Keywords: proximity sensor, als, optical, infrared, sensors, tablets, laptops, notebooks, human presence detection, touchscreen, smartphone, gesture detection

APPLICATION NOTE 5267

Touching Air: Wake Your Device with a Simple Gesture

By: Ilya Veygman, Strategic Applications Engineer
Jun 15, 2012

Abstract: This article discusses the idea of waking up a touch-screen-based device, such as a tablet, using a very basic form of gesture recognition. Several ideas are discussed about implementing this using only a proximity sensor. These include considerations on physical layout, speed limitations, detection thresholds, and an overview of high-level system integration, such as context. Some example code is provided as well to illustrate software implementations.

A similar version of this article appears on *Electronic Products*, June 1, 2012.

Introduction

This article discusses how to wake up a touch-screen-based device such as a tablet...without a touch. Instead you use a very basic form of gesture recognition and a novel proximity sensor. Topics discussed include physical layout, speed limitations, detection thresholds, high-level system integration, and the "human" factor. Example code illustrates software implementations.

Cooking Up Some Wild Ideas

If you ever used a touch-screen device while cooking, you probably noticed that following a recipe from your device is not as easy as it seems. Tech-savvy cooks, such as yours truly, like to use tablet computers or smartphones to reference recipes while making dinner. "Fine," you say, "but what is so challenging about that?" The handheld device normally goes to sleep after a minute or two, since keeping the screen on can burn up considerable power. Then when you want to reference something and the device is asleep, you are faced with two choices: either force the screen to stay on permanently, or risk smudging the screen with food-stained hands. There's always the option of washing one's hands every time that something has to be checked, but constant washing and drying is tedious and wastes water.

This is where I ask myself, "Can I avoid keeping the screen on permanently and also avoid the risk of fouling the device?" There is, actually, a way to do both. You can gesture for the screen to come back on without actually touching it. Seems complicated, right? Fortunately, this is simpler to do than it sounds.

Getting Close to Proximity Sensors

Many touch-screen devices, especially smartphones, have infrared (IR) proximity sensors built in. These sensors are generally used to turn the screen on and off during a call and, thus, prevent undesired input from being applied to the phone. This sensor, and some clever software design, is all you need to make a touch-free wakeup operate with the wave of a hand.

The basic idea is when the device is asleep, i.e., when its touch screen is turned off and the applications processor is in a low-power mode, have the proximity sensor "look" for a large enough change from the background readings and react appropriately. This is almost exactly how the proximity sensor already operates to turn off the screen during a phone call. In our application, the data is simply interpreted somewhat differently.

You start by noting the amount of "normal" background counts read by the sensor. This may be zero counts, although it is useful to account for systemic offsets (e.g., backscatter or crosstalk), when applicable. Then set it up to either trigger an interrupt or send a signal to the applications processor when the signal climbs past the threshold that was set. This will then cause the system to come back online and turn on the screen. Overall, this is quite straightforward and can be implemented with an ambient light and IR proximity sensor.

This demonstration uses the MAX44000, which takes a proximity reading every 1.56ms or as slow as every 100ms (when interleaved with the ambient light sensor). Assuming a maximum detection range of 10cm and an LED with a radiation angle of ±15°, then the coverage area is about 22cm^2 or approximately 5.35cm across. A target moving across this screen area must be sampled at least once to be seen. Hence, the fastest gesture that can be reliably sensed, or "picked up," in the slowest and also lowest-power sampling speed is about 0.53mps. We are also assuming here that the sensor needs to sense only one sample above a threshold for it then to recognize a simple target passing through the coverage area.

It's All in the Wrist...

As a theoretical approach, this is very simple. When the device enters sleep mode, set its proximity sensor to scan the environment and send an interrupt when it detects a target, indicated by the signal moving above a preset threshold. This can be done by simply polling the sensor repeatedly over the I^{2}C interface. Unfortunately, this can also consume more power than most operators would like.

This is the point at which the specific capabilities of your proximity sensor become most important. The MAX44000 sensor is designed to let the applications processor take a more hands-off (and lower power) approach.

By enabling the MAX44000's internal proximity interrupts (bit 1 in register 0x01), you can write the wakeup threshold to an internal register (0x0B and 0x0C). When the proximity reading passes this threshold, an interrupt flag is raised. This drives the MAX44000's active-low INT pin low, which indicates an interrupt on the external line. When the applications processor sees that this line has been driven low, it can be either awakened to bring the device out of its low-power state and bring the screen back online, or it can do whatever else needs to be done.

...But Let's Not Hand-Wave Through This

As is often true, the practical application is not as easy as the theoretical. Unfortunately, hands-free wakeup is not quite as simple as seeking one sample above a threshold. Instead, there are several factors to consider when attempting to implement this design.

Signal Level and Layout

Perhaps the most critical consideration is the signal level chosen to trigger a wakeup condition. An important trade-off exists between the responsiveness of the system and its immunity to false detection. Setting the threshold low makes it easier to detect an input (e.g., your waving hand), but increases the risk of false detection from transient noise or incident motion. Conversely, too high a threshold will almost surely reduce the likelihood of a false detection to zero, but it can also cause the system to only detect very close targets or even be unresponsive to most any input (e.g., your frantic waving hand).

The best way to address this is, first, to mitigate the amount of noise in the system. This can be done by optical solutions or careful electrical routing and component placement. By lowering the noise floor, you reduce the chances of a false detect. Next, choose an "average" detection distance (e.g., 4cm to 5cm) and measure the signal with a benchmark target. An 18% gray card is ideal. Note that it might seem obvious, but if the application uses a dark glass in front of the sensor, then the measurements should be done with that glass in place. The signal level measured can be used as a best guess for setting the threshold. A good rule is to set the level at 8% to 15% of full scale, although your level can vary.

The MAX44000 sensor's proximity threshold registers let you set a wakeup threshold based on the above experiment. **Figure 1** shows a signal versus distance plot for an 18% grey card with a 100mA drive current and no glass in front of the sensor. The blue line is a possible choice for wakeup threshold.

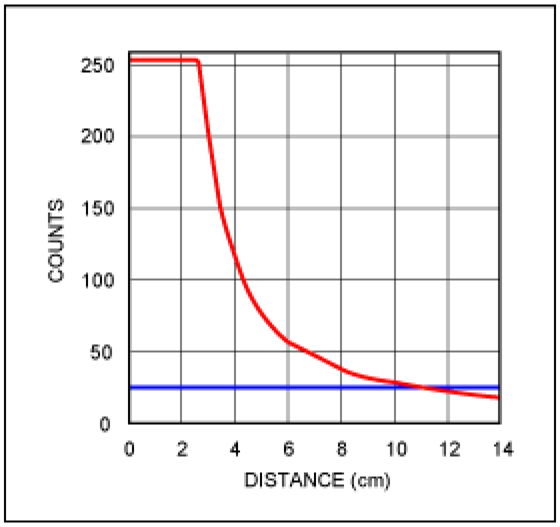

Figure 1. Distance vs. signal strength using the MAX44000 proximity sensor with an 18% grey card, 100mA current, and no glass.

Noise and Lowpass Filtering

If noise is still a concern, you can implement a lowpass filter to clean up the signal. There are two ways to do this with the MAX44000, because it also has a few bits to enable threshold persistence before an interrupt flag is raised. This setting, which requires the proximity reading to stay *outside* the threshold for a set number of samples, can be used to reduce the effects of noise.

A slightly more complex method involves storing the readings from the sensor in a data queue, then applying a customized FIR filter to them in software. Unfortunately, this approach has a flaw. If the sampling rate of the proximity sensor cannot be increased, then the rate at which you can swipe your hand through the sensor's field of view and still be detected is reduced. The 100ms effective sampling rate is particularly vulnerable to this. When using the persistence threshold, this reduced detection can be as much as 16x (although the 4x persist should be enough).

Wave Speed

This brings us to the next consideration, wave speed. The maximum speed is determined by first, the sensor's field of view; second, the distance of the wave from the sensor; third, the sampling rate; and fourth, the threshold level. Determining the first two criteria is obvious: the angle that the sensor can see, coupled with the distance of the target from the sensor, can be used with some basic trigonometry to calculate the distance which the target can travel while the sensor tries to pick it up. For example, if the sensor's viewing angle is 30 degrees in total and the valid range is up to 10cm, then the target can travel 5.35cm over the sensor and still be seen. This is a coverage area of about 78cm^2. This linear distance, coupled with the sampling rate, implies a speed limit. Specifically, if the sampling rate is T, then the target must cross the visible area in no less than T. For example, if T is 100ms (the MAX44000's slowest rate), then following the previous example, the maximum speed theoretically allowed is 1mps. (This is actually quite fast.) You might want to capture multiple samples to confirm your assessment, so this speed limit will likely be reduced.

The detection threshold also plays a part in the maximum speed allowed. As a rule, the lower the threshold, the faster the gesture that can be picked up. As mentioned above, this low threshold should be carefully selected to avoid false detections.

The "Human" Factor

This application is subject to the inconsistencies of the human hand and the person waving it. Consequently, the design ought to involve use cases to determine how fast typical users move their hands in front of the screen, their distance from the screen, and whether they are wearing gloves. This interaction with a device can also vary greatly among applications like smartphones, tablets, or something on an automobile dashboard. Ultimately, the application must account for these user-interface and experience variables during the design process.

There is one final thing to consider, when is a wave not a wave? To be more specific, how can a device know if a received signal comes from a waving hand or another simple movement like being placed inside a case; a pocket or backpack, or face-down on some surface? We need to face the truth here—there is no easy answer to this question, unless context is given to the device. How that is done is a different matter entirely.

In the end, you can choose to implement this wakeup solution only when the touch-screen device is running certain applications or you can require the user to enable it. Additionally, many of these devices have accelerometers that can tell the device when it is lying face-down. The feature can also be disabled if the device was put to sleep manually by the user (e.g., when the device is turned off).

Try It Yourself

For your convenience, three pieces of sample code are attached to this article. The first implements a conceptually simple version of this wakeup solution using manual checks of the MAX44000's proximity reading. The second code expands on the first and implements the filtering concept discussed earlier. The last code shows a quick way of doing this wakeup using the MAX44000's interrupt feature.

Example Code 1

```
__interrupt void TimedInterrupt( void )
{

  uint8 proximity_counts;
  ....
  ....


  if ( device_status == SLEEP_MODE )
  {
    // read one byte from register 0x16
    proximity_counts = read_i2c_register(MAX44000_ADDR,0x16,1);
    if (proximity_counts > WAKEUP_THRESHOLD)
    {
      device_status = WAKE_MODE;
      ...
    }
    else
    {
      // do whatever it is you need to in sleep mode
      ...
      ...
    }
  }

  ...
  ...

}
```

Example Code 2

```
// example interrupt function where this might be implemented
__interrupt void TimedInterrupt( void )
{

  uint8 proximity_counts;
  uint8 filtered_counts;
  ....
  ....


  if ( device_status == SLEEP_MODE )
  {
    // read one byte from register 0x16
    proximity_counts = read_i2c_register(MAX44000_ADDR,0x16,1);

    // weights[QUEUE_SIZE] contains the filter weights for the FIR filter
    // data_queue[QUEUE_SIZE] is a FIFO queue meant to be the input to the
filter
    filtered_counts = fir_filter(proximity_counts,weights,data_queue);

    if (filtered_counts > WAKEUP_THRESHOLD)
    {
      device_status = WAKE_MODE;
      ...
    }
    else
    {
      // do whatever it is you need to in sleep mode
      ...
      ...
    }
  }

  ...
  ...

}

/**
 * fir_filter()
 *
 * Implements an FIR filter in the form
 *   y = w[0]*x[0] + w[1]*x[1] + ... + w[QUEUE_SIZE]*x[QUEUE_SIZE]
 *
 * Arguments:
 *      uint8 input - newest datapoint taken (that is, x[0])
 *      uint8 *weights - w[0]...w[QUEUE_SIZE]
 *      uint8 *queue - the discrete sequence x[0]...x[QUEUE_SIZE]
 *
 * Returns:
 *      The FIR-filtered output, y
 */
uint8 fir_filter(uint8 input, uint8 *weights, uint8 *queue)
{
  uint8 i;
  int sum = 0;

  // pop first entry in the queue, then
  // push new data into the last position
  push_into_queue(queue,input);

  // input is now x[0]
  for (i=0; i<QUEUE_SIZE; ++i)
  {
    sum += weights[i]*queue[i];
  }

  return (sum/QUEUE_SIZE);
}
```

Example Code 3

```
// this handles hardware-level interrupts on the micro
__interrupt void irq_handler( void )
{
  ...

  // if the hardware interrupt came from the MAX44000 sensor
  // pulling its \INT pin low
  if ( irq_source == MAX44000 )
  {
    // if the device is in sleep mode
    if (device_status == SLEEP_MODE)
    {
      device_status = WAKE_MODE;        // wake up the device
      ...
      // reconfigure whatever else you need here as the system wakes up
    }
    // otherwise, handle it however it is you wish
    else
    {
      ...
    }
  }

  ...
}

/**
 *      configure_max44000_for_sleep_mode()
 *
 *      Sets up the MAX44000 to trigger a hardware interrupt when the
proximity
```

```
 *      counts go above some set threshold.
 *
 *      Arguments:
 *              uint8 upper_threshold - the set threshold (8-bit mode)
 *
 *      Returns:
 *              n/a
 */
void configure_max44000_for_sleep_mode(uint8 upper_threshold)
{
  uint8 max44000_thresh_registers[] = {0x0B,0x0C};
  uint8 max44000_upper_thresh[] = {0x40,0};

  max44000_upper_thresh[1] = upper_threshold;

  // do a consecutive write of 0 followed by upper_threshold to
  // registers 0xB and 0xC, respectively
  //    MAX44000_ADDR is usually 0x94
  // interrupt will trigger only if proximity value is above the threshold
  write_i2c_register(MAX44000_ADDR,max44000_thresh_registers,
                              max44000_upper_thresh,2);


  // write to bits 2 and 3 of register 0x0A here if you wish to set the
  // persist time to anything other than one sample

  // writes to register 0x01 to enable interrupts on the MAX44000
  max44000_enable_interrupt();

  return;
}
```

Related Parts		
MAX44000	Ambient and Infrared Proximity Sensor	Free Samples