Software Handling for Capacitive Sensing

Author: Tom Perme
Microchip Technology Inc.

INTRODUCTION

This application note describes various ways of detecting button presses using capacitive sensing. It assumes general knowledge of the sensing process, and it is suggested that *AN1101, "Introduction to Capacitive Sensing"* be read prior to this application note in order to understand the hardware concepts.

Some capacitive sensing solutions on the market offer only a "black box" approach to capacitive sensing, where an IC is purchased and it signals button presses with limited configurability. Microchip's capacitive sensing solution offers the utmost in flexibility because the software routines to detect a button press can be completely user-written. This is not to say the user must develop their own software routines as Microchip provides capacitive sensing routines which may be used to get started immediately with your capacitive sensing solution.

INTRODUCTION TO SOFTWARE

All of the detection schemes described operate on the same fundamental principle that a drop in frequency count from the running average indicates a button press. The basic physical process to scan buttons is to set an oscillator, sensitive to capacitance, to oscillate on a button pad for a fixed time period. After the fixed period, measure the frequency and check if the frequency is different than normal. Then, move the oscillator to the next button pad to scan. Scanning numerous buttons is accomplished sequentially.

There are two primary pieces of code which a user must create. The flowchart in Figure 1 shows the basic flow of a program utilizing capacitive sensing. The first capacitive sensing section of code is "Capacitive Initializations", where the initializations to enable the oscillator, port direction pins and all appropriate initialization settings are made. The second important section of code contains the series of blocks dubbed, "Cap ISR", which are blocks of code executed on an interrupt when the T0IF flag is set. These blocks execute decision making code to determine if a button is pressed or unpressed and to scan all buttons sequentially. Each important block is described in detail in the following paragraphs.

Initialization

To begin, the hardware must be properly initialized. A detailed depiction of the proper settings for the PIC16F88X family is shown in **Appendix A: "Register Settings for the PIC16F88X Family"**. The different families of parts may have slightly different register setting values, but the key bits and signal paths are to be set the same. Therefore, **Appendix A: "Register Settings for the PIC16F88X Family"** may be used as a guideline for which bits to set in the other families of parts where differences in registers occur.

Below is a short checklist to ensure everything is set properly:

- Port Direction and Analog/Digital Selection
- Oscillator Signal Paths Enabled
- Enable Timers and Set Timer0 Prescalar
- Enable Interrupts

Servicing Interrupts

The capacitive sensing is interrupt-based on the Timer0 interrupt signaled by the flag, T0IF. The first thing the Interrupt Service Routine (ISR) should do is check if the flagging interrupt is a Timer0 interrupt or another interrupt. If it is a Timer0 interrupt, then the capacitive sensing must be serviced.

If another interrupt vectors program flow to the ISR, the ISR should service that interrupt, and at the end of the interrupt routine, it must check that Timer0 did not roll over during the ISR. If it did, the T0IF flag will be set, and the sample it represents should be assumed as bad. T0IF should then be cleared and the timers restarted to take another sample. The sample is bad because the fixed period for measurement, based on Timer0, becomes variable if not serviced immediately.

The following sections step through an ISR item in the flowchart, beginning with the "Read TMR0" block.

Servicing Interrupts: Take a Reading

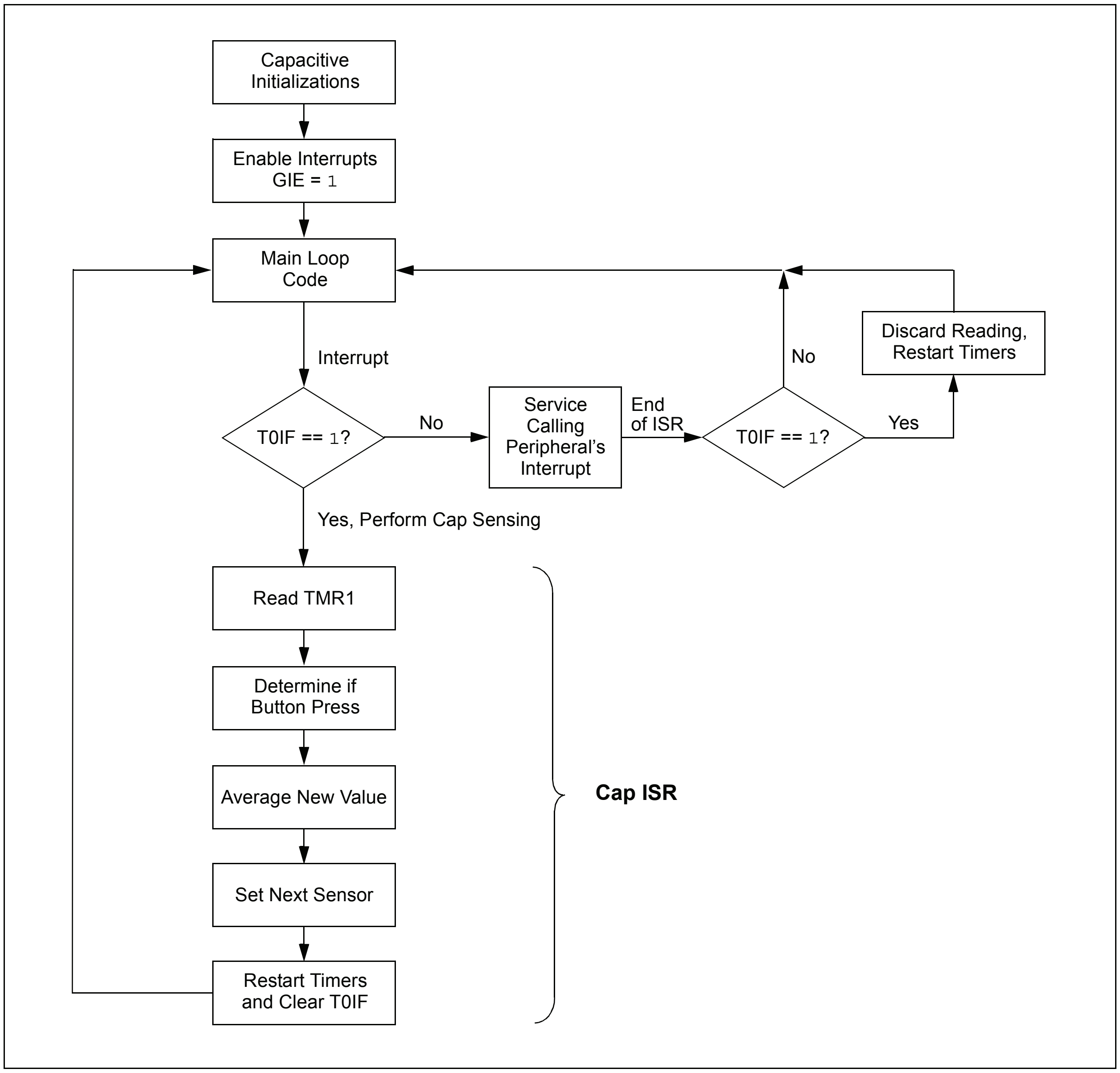
To obtain the reading of the current sensor which has just completed its scan, the Timer1 value must be read. An unsigned integer variable is required to hold the raw value. The code to obtain a reading is shown in Example 1:

EXAMPLE 1: READING FREQUENCY

```
unsigned int value;
value = TMR1L + (unsigned int)(TMR1H << 8);
```

The result in `value` will be the current reading of the sensor which was set to scan on the previous pass of the capacitive service routine. The variable, `value`, will next be compared to the 16-point average to determine if a significant drop in frequency count is present.

FIGURE 1: SOFTWARE FLOW


Capacitive Initializations
Enable Interrupts GIE = 1
Main Loop Code
Interrupt
T0IF == 1?
No
Service Calling Peripheral's Interrupt
End of ISR
T0IF == 1?
Yes
No
Discard Reading, Restart Timers
Yes, Perform Cap Sensing
Read TMR1
Determine if Button Press
Average New Value
Set Next Sensor
Restart Timers and Clear T0IF
Cap ISR

Servicing Interrupts: Determine if Pressed

In this section, a very simple button press detection algorithm will be introduced. It is shown in Example 2.

EXAMPLE 2: SIMPLE DETECTION

```
    if (raw < average-trip))
       // Button pressed
    else
       //Button not Pressed
```

The variable trip holds a value which is the distance in counts below the average that the raw must drop before a button press is detected. For a quick example, if the running average is 9000 and the trip is set at 800, the raw must drop to 8200 before a button is considered pressed.

All the detection schemes operate on the same fundamental principle that a drop in frequency count from the running average indicates a button press. More sophisticated methods are detailed in the **"Button Detection Algorithms"** section along with some of the details and compensations which may be required.

Servicing Interrupts: Averaging

Averaging the current reading is a fairly simple step. To make the averaging efficient, it does not store 16 variables to do a 16-point average. Instead, the current reading is given a weight of 1/16th, while the running average is weighted as 15/16th. The line of code shown in Example 3 performs the 16-point averaging with indexed buttons.

EXAMPLE 3: COMPUTE AVERAGE

```
  average[index] = average[index] + \
       (raw - average[index])/16;
```

Using a number which is a power of 2 for the N-point average, saves processing time because right-shifts can be used instead of software division. When using assembly, one should perform right-shifting, but when using an intelligent C compiler, such as HI-TECH PICC™, the compiler will recognize division by a power of 2 and use right-shifts.

Servicing Interrupts: Preparing Next Sensor

Once the Timer0 interrupt has been serviced, the next sensor to be tested should be set. This involves indexing the next sensor and setting the appropriate connections for comparator inputs.

1. Set Index.
2. Set Comparator Input Channels.
3. (Optional) Set External MUX Lines Control.

Increment the index variable and roll over to zero when applicable. This assumes that an array of averages is created to hold the average values for all the buttons. In Example 4, the use of four buttons is assumed as the standard unaltered capacity of a PIC® device with an SR latch.

EXAMPLE 4: INDEX SEQUENCING

```
    if (index < 3)
       index++;
    else
       index = 0;
```

Note: When using powers of two, an AND operation can simplify rolling over from $2^n - 1$ to zero. Code for 4 buttons: `index = (++index) & 0x03;`

Next, configure the comparator inputs using predefined constants. These constants are derived from the proper settings for registers, CM1CON0 and CM2CON0. They hold the proper signal settings and different bit values on CMxCON0<1:0> for the internal MUX channel to determine which negative comparator input channel is used (C12INx-). The predefined constants are shown in Example 5:

EXAMPLE 5: PREDEFINED CONSTANTS

```
 // C12INx-    0      1      2      3
 COMP1[4] = {0x94, 0x95, 0x96, 0x97};
 COMP2[4] = {0xA0, 0xA1, 0xA2, 0xA3};
```

The comparator registers which must be set to one of these constants are shown below and are based on the index variable. The index for the buttons maps directly to the comparator input channel when using a part's four inherent buttons.

EXAMPLE 6: INDEX FOR BUTTON MAPS

```
CM1CON0 = COMP1[index];
CM2CON0 = COMP2[index];
```

However, when handling many buttons, care must be taken that the comparator input channel is properly set based on the index of the button being scanned. This becomes an important issue when greatly expanding button capacity with external MUXes because the index is then detached from which comparator input channel it represents. For more detailed information on this specific topic, see *AN1104, "Capacitive Multi-Button Configurations"*.

Servicing Interrupts: Restart Timers and Clear T0IF

Timer0 and Timer1 must be cleared each time a reading is started in order to maintain consistent readings. Setting TMR1ON re-enables Timer1. Lastly, the T0IF interrupt flag must be cleared or else the program will immediately return to the ISR.

EXAMPLE 7: SERVICING INTERRUPTS

```
TMR1L  = 0;
TMR1H  = 0;
TMR1ON = 1;
TMR0   = 0;
T0IF   = 0;
```

The next sensor is now prepared, and on the next interrupt caused by Timer0, the capacitive service routine will be run again and complete the scan for that sensor.

SCAN RATE

At this point, each button is being scanned sequentially. A pertinent question to ask is, "How long does it take?" The scan rate for a single button is primarily defined by the equation below:

EQUATION 1:

$$T_{SCAN} = 256 \times (4 \times T_{OSC}) \times PS$$

This is the time it takes for Timer0 to roll over and create an overflow interrupt. The three chief parameters are the number of counts to overflow Timer0 (256), the instruction cycle time of each (4 x TOSC) and the Timer0 Prescalar amount (PS). For a prescalar of 1:4, the value, PS, would be 4. This equation is ideal as it assumes that there are no overhead losses. For instance, when using a computation intensive method, such as the percentage method described later, an additional overhead will be added to this time.

BUTTON DETECTION ALGORITHMS

The first action to create a good system prior to writing firmware is to enable easier detection by creating a sensor with small parasitic capacitance so that a bigger change is detectable. This will make a system function with more ease and less development time.

With a reasonable system, even a very small change can be detected. Being able to control how a change is detected becomes a finer point regarding how the application should behave. A couple of common things desired for systems are simple buttons and sensing a person at a distance.

Microchip has developed several software techniques to detect a button press which work through window glass, Plexiglas® or other non-conductive surfaces. As stated in the introduction, an average is always used, and it is usually helpful to slow this average down because the microprocessor can average in a finger approaching the pad before the finger touches the pad; in which case, a button press never occurs. This is accomplished by averaging every 2nd time through the service routine, or every 4th for example; that is done while each and every pass the if-statement to check if a button is pressed is executed for immediate detection.

The three button detection algorithms are:

- Method 1: Explicit Trip, Fixed Hysteresis
- Method 2: Percentage Trip
- Method 3: Percentage One at a Time, Always Average

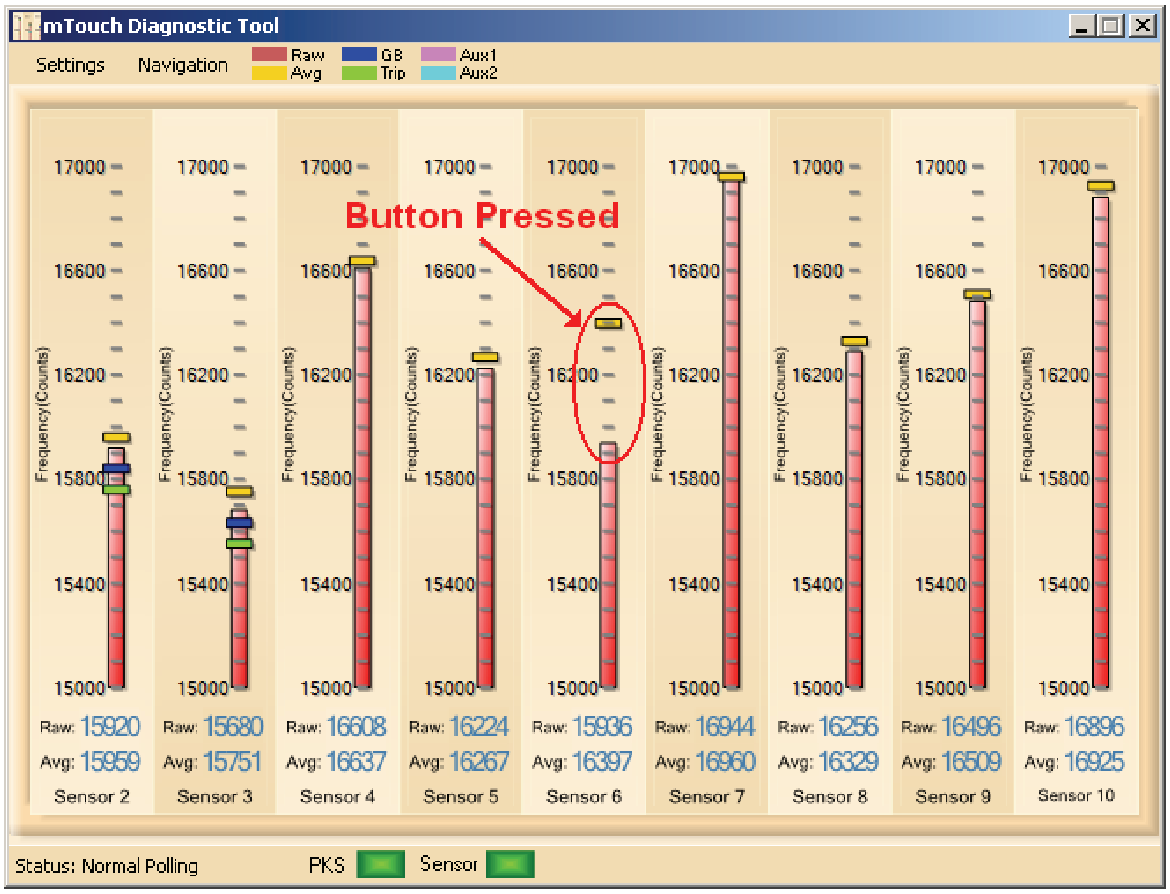
Method 1: Explicit Trip, Fixed Hysteresis

This button detection algorithm is the fastest in terms of execution cycles and deals with raw values directly. It relies on three key items: slowing the averaging down, providing a small hysteresis beyond which averaging stops and knowing a good threshold to set for the trip value. Knowing a good threshold must be determined experimentally, but given a known system, it does not vary greatly. Microchip provides an mTouch Diagnostic Tool to aid in analyzing your capacitive system. This tool can communicate with your system via I^{2}C™, using the PICkit™ Serial Analyzer, and it displays real-time data to characterize your system. It can be very helpful in providing a more intuitive understanding of what is going on in your capacitive sensing system.

To determine the trip threshold and an acceptable amount of hysteresis, the following exercise using the mTouch Diagnostic Tool will demonstrate how to pick good values for a Microchip demo board.

Figure 2 shows Sensor 6 with a finger pressed on the button. The gold bar near 16,400 is the average and the red bar just below the 16,000 tick mark indicates the current raw value. The average has ceased to track the raw value because it has crossed the trip threshold, although it is not shown on the diagnostic tool. The blue and green bars on Sensors 2 and 3 are informational bars which may be set by the user to help visualize trip levels (they do not write or read values to or from the microcontroller).

FIGURE 2: mTouch DIAGNOSTIC TOOL SCREENSHOT



Now, for the system which is being tested, Sensor 6 has a normal unpressed value of about Average = 16397. This value can be read below the graph. The pressed value is Raw = 15936. The difference from the average value is 461. So, a good trip threshold would be 80% of that or roughly 370.

Choose:

```
trip = 370
```

Setting it exactly to 461 would not allow any tolerance for environmental changes or unknown changes. It could result in being unable to cause a press. Setting it too small would make it extremely sensitive, and then it might fire before the button is physically touched. If set too sensitive, adjacent buttons may also unintentionally trigger the button.

Now, there is only 1 large choice remaining: namely, how much hysteresis to provide. Only a little hysteresis is needed and it be determined experimentally. The hysteresis that is required is simply to prevent button jitter. A small hysteresis is desirable to prevent 'stuck' buttons. The simplest example is to assume after a press that the raw value must rise all the way to the previous average. If another button is being pressed, or metal, water or other environmental factors change, including large current draw through the microcontroller, the reading may sag and the raw value may not rise fully to its previous average. Jitter in a reading due to a finger being wiggled while on the button is reasonably small and usually the jitter is within 16 counts if a finger is held still. So, a hysteresis of 64 will provide between 48 to 64 counts of hysteresis for a shaky finger press (or other source of jitter).

Choose:

```
hysteresis = 64
```

A final check for the hysteresis is that it is not greater than the average. This would happen in a system which has a very small change due to a finger press. The current system has a mediocre change of 461, and with the trip at 370, the hysteresis takes 64 away, leaving an average range of 306 within the running average. The design is ready to be implemented. Example 8 shows the implementation in code of the key concepts.

EXAMPLE 8: CODE FOR METHOD 1: EXPLICIT TRIP

```
// Assume the sensor we designed for has an index 6 and there are 16 buttons.
unsigned int  average [16];
unsigned int  trip [16];

CapInit()
{

   // Initializations not shown complete

   trip[6] = 370;     // Set sensor's trip level
}

CapISR()
{

   GetReading();

   // Example for index = 6
   if (raw < (average[index] - trip[index]) ) {
       // Button Pressed
       // 1. Set Button Flag for Sensor # index
       // 2. Do not average (requires no action)

       // 1
       switch(index) {
           case 0:            Buttons.BTN0 = 1; break;
           case 1:            Buttons.BTN1 = 1; break;
                              …
           case 6:            Buttons.BTN6 = 1; break;
                              …
           default   :    break;
       }
} else if (raw > (average[index] - trip[index] + 64)){
       // Button unpressed

       // 1. Clear Button Flag for Sensor # index
       // 2. Perform Slow Average

       // 1
       switch(index) {
           case 0:            Buttons.BTN0 = 0; break;
           case 1:            Buttons.BTN1 = 0; break;
                              …
           case 6:            Buttons.BTN6 = 0; break;
                              …
           default   :        break;
       }

       // 2
       if (AvgIndex < 2)      AvgIndex++;
       else                   AvgIndex = 0;

       if (AvgIndex == 2)
            average[index] = average[index] + ((long)raw-(long)average[index])/16;

   }

   SetNextSensor();
   RestartTimers();

}
```

Note:	Using the PC software, mTouch Diagnostic Tool and a PICkit Serial Analyzer does introduce marginal capacitance, but as long as the system is not operating near the limits, the effect is insignificant. When possible, always test the system using the true system components instead of simulated tests.

Method 2: Percentage Trip

A good way to abstract from the absolute values of the average and raw is to base detection on a percentage change from the average. Doing so makes tuning a capacitive system easier and can provide more reliability. It requires more processing time to do the necessary computations, but it also can reduce memory usage in comparison to Method 1 by not storing an array of trip values unique to each button. The other tasks, such as slow averaging and setting up sensors, are still the same.

When doing mathematical computations to find the average, the result is usually a fractional number, such as 0.05. Instead, it is easier to work in the microcontroller with whole numbers, therefore the result will be multiplied by 100 to achieve 5 for 5%. Multiplying by 1000 can include the first tenth of a percent and would represent 5.2% with 52. Code that performs the percentage follows in Example 10.

Note that if the percentage is negative, the raw value is greater than the average. This range should be included in setting the average, and so, negative values are ignored and set to zero for making averaging logic simple.

Now the trip level will be a percentage “ON” that the button is pressed, PCT_ON. This threshold, like the trip from Method 1, also comes from experimental data, but can be anywhere between 1% for a weak press to 25% for a very strong press. A separate value for the percentages which is less than a percentage “OFF”, PCT_OFF must also be set, and making it different than PCT_ON provides hysteresis. Lastly, a slowed average is performed as done before until the average reaches the button off threshold.

For an example, assume the following values. These values are made to be nice even numbers, but they are realistic numbers.

Unpressed Average	15000
Pressed Reading	13500
Difference	1500
Percentage Difference	10%

So, to apply a safe percentage on, a good PCT_ON value would be about 8%, or 80% of the 10% change. A 1% hysteresis is plenty for the percentage off, because the absolute charge is of good size; a 1 percent hysteresis is 150 counts for this example. This is typically the case for using the percentage-based method and only differs for very low-frequency count readings. So, PCT_OFF should be 7%. If more hysteresis were desired, choose 6% or 5%.

Choose:

```
PCT_ON = 8
PCT_OFF = 7
```

EXAMPLE 9: PERCENTAGE CALCULATION

```
long percent;

CapISR()
{
...
     percent = ((long)average[index] - (long)raw[index]);
     if (percent < 0){
          percent = 0;
     } else {
          percent = percent * 100;
          percent = percent / average[index];
}
...
}
```

The design is now ready to be implemented and is done so by the code shown in Example 10:

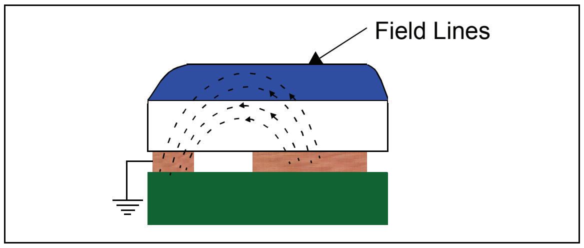
EXAMPLE 10: PERCENTAGE IMPLEMENTATION

```
// Define Percentage on/off
// presses for all buttons
#define PCT_ON     8
#define PCT_OFF    7

CapISR()
{
 GetReading();

 // GetPercentage() is the code
 // from the previous example.
 percent = GetPercentage();

 if (percent < PCT_OFF) {

    // 1. Clear Button's Flag
    // 2. Perform Slow Averaging

    // 1 (just like Method 1)
    switch(index) {
      case 0:   Button.BTN0 = 0; break;
                ...
      default:  break;
    }

    // 2 (just like Method 1)
    if (AvgIndex < 2)  AvgIndex++;
    else               AvgIndex = 0;

 } else if (percent > PCT_ON) {

    //  1. Set Button's Flag
    //  2. Do not average (take no
    //     action)

    // 1 (as before)
    switch (index) {
      case 0:   Button.BTN0 = 1; break;
                ...
      default:  break;
    }
 }

 SetNextSensor();
 RestartTimers();
}
```

Method 3: Percentage Voting

A third suggested method exists to counter drastic environmental changes, such as the sustained appearance of water near a button. In this method, only one button may be pressed at a time, and the most pressed button is the one that is chosen. It is "voted" most pressed. If enough water is spilled on a surface of sensors, it will increase the dielectric to nearby ground and cause a false button press. It is because of water's high dielectric, that it creates a strong capacitive link over the entire surface on which it is present.

FIGURE 3: WATER ON SENSOR GLASS w/FIELD LINES



There is no perfect capacitive solution to the problem of water splashing on the sensing surface simply due to physics. A water drop splashing on the surface can look just like a finger press, but to attempt to combat standing water, this third method was developed.

When water splashes on a surface and stays there, often a finger press can still be detected through the water. In this case, the running average must be set based on the conditions experienced, having water on the sensing surface. This requires the average to be run all the time and button presses have only a finite duration until a new average settles to the lower pressed value.

When water spans more than one button, the most pressed button is often the correct button. For example, assume water spans three buttons, called 1, 2 and 3, as in Figure 4. The system has averaged to its state with water on the surface and a person presses button 3. All three buttons will drop in frequency even more due to the water making capacitive connections to all three pads. However, the frequency count on button 3 will likely drop more than those of the other buttons 1 and 2. Now, the algorithm dictates the button with the highest percentage, as described in Method 2, should be selected from all buttons.

COMPARISONS

The three methods each have pros and cons, but all are suitable for use. If program and RAM memory are limited, as on a PIC16F610, the percentage or percentage voting methods may or may not fit, and will consume a lot of device resources. Using a larger part, such as a part from the PIC16F887 family or the PIC16F690 family will provide more RAM to enable use of the percentage methods. Table 1 shows some comparative trade-offs between the three methods.

TABLE 1: SOFTWARE METHODS COMPARISON

	RAM/PGM Memory	Ease of Setup	Foreign Objects (H_20)
Explicit	+	-	-
Percentage	-	+	-
Percentage Voting	- -	+	+

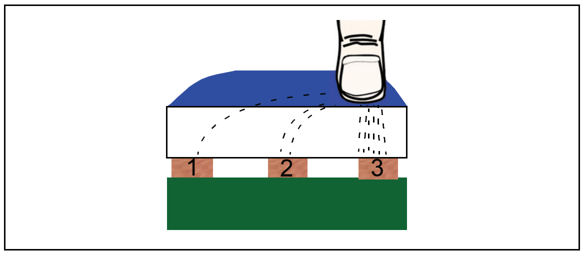
The percentage-based routines use more math to determine if a button is pressed or not, so they require more memory. The percentage voting system additionally requires two arrays to sort which button is the most pressed, which requires more memory on top of the percentage math routines.

They key benefit of the percentage-based routines is ease of setup. By abstracting one step away from the raw values, it is easier to have good working percentage values for several boards, or even across designs. Where the abstract values may change significantly across designs, or even on different buttons of a specific design, the percentage changes are roughly the same. Therefore, it helps to get to a working solution quicker than determining proper trip values as required for Method 1: Explicit Trip.

Additionally, using the percentage voting system can help with conductive foreign objects in the proximity of the sensing area. It is very helpful with thin films of water, where the other methods are prone to detect presses on all covered buttons, but when pooled water occurs, or metal, or another highly dielectric or conductive material is nearby, it will lose its reliability like the other two methods.

The first two methods, Explicit Trip and Percentage Trip, are intended to be good starting points to use as initial methods which may be customized for the given application. Other features, such as time-outs on buttons, activate a button on the press and release of a pad, and other application desired features can be integrated into these methods.

FIGURE 4: FINGER THROUGH WATER



Determining a button press like this is not ensured, though, but it provides reasonable reliability with only small to medium amounts of water on the board. For a small film of water on the sensing surface, this works well; likewise for condensation or dew. However, when the surface is heavily flooded, the method loses its reliability.

The best way to counter water problems is to design the physical system such that water has a hard time hitting, running over or standing on the sensing surface. For example, mount the button surface at an angle so that water can not stand on the surface; then, only a water film might be present. Reliability is improved greatly when standing water is not a factor, and then only splashes of water drops can cause false detection problems.

Code to implement the capacitive service routine of Method 3 is provided in the software in addition to this application note. The code is more involved than the code in the other methods, and the key considerations are that the average is always performed and that the most pressed button, determined by a percentage for each, is considered the pressed button when any of them are above a percentage on threshold.

PRECAUTIONS

Timer1 Overflow

Since the principle measurement is read from the TMR1 value, Timer0 must overflow and cause an interrupt to read Timer1 before Timer1 can overflow. This is determined by the Timer0 prescalar and the oscillating frequency of the system. A typically safe and good prescalar to start with is PS<2:0> = `0x2` in the OPTION register. A longer Timer0 period will allow more counts, and a steadier reading than an extremely fast period, but it does so at the expense of the longer period to scan an individual button.

Timer1 is a 16-bit timer, so as long as the frequency count readings are well below the unsigned integer maximum of 65,535, then Timer0 and the prescalar bits are acceptable.

Stuck Buttons

A stuck button, as described here, is a button that turns on but does not turn off when released. “Stuck” buttons often come in a form of two varieties. The first is a poorly tuned button which tends to cause some current draw. The second type is due to large abrupt current draw, and this variety is discussed in the **“Large Abrupt Current Draw”** section.

As a designer, constantly tuning trip thresholds is not preferable, and so it is desirable to keep it to a minimum, or better, non-existent. With an established production design, once the system's optimal configurations have been found for trip values or percent thresholds, they often do not need to be changed for each system produced. Both the trip and percent threshold button detection schemes described earlier can suffer from poor settings, although it is more problematic with absolute trip thresholds.

The typical cause is that on a button press, a reaction to the press causes extra current to draw through the part, such as an LED, which pulls down the frequency count. When the button is released, even though the person's finger is gone, the current still pulls the frequency down some. If the trip is set too small, such that a very easy press turns on the button (and the current drawing reaction), the amount of pull down on the frequency may be enough to keep the frequency count below the trip threshold. Also, if the trip threshold is too small, an adjacent button press may trigger a false press on the button next to it.

The fix is to make the largest trip threshold possible without losing reliability that a touch is detected. In a previous example, it was suggested to use 80% of the change in frequency count caused by a finger press. This is done because a lighter finger press causes less change than a harder one. So, the lightest press desired should be tested, and then some margin of safety should be provided for that press; any harder press will be detected.

Stuck buttons under normal, non-abusive use can be easily prevented. The **"Large Abrupt Current Draw"** section covers issues regarding large and abrupt current draws through the microcontroller, which create more problems, and require more than good settings to fix.

Large Abrupt Current Draw

A significant increase in current draw through the PIC MCU can cause the oscillator frequency to slow down slightly. If unexpected, this can potentially create a stuck button condition. Unlike some competitors, Microchip can, however, supply current to numerous LEDs while performing capacitive sensing, a simple example of high-current draw. If a very large current must be switched on, increase current consumption gradually if possible. The amount of current draw is not a problem, but rather the abrupt change in current draw is.

Interleaving periods of time for button press scanning and separate current draw times is a good way to separate the two tasks.

Without interleaving, often no compensation is required, but for some objects, such as sliders which have many buttons in close proximity, the combination of capacitive "cross-talk" between adjacent buttons and driving LEDs can create stuck buttons. A button press will drag down the frequency of an adjacent button slightly which is also undesirable.

When current consumption compensation is required for a single button and a large abrupt current usually appears, one idea is to manually adjust the average to a point closer to the trip threshold than it currently is resting. It is application-dependent if this is an acceptable thing to do, but in many cases, an abrupt current change can be expected from looking at the design. If a 50 mA load is expected instantaneously in response to a button press, it can be accounted for. Lowering the average artificially allows the button to be released more easily.

For buttons where one button creates a situation that causes another button to become stuck like the slider, if it is ok to do so in the application, reset the average of the stuck button to its current raw reading. The previously stuck button will now be calibrated for the system's current configuration with LEDs drawing current, and/or a finger still pressed, etc., and it will then be easier to press another button to change the level of the slider.

Lastly, if the current draw of the device being switched on is too great, an external driver may be used to avoid the current going through the microcontroller.

Insufficient Hysteresis

If very little hysteresis is given, a capacitive button that switches a current load, like an LED, may enter an oscillatory state. Typically the button is successfully pressed, but with the load drawing current, there is enough variation to continue switching between the on and off thresholds for the button. The simple fix is to provide more hysteresis between the threshold for on and the release threshold.

Kitchen Condiments

The household kitchen environment has a set of issues other environments are unlikely to have. A measure of safety and quality is for a kitchen appliance to receive a splash of ketchup and not indicate a false button press. Throughout this section, ketchup will be discussed, but keep in mind it applies to other condiments.

A large splash of ketchup, like squirting a bottle on an interface panel, will likely trigger a drop in frequency in a capacitive system. Appliance safety measures require that this not indicate a false trigger, since it is ketchup and not a person's finger. Hoping the splash will be a small enough quantity is not a good solution. There is no way to change physics; so the task is to create firmware that will properly detect or not detect a ketchup splash given the hardware setup.

Since it is not always possible to prevent the drop in frequency, watching for a press and release in a certain time period is a viable solution. This requires that a person press and release the button within a fixed time, say one second. In firmware, this is seen as a drop of frequency and an increase of frequency within the allowed period. If a ketchup splash occurs, it will stick and linger on the surface, and a release will not occur. The firmware can even be more advanced, requiring the button to be held for a certain time before it can be released to prevent a very rapid press.

Smaller splatter effects, such as grease from a pan or stir-frying, will have a much lesser effect. Usually, these splatters have no appreciable effect on the system, and only the large changes, which significantly increase the capacitance over the button, present possible concerns. These items are typically a water-based item, like ketchup, mustard, and of course, water itself. All of these items have very high dielectric constants, which is the physical reason for the capacitance increasing when it is introduced on the panel surface.

CONCLUSIONS

Software is provided with this application note to aid in understanding and expediting design. The software to drive capacitive sensing can be either very simple or can handle complex algorithms for button detection. Since the software may be easily changed, the user has more ability to define how their capacitive system should operate.

Additional reference materials include:

AN1101, “Introduction to Capacitive Sensing”

AN1102, “Layout and Physical Design Guidelines for Capacitive Sensing”

AN1104, “Capacitive Multi-Button Configurations”

APPENDIX A: REGISTER SETTINGS FOR THE PIC16F88X FAMILY

The following registers can be found in the *"PIC16F882/883/884/886/887 Data Sheet"* (DS41291). Detailed explanations of each bit may be found in a part's Data Sheet. These register settings provide a guideline for setting the other families' registers.

REGISTER 8-1: CM1CON0: COMPARATOR C1 CONTROL REGISTER 0

R/W-0	R-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
C1ON	C1OUT	C1OE	C1POL	—	C1R	C1CH1	C1CH0
bit 7							bit 0
1	0	0	1	—	1	0	0

REGISTER 8-2: CM2CON0: COMPARATOR C2 CONTROL REGISTER 0

R/W-0	R-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
C2ON	C2OUT	C2OE	C2POL	—	C2R	C2CH1	C2CH0
bit 7							bit 0
1	0	1	0	—	0	0	0

REGISTER 8-3: CM2CON1: COMPARATOR C2 CONTROL REGISTER 1

R-0	R-0	R/W-0	R/W-0	U-0	U-0	R/W-1	R/W-0
MC1OUT	MC2OUT	C1RSEL	C2RSEL	—	—	T1GSS	C2SYNC
bit 7							bit 0
0	0	1	1	—	—	1	0

REGISTER 8-4: SRCON: SR LATCH CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/S-0	R/S-0	U-0	R/W-0
SR1(2)	SR0(2)	C1SEN	C2REN	PULSS	PULSR	—	FVREN
bit 7							bit 0
1	1	1	1	0	0	—	0

2: To enable an SR latch output to the pin, the appropriate CxOE and TRIS bits must be properly configured.

REGISTER 8-5: VRCON: VOLTAGE REFERENCE CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
VREN	VROE	VRR	VRSS	VR3	VR2	VR1	VR0
bit 7							bit 0
1	0	0	0	0	1	1	1

REGISTER 3-3: ANSEL: ANALOG SELECT REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
ANS7(2)	ANS6(2)	ANS5(2)	ANS4	ANS3	ANS2	ANS1	ANS0
bit 7							bit 0
0	0	0	0	0	1	1	1

2: Not implemented on PIC16F883/886.

REGISTER 3-4: ANSELH: ANALOG SELECT HIGH REGISTER

U-0	U-0	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
—	—	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8
bit 7							bit 0
—	—	0	0	0	1	1	0

ANSEL bits selected enable all four comparator inputs.

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

06/25/07