be able to be asserted.) Upon the to-be-debounced switch changing to an appropriate state, the relevant I/O interrupt should then trigger. Within the respective interrupt service routine, the I/O interrupt should be disabled (to prevent unnecessary interrupts) and the chosen timer/counter module should be enabled to count. Following this, the interrupt handler should be terminated.

When the relevant timer/counter module has counted for the designated length of time (i.e., when the relevant debounce delay has elapsed), the pertinent hardware flag should be automatically asserted, and the polling loop should terminate. Thereafter, in the same routine as the polling loop, it should be ensured by the programmer that [1] the timer/counter is disabled, [2] the timer/counter count value and timer/counter interrupt flag are reset to their default states, [3] the pin level of the switch is conditionally checked as described previously in this document, with any relevant operation(s) being performed based on the pin level, and [4] the I/O interrupt flag is reset to its default state. Note that the relevant *I/O interrupt flag* must be reset **only after the debounce delay has completely elapsed**, since this is the only point in which one should be able to safely assume that the switch has stopped bouncing. Finally, whenever additional switch input is needed, the I/O interrupt should be re-enabled.

Overall, although this strategy can be useful because it does not require that a software delay be utilized, it is still very non-modular; this is because the hardware flag must continually be checked within the main program, wherever it is expected that the to-be-debounced switch will be used. In some situations, this is adequate; however, this strategy is not allowed for our course.

NOTE: There are other situations, e.g., not just when debouncing a switch, that a hardware flag should be polled. In these same contexts, a user-defined bit within a register or memory, often otherwise known as a *software flag*, might also be sufficient.

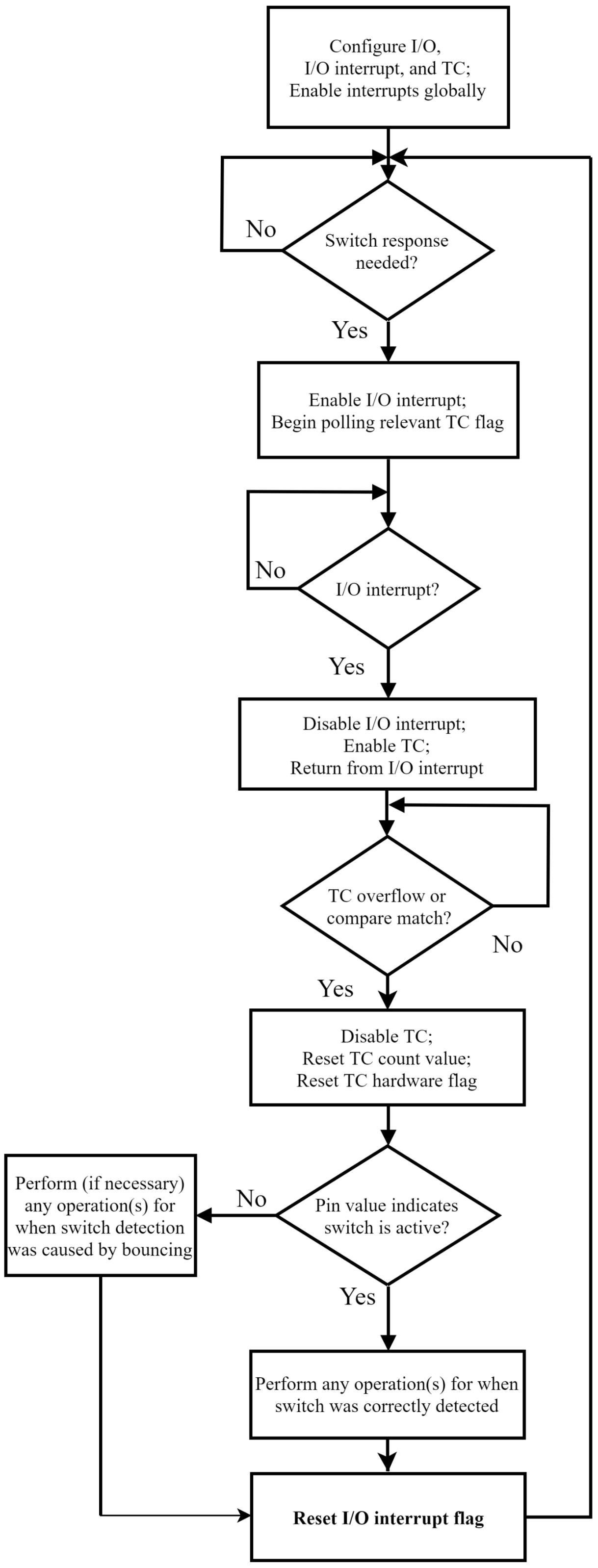
Configure I/O, I/O interrupt, and TC; Enable interrupts globally
Switch response needed?
No
Yes
Enable I/O interrupt; Begin polling relevant TC flag
I/O interrupt?
No
Yes
Disable I/O interrupt; Enable TC; Return from I/O interrupt
TC overflow or compare match?
No
Yes
Disable TC; Reset TC count value; Reset TC hardware flag
Pin value indicates switch is active?
No
Yes
Perform (if necessary) any operation(s) for when switch detection was caused by bouncing
Perform any operation(s) for when switch was correctly detected
Reset I/O interrupt flag

Figure 2: Example flowchart for the second strategy, i.e., the strategy of switch debouncing using a hardware flag

USING A TC INTERRUPT

The final strategy described here involves, in addition to a timer/counter and I/O interrupt, configuring a timer/counter interrupt handler to handle all operations meant to ultimately occur upon a to-be-debounced switch pin changing to a desired state. A description of this strategy is provided below, and an example flowchart for the strategy is given in Figure 3. (Note that the exact implementation of the code could vary from application to application.)

First, the relevant program should configure, *but not enable*, [1] an I/O interrupt for an appropriate pin and [2] a timer/counter system, including a relevant interrupt for an overflow or compare match condition. After these initial configurations, interrupts should be configured globally. (Global interrupt configurations could also come further on, but it is likely most reasonable to do them at this point.)

Next, whenever it is desired to use the relevant switch within the main program, the I/O interrupt should be enabled, *but unlike the previous strategy described, no polling loop for the timer/counter should be implemented – the code can proceed in pretty much any other manner*. Upon the to-be-debounced switch changing to an appropriate state, the relevant I/O interrupt should then trigger. Within the respective interrupt service routine, the I/O interrupt should be disabled (to prevent unnecessary interrupts) and the chosen timer/counter module, along with its interrupt, should be enabled. Following this, the I/O interrupt handler should be terminated, and the processor should return to the relevant previous routine to handle anything else pertinent, while the initiated debounce delay continues silently in the background.

When the timer/counter has counted for the specified length of time (i.e., when the relevant debounce delay has elapsed), the overflow (or compare match) interrupt should trigger. Within the respective interrupt, the programmer should ensure that [1] the timer/counter and its relevant interrupt are disabled, [2] the timer/counter count value and timer/counter interrupt flag are reset to their default states, [3] the appropriate switch pin level is conditionally checked as described previously in this document, with any relevant operation(s) being performed based on the pin level, and [4] the I/O interrupt flag is reset to its default state. Note that the relevant *I/O interrupt flag* must be reset **only after the debounce delay has completely elapsed**, since this is the only point in which one should be able to safely assume that the switch has stopped bouncing. Finally, whenever additional switch input is needed, the I/O interrupt should be re-enabled.

Overall, the most noteworthy aspect of this debounce strategy is that, unlike the previous debounce strategies presented, this strategy is modular, since no other thread of execution within the program is affected by the switch. The only notable downside of this approach is that it is the most complex to implement of those mentioned. However, for our course, this debouncing technique is required whenever asynchronous responses to a switch are desired.

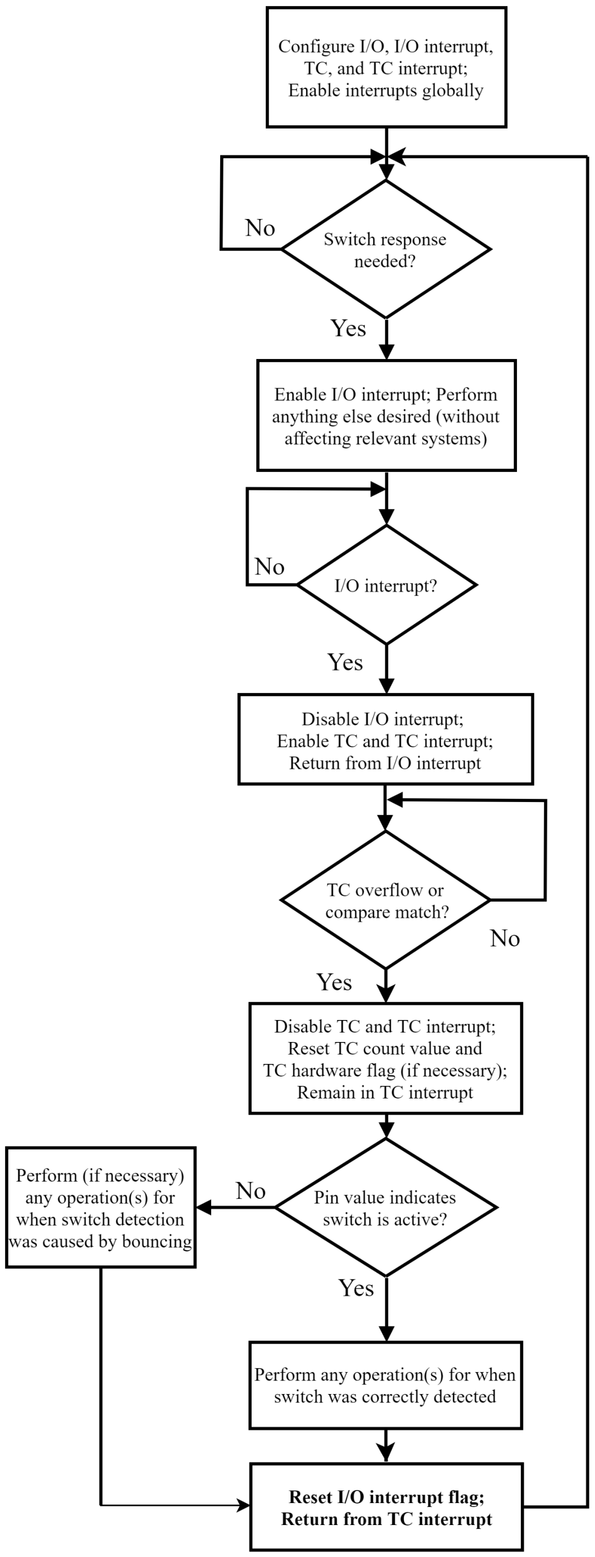
Configure I/O, I/O interrupt, TC, and TC interrupt; Enable interrupts globally
Switch response needed?
No
Yes
Enable I/O interrupt; Perform anything else desired (without affecting relevant systems)
I/O interrupt?
No
Yes
Disable I/O interrupt; Enable TC and TC interrupt; Return from I/O interrupt
TC overflow or compare match?
No
Yes
Disable TC and TC interrupt; Reset TC count value and TC hardware flag (if necessary); Remain in TC interrupt
Pin value indicates switch is active?
No
Perform (if necessary) any operation(s) for when switch detection was caused by bouncing
Yes
Perform any operation(s) for when switch was correctly detected
Reset I/O interrupt flag; Return from TC interrupt

Figure 3: Example flowchart for the third strategy, i.e., the strategy of switch debouncing using a timer/counter interrupt