8

Physical Interaction Design, or Techniques for Polite Conversation

By now, you've got the basic electronic and programming techniques down and you are starting to combine them to pull off your particular idea. This is a good time to step away from the technology for a moment and consider how well your project works for your users. For the first part of this chapter, we'll discuss some ways to approach that problem and lay out some basic interaction design guidelines. In the second part of the chapter, we'll provide some techniques for putting these approaches into action.

The Conversation: Listening, Speaking, and Thinking[1]

In any well-designed physical computing application, the flow of activity between the person and the computer should follow the same comfortable flow of a good conversation. Designing the system so that this happens is what interaction design is all about. This means balancing the timing of your listening, thinking, and speaking to coordinate with the expectations and patterns of the user. When you do this work well, the interaction between the person and the computer flows naturally enough that the person doesn't have to think consciously about their performance, but only about the overall result.

Listening

In actual conversation, we don't often plan the taking of turns. Human beings are capable (to a limited degree) of talking and listening at the same time. When a listener wants to interrupt a speaker, she gives subtle physical cues, and the speaker knows to stop talking and listen. There are also natural pauses in a conversation for the listener to

[1] The ideas in this chapter rely heavily on Chris Crawford's explanation of computer interactivity in *The Art of Interactive Design: A Euphonious and Illuminating Guide to Building Successful Software* (No Starch Press, 2002).

digest information and prepare a response. We know intuitively that when we present information in conversation, we have to give our listeners time to digest the information. We know how long it takes to perceive a change or digest an idea consciously, and we factor that knowledge into our conversation. We give each other that time, and if we feel that the person we're speaking to should have responded, we prompt them to see if they're still understanding us: "Do you get it?" Listening for cues while speaking takes a level of sophistication that we seldom give ourselves credit for, because we're so well trained in doing it that we don't give it a second thought.

When you program a computer to interact with the world, however, you realize how much we take for granted in the course of everyday human interaction. A computer can't spontaneously react to a shout or a movement. If it's not listening when the event happens, it misses it. In fact, the very idea that a shout or a movement is an event that requires response is something that's got to be programmed in advance. It's an important notion because all interaction is made up of events or physical phenomena that must be sensed, interpreted, and responded to. In order to plan interaction between computers and humans (or anything else in the world) at a physical level, the first step is to teach it to listen. You have to articulate the possible events that the computer will respond to, define those events in terms the computer can sense, assign meanings to the events you've defined, and choose an appropriate response to each event. If an event's meaning changes based on the events that precede it, you've got to give the computer instructions about that, too.

There are two main quantities you need to consider when you detect actions with sensors: how intense the sensation was, and how long it took. When you're dealing with digital input sensors, you'll only have two possible values for how intense the sensation was: either you sensed something or you didn't. With analog sensors, you'll have a range from the most intense to the least. Since your sensors convert other forms of energy into electrical signals, you measure the intensity of the signal in volts. To measure how long the event took, you use seconds, microseconds, or milliseconds, depending on the event. To describe the event, then, you can use a graph of voltage and time. For example, an analog sensor might produce a graph like the one in Figure 8.1.

And a digital sensor might produce a graph like the one in Figure 8.2.

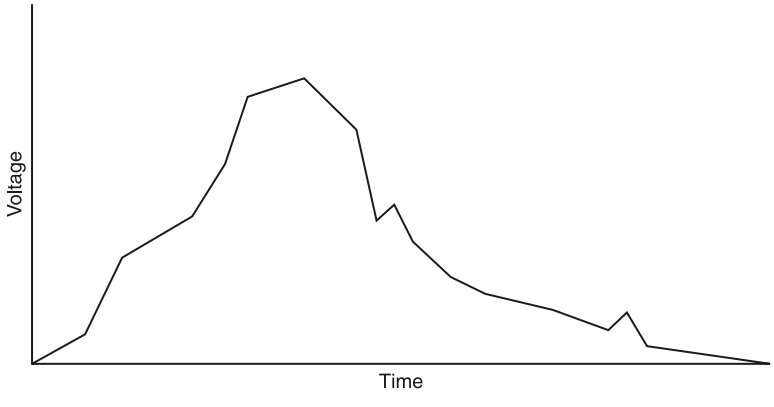

Figure 8.1
Analog sensor readings over time.

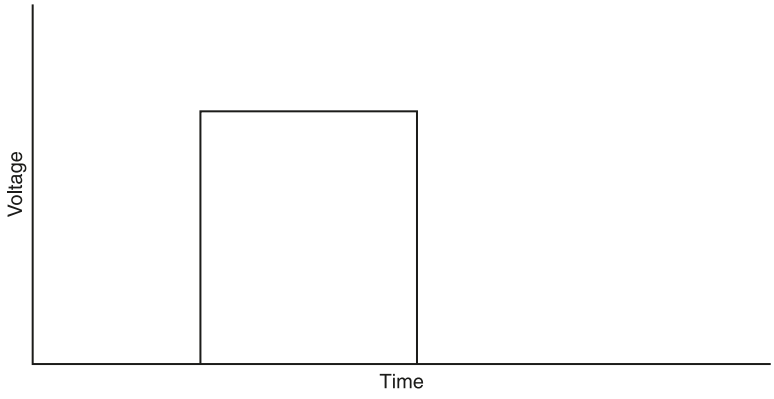

Figure 8.2
Digital sensor readings over time.

Keep these images in your head, as they'll come in handy when we begin to talk about things like threshold setting, edge detection, peak detection, and other sensor-reading methods.

In planning the range of possible microcontroller responses, the first thing you should do is to describe the events that you expect to occur over time. Plot out some of the dynamics that you expect the microcontroller to sense so that you can decide what techniques you'll need to use for your sensors to work optimally. Think through the actions to be sensed, and draw a rough graph of what they should look like. In some cases, this is simple enough that you can do it in your head, but in any complex system, it's often useful to have it on paper.

For example, Figure 8.3 is a rough graph of a person walking down a hallway, past several distance ranging sensors.

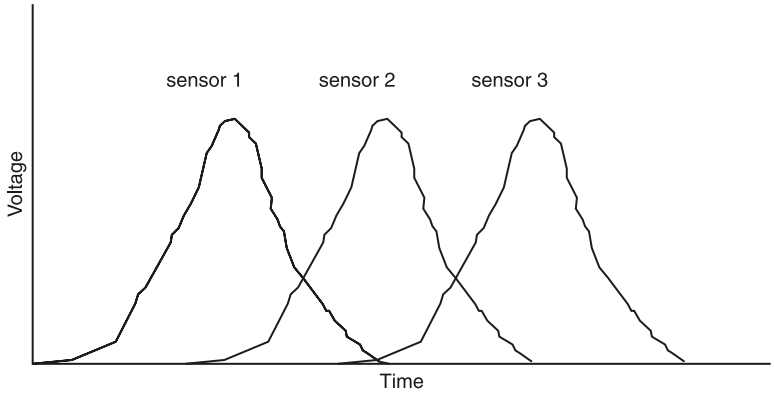

Figure 8.3
A person walking down a hallway, as seen by a microcontroller.

If you just wanted to know where the user is, you could look for the sensor with the highest reading and assume she's near that sensor; but on which side? Checking the readings of the sensors on either side would give you a better picture because the one she's nearer to might read higher than the one she's farther from. If you want her direction, you have to keep track of which sensors have already sensed her presence and which haven't. To get her speed, you could time the delay between sensing a peak on sensor one and sensor two. The possibilities can get very complex, and having a visual model of what you expect to sense can make it much easier to interpret what you actually sense.

Speaking

When we listen, whether we're listening to humans or non-humans, we bring with us similar expectations about how the pattern of a conversation will go. We know that pets, for example, react to our actions in about the same amount of time as people (and in some cases faster). We expect the same from our devices.[2] We expect that when we push a button, flip a switch, wave our hand, shout, or take whatever action is expected, the device will react with at least conversational immediacy. Let's take the Clapper, which can turn on or off a light when it hears a loud sound (such as the clapping of our hands), as an example. If the Clapper reacts too quickly and the lights come on when our hands first touch, we are startled. If the Clapper reacts too slowly, and the lights aren't on by the time we're consciously aware of the end of our clap, we prompt the device again, just as we do in conversation. Think of the number of times you've jiggled the toilet handle, flipped a light switch off and on in rapid succession, or jabbed repeatedly at a remote control power button, and you know what we're talking about.

When you design interactive devices, you have to factor this expectation in, and either meet it or give the person using it a new set of expectations that the device can meet. Your device should respond in ways a person expects or can learn to expect. Once you get to know the pace of response of your devices, you learn to factor that in when you interact with them. If you know the garage door opener takes a second to start moving, you don't jab it again right away. If the fluorescent lights take a few seconds to warm up, you give them time. We're especially tolerant with computers, because we figure they're computing, and we think computing takes time. The truth of the matter is often quite different.

Unlike humans, computers can do only one thing at a time. However, they can do things much faster than us, so it's possible for a computer to have completed several tasks—for example, reading a sensor, interpreting the result, using it to adjust the image onscreen or the position of a motor, and preparing to read again—all before the human that's interacting with it is aware that she's finished speaking. They're so fast, in fact, that multimedia designers often have a tendency to overburden them with complex tasks, making them seem slower than they really are. If you've ever had a computer react sluggishly as you attempt to drag a window across the screen, you've seen this in effect. Each time the computer reads the sensor (the mouse position sensors), it then has to complete several million tasks: figuring the new position of the window; examining what's already drawn there; calculating the effect of fancy things like drop shadows of the window on the images beneath it; redrawing the cursor, the window, and the screen beneath the window's edges; and making a cute dragging sound. One challenge an interaction designer faces is to determine how much the computer can do before the user expects a reaction and to provide that reaction in a timely way.

---

[2] In *The Media Equation: How People Treat Computers, Television, and New Media like Real People and Places* (Cambridge University Press, 1998), Byron Reeves and Clifford Nass make a very interesting case that people unconsciously treat computers like real people.

Expectations and Misunderstandings

Sometimes the problem of interaction is not one of precise timing but of clear indication of expectations.[3] Going back to our conversational model, if you enter into a long and detailed explanation of an idea that causes your listener to go silent for several minutes, you need to continue to pay attention to him in order to gauge his level of interest. You may need to cue him to respond at the end, perhaps through a silent pause and a questioning look. If he wants to interrupt you, you need to pay attention to him as you speak so that you can offer a response. If you want to finish your sentence, you might raise your finger to acknowledge his interruption so that he knows you heard him and will let him respond once you finish.

It's wise to incorporate similar small indicators for feedback into any system to acknowledge user input and to avoid misinterpretation. For example, turning an LED on when a button is pushed and off when the button is released takes negligible processing work, but gives the user a definite sign that the input was "heard" by the computer. The button may start or interrupt many other tasks besides lighting the LED, but lighting the LED lets the user know that those other tasks have been put in motion. Having been acknowledged immediately, the user is prepared to wait for the other tasks to finish as long as the wait is reasonable. If no indication is given, the user might push the button again, triggering unexpected consequences.

In many physical computing applications, the participant is more actively physically engaged than in multimedia computer applications, and her tolerance for delay is lower as a result. The patience that we're willing to give to computers doesn't generally extend to devices where the computer is not visible. Because of this, it's very important to keep system response time as low as possible, on a par with human reaction time or faster at all times.

At other times, you might need to prompt user input. For example, if the user's footsteps trigger a complex series of sound or video that brings her to a standstill, then perhaps a subtle visual cue could be used to prompt her to walk again when it's appropriate. This is particularly true when an output sequence has a subtle ending or does not have a definite ending point.

A narrative description of the participant's experience of your work can be very useful at this point. In describing what she will see, hear, feel, touch, speak to, step on, and so forth, in sequence, or using a branching diagram of possible sequences, you can identify points where her focus on a particular phenomenon is crucial. From there, you can plan ways to get the focus there or keep it there. For example, if the experience depends on the participant hearing and responding to a sequence of tones in a particular order and a particular rhythm, then audio cues to acknowledge her input could be counterproductive. On the other hand, a subtle—and quick—visual acknowledgement could enhance the precision of her responses.

[3] Donald Norman writes about people's mental models of systems and how those models affect the interaction with the systems in *The Design of Everyday Things* (Basic Books, 2002). His explanation goes into more depth on the idea presented here.

These kinds of cues are important because they're the cues on which a person builds her mental model of how a system works. If the system doesn't respond, she may think it's broken. For example, in a system we designed once, a series of buttons triggered playback of a series of video clips on a screen above the buttons. When a given clip was playing, the clients did not want another button press to interrupt the clip. In order to discourage viewers from hitting buttons at will, we put lights in the buttons, and dimmed them when a video clip was playing. When the clip was over, the buttons lit up again. This cue told viewers that the system was working properly, but that pressing a button when a video clip was playing was not a desired behavior. If they pressed a button during a video clip, the button would brighten very slightly while it was pressed, to ensure the viewer that the system was working, but then would fade again when the button was released, to limit distraction from the video. All the feedback in the world will not stop some users from feeling locked out by not being able to change the video. The designer must decide which is worse, interrupting the video or alienating that user.

Avoiding Modal Behavior

Software interaction designers are lucky people. They have the luxury of unlimited real estate on which to design. In software, you can always add another menu item, another pop-up list, or go to another screen when the number of steps to complete a task gets to be too much to fit on one screen. Virtual real estate is infinitely expandable. Physical interfaces lack this luxury because the device can only be so big, the room is only so many steps across, the camera's field of view can take in only so much, or the microcontroller can only read so many switches (at least until you get to Chapter 14, "Managing Multiple Inputs and Outputs"). Because you're combining a software control system with a physical interface, you can add features as long as you've got the program memory to fit it, but you won't have enough hardware sensors to trigger these new functions. You may be tempted to have the same sensors control multiple functions. This is a slippery slope, and utter confusion lies at the bottom.

For example, imagine you're building a musical instrument with a distance sensor as the main sensor. Normally, the sensor controls the pitch of the instrument. But if the user presses a green button, the distance sensor gets shifted from controlling the pitch of a sound to controlling the volume. What if the user wants to modulate the volume and the pitch at the same time? A red button changes the key signature the instrument plays in. But there are lots of key signatures, so she's got to press the button repeatedly to get to the next one. How does she know which key she was in last or which is next? What if she learns the whole sequence of key signatures, then presses the button repeatedly but overshoots the one that she wants. Can she go backwards?

Systems like this, where the behavior of the controls changes depending on the mode the system is in, work against the clarity of the physical interface. They give the user a layer of organization to remember that's not indicated in the controls themselves. One workaround to this is to change the ambient conditions depending on the mode. For example, you might cause the color of the instrument to change from blue to green when the sensor is shifted from pitch control to volume control. This solution still requires the user to memorize the colors of the modes. Another solution is to force the user to maintain contact with the mode-changing control in order to maintain the mode. The sustain pedal on a piano is an

excellent example of this. It's not possible to forget that you're in sustain mode because your foot is strained in order to maintain the mode.

Unless the mode can be clearly indicated with no need for the user to remember what the change means, modes tend to be confusing. A better solution is to avoid modal systems whenever possible by mapping each control to a unique and consistent response.

Complex Responses

We have been proposing a kind of straightforward interaction scheme that may sound a little too dry and methodical if you're planning an ambient musical installation or a responsive sculpture that changes subtly over time. However, these are precisely the kinds of physical computing projects that need this kind of planning the most. Humans love patterns. We automatically look for patterns in everything from clouds to tea leaves to cracks in the sidewalk. In interactive systems, we look for patterns in the connections between the inputs and the outputs. The more abstract or subtle those connections are, the harder it will be for a viewer or participant to find a pattern. Sometimes the connections are too complex, or they're based on a set of conventions common only to you and other people with your training and experience. After the user gives up on finding the pattern, she might feel bored by a seemingly random sequence of events. She might even feel angry that you were implicitly inviting her to find a pattern where none apparently exists. This is also true in performance; the classic example of this is the "laptop music performance," in which several people sit onstage typing on laptops while the audience hears music, apparently generated by the performers. The audience may not appreciate the music as much if they cannot find patterns between the actions of the performers and the music coming out. Musical performance is often more satisfying when there's an apparent connection between the gestures of the performer and the sounds heard.

When you're planning interactive systems, no matter how abstract, keep in mind the tendency to look for patterns. You should give the viewer or participant a general sense of the connection between the inputs and outputs. To give your project a more complex feel, consider using multiple inputs. Let each input have a straightforward response, but design the system so that they combine in complex ways. Think about the ripples produced by throwing stones in a pond. The response to each stone is simple and predictable, yet the pattern generated by throwing in several is rich, layered, and unpredictable.

In addition to clear reactions, clear interfaces are also essential. If you choose to hide or disguise the sensors so that the person doesn't know how she is triggering this behavior, you might as well play a pre-recorded sequence because she won't understand how she is a part of the system. This doesn't mean, for example, that she needs to see a large red footprint labeled "stomp here" if you're sensing her footsteps. However, if her footsteps are what your system is responding to, then she should be able to see a connection between a single footstep and the system's response when the system is at a state of rest. You may want to design a system in which it takes a thousand footsteps in order to elicit a response, but she'll never get to the thousandth step if she's not clued in early on that "the game is afoot." In general, it's always best to err on the side of simple responses to simple actions, letting the combination of simple responses generate a complex pattern.

Random Numbers

We have been warning against interactions that appear to the user to produce random results. On the other hand, noise or random variation is everywhere in our lives. The movement of whiskers on your robotic cat or the flickering of stars on your ceiling would feel wrong without some random variation. The frustration of random responses is often exactly what you want in a game or simulation. For example, a roll of the dice or the roulette wheel, shuffled deck, or a phonebook flipped open to a random page are all events that rely on the randomness of the physical world to introduce surprise and excitement. While these random patterns are ubiquitous in life, the computer has a hard time generating random events. The beauty and utility of random numbers is not that they are random, but that they spring from patterns too complex to identify immediately. When these patterns come from mathematical algorithms divorced from physical reality, they're not always very convincing. Instead of generating randomness with algorithms, consider tapping into the noisy patterns that we see all around us in our physical environment for random numbers.

Every time you think you need a random number, ask yourself if there is a way to get that number from the noise in the physical system you are building. The standard trick on a multimedia computer is to use the milliseconds since the user last touched the keyboard, because unlike computers, people are unpredictable in their actions, touching the keyboard at erratic intervals. With the microcontroller, you have many more opportunities for sensing noise in the world. One simple example is to use the random static electricity generated by people, radio waves, and so on. Try this: stick a bare wire on an analog input of your microprocessor. Make the wire a long one, say, a foot or more. Strip off the insulation. Coil it if you want. Don't attach the other end to anything. Program the chip to read the wire as an analog input. As it's running, touch the wire. The changes appear to be random, and you should see a marked difference, but still a random pattern, when you touch it. This is because you're reading micro-voltage changes due to anything in the room that generates an electrical signal: you, a TV monitor, a cell phone, and so on. The bare wire is an antenna for your chip. Use its results as a random number. You could do the same thing with a pressure sensor or a tilt sensor in environments where pressure and tilt are changing.

These examples take advantage of complex processes in the physical world and use them as design elements. They allow you to include the unexpected but also to control it. A truly random system can often be very frustrating for the person using it, whereas a complex but ordered system is more engaging. It gives the person using it a structure on which to model her idea of how it works. Ultimately, she has a better chance of mastering such a system, and therefore finding it more satisfying to work with.

Your mindset as programmer will really benefit from including the possibilities and constraints of the physical world in your work. Without physical computing, everything that the user interacts with is abstracted. You have to simulate buttons and handles, movement, and orientation. With physical computing, you factor all of your user's physical understanding of the physical world into your interface. You get to take advantage of the fact that she knows how buttons and springs and doors and all kinds of other physical things work.

Thinking

Even though computers can do only one thing at a time, it only takes a few microseconds to execute each line of code. You can use this to your advantage by programming the computer to listen to its inputs for short periods, then control its output for short periods. If all of these periods are shorter than the human reaction time, it can appear to a human being as if the computer is doing many things, like speaking and listening at the same time.

Sensing and reacting in momentary fragments necessitates a good bit of work if the computer is going to sense events and respond at a human time scale. In order for the computer to be able to interpret events, it has to reconstruct events from the fragments it senses each time it listens to its sensors. To interpret a shout as a cue to play a song, for example, it must first have sensed silence, and be able to tell how much noise is different enough from silence to interpret as a shout. If two shouts constitute a cue to play music, it's got to hear both the shouts, and the silence in between, and be able to count.

In a similar fashion, a computer has to react in ways that a human can interpret. If its output happens in small bursts, it has to create longer responses from a sequence of small bursts. If it's meant to play a song all the way through when cued, and not respond to cues while playing, it's got to be able not only to start playing the music, but also to keep track of when it ends, as well as to offer an alternate response if it senses a "play" cue while the music is playing. Though this may seem like a lot of work, it's not too bad when you understand a few of the techniques below.

Techniques for Effective Interaction

Following are some examples that put the principles described above into action. These examples give you ways to make your microcontroller handle multiple tasks simultaneously and recognize the beginnings and ends of complex events based on what it senses.

Multitasking

Imagine you want to make an LED flash three times in response to the push of a button. You want the flashing to happen once per second. Here's the pseudocode for it:

```
read button
if button is pushed then
    loop 3 times:
        turn on LED
        pause 1 second
        turn off LED
        pause 1 second
    end loop
end if
```

If you were to write this routine using pause commands (or delay commands on the BX-24), there would be six seconds in between the time when the button is pushed and the next reading. If you want to make the LED react to a button push in the middle of the flashing sequence, you're out of luck. In the middle of the flashing loop, the microcontroller isn't reading the button at all.

You could find a flashing LED processor to flash the LED for you, attach that to your microcontroller, and turn it on and off in response to the sensor, but that would be more technology than the project needs. In this case, it would be better to write a more responsive program so that there's not a six-second pause between readings of the sensor.

You need a way to make sure that you're listening to the user at a rate faster than human response time, while still controlling the blinking LED at a rate you want. Ideally, you would have two different loops (or threads): one that runs very fast, like the main loop, to keep checking the sensor, and one a bit slower, to blink the LED if necessary. This way, if something changes with the sensor while the LED is blinking, you can provide an appropriate response. One way to do this is by using counters to count the number of times through the main loop. Let's assume the main loop will run in a thousand times a second (in most cases, it's faster than that, but this makes for easy math). You can use the counter to listen to the sensor every loop and change the LED every thousandth loop.

Here's an example:

```
Set counterForLEDs = 0
Set LEDFlashCounter = 0
Set flag = down
Loop:
    Check the switch
    If the switch is pressed then
        Put up a flag to tell the LEDs to flash
    end if

    If the flag is up then
        Subtract one from the counterForLEDs
        If the counterForLED has reached zero
            Reset counterForLEDs to 1000
            If we have not flashed three times
                Add one to the flash counter
                Flash
            Else
                You have finished flashing so
                Reset flash counter
                Reset counterForLEDs
                Reset the flag
            End if
        End if
    End if
End loop
```

You may be confused by the flag in the pseudocode above; so far, we haven't talked about making computers control flags. A *flag* is just a variable used to send a message from one loop to another. Think of it like a semaphore flag, or the flag on the mailbox that you put up to tell the mail carrier that there's outgoing mail inside. In the example above, it's possible that the user might stop pressing the switch before the LED stops flashing, so you can't rely on just the state of the switch to know whether the LED should flash. The flag gets set to 1 every time you're supposed to start flashing the LED and set to 0 when you've successfully flashed the LED. Using loop counters and flags like this, you can set one process in motion at its own pace (the flashing of the LEDs), while keeping the previous process in motion at another pace (checking the sensor).

Here's an example in code:

PBASIC
MBasic

```
counterVar VAR Word   ' we want to count to 1000, so we need a word
needFlashingFlagVar VAR  Bit
timesFlashedVar VAR  Byte
ledState VAR Bit

INPUT 7      ' the Switch is on this pin
LOW 8            ' the LED is on this pin
counterVar = 1 'set the countdown close to zero
Main:
    'check every loop for the button press
    IF IN7 = 1 THEN
        needFlashingFlagVar = 1 'set the flag
    ENDIF

    IF needFlashingFlagVar THEN
        counterVar = counterVar - 1 'count down to next change
        IF counterVar = 0   THEN 'if you have counted down to zero
            counterVar = 1000 'set the counter back up
            IF timesFlashedVar < 3 THEN ' if you have not done all the flashes
                IF ledState = 0 THEN 'flash the led
                    timesFlashedVar = timesFlashedVar + 1
                    HIGH 8
                    ledState = 1
                ELSE
                    LOW 8
                    ledState = 0
                ENDIF
            ELSE
                'if you have finished flashing the led set variables to initial state
                timesFlashedVar = 0
                needFlashingFlagVar = 0
                ledState = 0
                counterVar = 1
                LOW 8
```

```
            ENDIF
        ENDIF
    ENDIF
GOTO main
```

PicBasic Pro

```
counterVar VAR Word   ' we want to count to 1000, so we need a word
needFlashingFlagVar VAR  Bit
timesFlashedVar VAR  Byte
ledState VAR Bit

Input portb.0         ' the switch is on this pin
Output portb.1        ' the LED is on this pin

Low portb.1
counterVar = 1 'set the countdown close to zero
Main:
    'check every loop for the button press
    IF portb.0  = 1 THEN
        needFlashingFlagVar = 1 'set the flag
    ENDIF

    IF needFlashingFlagVar THEN
        counterVar = counterVar - 1 'count down to next change
        IF counterVar = 0   THEN 'if you have counted down to zero
            counterVar = 1000 'set the counter back up
            IF timesFlashedVar < 3 THEN
                ' if you have not done all the flashes
                IF ledState = 0 THEN 'flash the led
                    timesFlashedVar = timesFlashedVar + 1
                    High portb.1
                    ledState = 1
                ELSE
                    Low portb.1
                    ledState = 0
                ENDIF
            ELSE
            'if you have finished flashing the led set variables to initial state
                timesFlashedVar = 0
                needFlashingFlagVar = 0
                ledState = 0
                counterVar = 1
                LOW portb.1
            ENDIF
        ENDIF
    ENDIF
GOTO main
```

BX-Basic

```
DIM counterVar as Integer   ' we want to count to 1000, so we need a word
DIM needFlashingFlagVar as  Byte
DIM timesFlashedVar as  Byte
DIM ledState as Byte

Sub Main()

    Call putPin(13,0)
    counterVar = 1 'set the countdown close to zero
    Do
    'check every loop for the button press
    IF getPin(12) = 1 THEN
        needFlashingFlagVar = 1 'set the flag
    END IF

    IF (needFlashingFlagVar = 1) THEN
        counterVar = counterVar - 1 'count down to next change
        IF counterVar = 0   THEN 'if you have counted down to zero
            counterVar = 1000 'set the counter back up
            IF timesFlashedVar < 3 THEN
            ' if you have not done all the flashes
                IF ledState = 0 THEN 'flash the led
                    timesFlashedVar = timesFlashedVar + 1
                    call putPin(13,1)
                    ledState = 1
                ELSE
                    call putPin(13,0)
                    ledState = 0
                END IF
            ELSE
            'if you have finished flashing the led set variables to initial state
                timesFlashedVar = 0
                needFlashingFlagVar = 0
                ledState = 0
                counterVar = 1
                call putPin(13,0)
            END IF
        END IF
    END IF
loop
End Sub
```

Keeping track of the timing of processes like this doesn't do much good, though, if the processes don't interact in some useful way. For example, what happens if the user presses the switch again while the LED flashing sequence is happening? Currently, nothing, which makes the program the same as if you just used pauses. If you want a second switch

press to interrupt the flashing sequence and bring it to an end, you need a little more information. You'd first need to watch changes (see the "Edge Detection" section below) in the switch's state, so you only get one event for each press of the switch. When you detect a press you would toggle the current state of the flag and counters.

Notice that the output routine happens relatively infrequently (every 1,000th loop), while your input routine happens very frequently (every loop). Generally, you want to place a priority on listening frequently so that you can change your response quickly.

You have to factor in the time taken to run the rest of our program as well, so you might have to make your pause smaller or change your counting. Perhaps you'd change to activating the LED process every 500th loop or every 100th loop. There's usually not an easy way to calculate this in advance, so you end up starting with an arbitrary value that you think is close enough, then changing the program until you find a value that works.

Certain commands that you give a processor will take more time than others. For example, any command that sends bytes out serially will take as long as needed to send the bytes (9600 bits per second). Even the if-then statements in the example above slow the processor down somewhat. The surest way to find out how a given command affects your program is to try it in practice. Your interaction with the user and with your output will be helped by removing any unnecessary commands, so when you've learned what you need from a debug statement, for example, you should comment it out or remove it.

This timing loop is not the only way to balance timing between input and output. There are as many schemes for this as there are programmers, and everyone has their own method. The key factor to keep in mind is that any time the processor is constrained to one task, like a pause, or a print statement, or any other command that takes time, then the system is not listening to the user. Whenever this happens, you must have a way of letting the user know and when it's appropriate for them to respond again.

Some processors have a more advanced operating system, and can handle the precise timing of multiple tasks for you. For example, the BX-24 has limited multitasking capability. (For details, see the BX-24 documentation.) Not many small microprocessors have such a capability built in, so you often have to find your own ways of handling it, like we've done above. The BX-24's multitasking is based on the idea of *timer interrupts*. Interrupts are basically routines that are scheduled to run every time the processor's timer counts off a certain interval, no matter what. On other microcontrollers, interrupts are somewhat more complex to use and involve knowing some lower-level programming. Besides timer interrupts, there are also *hardware interrupts*, which are input pins that are designed to stop the flow of a program when they receive an input, no matter what's going on in the code. Interrupts are beyond the scope of these notes, but see the documentation of your microprocessor to see if it supports interrupts and how to use them.

Another approach to the multitasking problem is to have separate processors handle the separate tasks. This could involve things like using a separate processor to drive a motor in a moving sculpture, letting a MIDI sampler handle the playback of music in a sound installation, or letting a desktop computer control a complex visual display while a microcontroller reads the input sensors. You're processing the task of interaction in parallel: each part of the job is given to an individual processor. All the processors talk to each other only when needed, such as when an event occurs that another processor

needs to handle. Their attention to each other is minimal relative to the attention they give to their tasks. Because each task is simple, each processor can do it quickly, so overall response time of the system is within the range of human expectations.

Dividing the task among several processing devices is a higher-level solution that saves you the trouble of having to coordinate tasks at the programming level on one processor. It comes at a cost, however. First, there's the cost of equipment. Then there's the cost of time in figuring out the connections between components. Then there's the cost of coordinating what has become a complex system of many components. You always have to balance these costs against the costs of building your own programming and electronics. Even when you do use several processors, you still have to manage when they're listening to their sensors or controlling their outputs, and when they're passing that information along to each other. If it's easier to use several devices, and the financial costs aren't too extravagant, then that's the way to go. For example, we often resort to using serially controlled servo motor controllers whenever we're building projects using multiple servos. The mini-SSC from Scott Edwards Electronics (http://www.seetron.com) is an excellent example. On the other hand, when the cost becomes prohibitive or the complexity of the system gets out of hand, then the best solution is to manage everything on one central processor.

Edge Detection

Let's take the simplest possible event you might want to sense. A person presses a button, and you want to know when she pressed and when she released. You already know how to read a digital input continually. Here's the pseudocode:

```
Loop:
    If input is high then
        Print "Input is high"
    Else
        Print "Input is low"
    End if
End loop
```

If you've written a program like this, you know that what you get is something like this:

```
Input is low Input is low Input is low Input is low Input is high Input is high
Input is high Input is high Input is high Input is high Input is high Input is
high Input is low Input is low Input is low Input is low Input is low Input is
low Input is low Input is low Input is lo Input is low
```

Say you want to count the number of times a button (digital sensor) is pressed and turn on an LED when it's been pressed three times. You might be tempted to do something like this:

```
Set ButtonCounter to 0

loop:
    If the button is pressed then
        Add one to the ButtonCounter
```

```
        End if

        If the buttonCounterVar > 3 then
            Light the LED
            'reset the buttonCounterVar to 0
        end if
    end loop
```

However, this loop gets run several thousand times a second on a microcontroller. If it takes a human being a tenth of a second to take her finger off the button once she's pressed it, then the loop will get run hundreds of times during what she perceives as one press of the button. That's not a reliable way to count button presses at all!

To get around this problem, you need to consider not only the state of the button, but also the state of the button during last time the loop was run. If the button is high and the last state of the button was low, then you know that the person just began pressing the button. If the button is low and the last state is high, then you know she just stopped pressing the button. This is sometimes called *edge detection* because you're finding the beginning and ending edges of the button press (refer to Figure 8.2). By counting the ending edges, you know how many times the button's been pressed. For those readers used to multimedia programming in Lingo, Flash, or other GUI environments, this is the equivalent to a `mouseUp` or `mouseDown` event.

The steps used in edge detection are simple:

1. Read an input.
2. Compare it to the last reading.
3. Take action based on the comparison.
4. Store the current reading as the last reading so you can take a new one.

Here's a program that uses edge detection and properly counts button presses:

PBASIC

```
ButtonStateVar var byte
LastButtonStateVar var byte
ButtonCountVar var byte
Input 7  ' the button is on pin 7

main:
    ButtonStateVar = in7
    ' if the button isn't the same as it was last time through
    ' the main loop, then you want to do something:

    if buttonStateVar <> lastButtonStateVar then
        if buttonStateVar = 1 then
            ' the button went from off to on
            ButtonCountVar = ButtonCountVar + 1
            debug "Button is pressed.", 10, 13
        else
```

```
            ' the button went from on to off
            debug "Button is not pressed", 10, 13
            debug "Button hits: ", DEC buttonCountVar, 10, 13
        endif

        ' store the state of the button for next check:
        lastButtonStateVar = buttonStateVar
    endif
goto main
```

MBasic

```
ButtonStateVar var byte
LastButtonStateVar var byte
ButtonCountVar var byte
Input 7  ' the button is on pin 7

main:
    ButtonStateVar = in7
    ' if the button isn't the same as it was last time through
    ' the main loop, then you want to do something:

    if buttonStateVar <> lastButtonStateVar then
        if buttonStateVar = 1 then
            ' the button went from off to on
            ButtonCountVar = ButtonCountVar + 1
            debug ["Button is pressed.", 10, 13]
        else
            ' the button went from on to off
            debug ["Button is not pressed", 10, 13]
            debug ["Button hits: ", DEC buttonCountVar, 10, 13]
        endif

        ' store the state of the button for next check:
        lastButtonStateVar = buttonStateVar
    endif
goto main
```

PicBasic Pro

```
ButtonStateVar var byte
LastButtonStateVar var byte
ButtonCountVar var byte
Input portb.7     ' the switch is on RB7
ButtonStateVar = 0

main:
    ButtonStateVar = portb.7
    ' if the button isn't the same as it was last time through
```

```
    ' the main loop, then you want to do something:

    if buttonStateVar <> lastButtonStateVar then
        if buttonStateVar = 1 then
            ' the button went from off to on
            ButtonCountVar = ButtonCountVar + 1
            serout2 portc.6, 16468, ["Button is pressed.", 10, 13]
        else
            ' the button went from on to off
            serout2 portc.6, 16468, ["Button is not pressed", 10, 13]
            serout2 portc.6, 16468, ["Button hits: ", DEC buttonCountVar, 10, 13]
        endif

' store the state of the button for next check:
        lastButtonStateVar = buttonStateVar
    endif
goto main
```

BX-Basic

```
Dim ButtonStateVar as byte
Dim LastButtonStateVar as byte
Dim ButtonCountVar as byte
' the button is on pin 12

Sub main()
    do
        ButtonStateVar = getPin(12)
        ' if the button isn't the same as it was last time through
        ' the main loop, then you want to do something:

        if buttonStateVar <> lastButtonStateVar then
            if buttonStateVar = 1 then
                ' the button went from off to on
                ButtonCountVar = ButtonCountVar + 1
                Debug.print "Button is pressed."
            else
                ' the button went from on to off
                debug.print "Button is not pressed"
                debug.print "Button hits: "; Cstr( buttonCountVar)
            end if

            ' store the state of the button for next check:
            lastButtonStateVar = buttonStateVar
        end if
    loop
end sub
```

The basic four steps of edge detection come up constantly when designing any kind of interactive system. Edge detection routines enable the microcontroller to interpret changes in sensor readings as higher-level actions. You'll use these steps all the time, whether you're reading digital or analog sensors.

Analog Sensors: Thresholds, Edges, and Peaks

When you're working with analog sensors, the beginnings and endings of the events you might sense are slightly different than for digital sensors. We will talk about three kinds of changes that you might look for: thresholds, edges, and peaks.

Finding Thresholds in an Analog Signal

Sometimes you only care whether your sensor has passed a threshold. For example, if you're working with a photocell, and you want to sense a flashlight falling on the photocell but not the ambient light in the room, you might write a routine to filter out all readings below a certain threshold. Testing whether or not your reading is above or below a threshold essentially turns your analog sensor into a digital sensor.

You could detect a threshold with a routine like the following:

```
establish threshold from a reading of ambient conditions at start up
Loop:
Read sensor
If sensor reading is higher than threshold then
        React
End if
End loop
```

This code is simple but it all depends on having a good number for the threshold. If you are in a controlled setting, you may be able to use a fixed number for your threshold. But if your sensor were a photocell, for example, this threshold might work in the morning but not at night. You might need to calibrate the threshold to different ambient conditions. One quick way of doing this it to grab a sensor reading before you enter your main loop and use that for the threshold. You would just have to make sure that your microcontroller is powered up under normal ambient conditions to get a good threshold and restart the chip to recalibrate the threshold. Another approach is to continually read a *control* sensor that is situated in an area away from the area where you expect change. The reading from the control sensor gives you the current baseline of ambient conditions and is used to set the threshold for the sensor that is actually changing. For thresholds that automatically recalibrate over time, you will need to average your readings over time. See the section called "Smoothing, Sampling, and Averaging" for more on this.

Finding Edges in an Analog Signal

If you think about the threshold as an edge, the process of sensing when the sensor is activated is similar to the digital sensor example above. You take a reading, determine if it's above your threshold, and if the previous reading was below the threshold, the sensor has just been triggered. If the reading is below the threshold and the previous reading was above, then the sensor has just ceased to be triggered.

This is a very simple routine. It's much like the digital input reading routine before you added edge detection. A slightly more refined routine might incorporate an analog version of edge detection, like so:

```
Loop:
    Read sensor
    If sensor reading is higher than threshold then
        If previous sensor reading was below threshold then
            React
        End if
    End if
    Save the current sensor reading as the last reading
End loop
```

Since the logic of analog edge detection is the same as it is for digital edge detection, we'll leave it to you to write your own actual code for this.

You might find that find your readings are fluctuating slightly above and then below your threshold, giving you many apparent edges when you are really being still. For example, in the photocell graph in Figure 8.4, you can see lots of ripples in the edge of the curve. These could appear as multiple threshold crossings when in reality there's only one crossing that you care about. You can use some of the techniques in the section called "Smoothing, Sampling and Averaging" to reduce this problem.

Finding Peaks in an Analog Signal

Analog threshold detection is great for testing when an analog sensor crosses a threshold when it's rising or falling. Sometimes, however, you want to know when it's reached a peak and is headed back down again. For example, imagine the sensor in the graph on the right in Figure 8.4 is the sensor for a key on a piano keyboard. To know how loud the note is to be played, you would want to know the peak, which tells you how hard the key was hit. In this case, a slight variation on the edge detection routine would do the trick. Instead of an edge, you have a peak, where the sensor reaches its maximum value. In order to find the peak, you look for the sensor's reading to cross the threshold going up, then wait until it reaches a maximum and take that maximum as the peak.

In order to determine a peak, you have to set a threshold for when you start and stop looking for the peak. Figure 8.4 shows readings from two different sensors. The figure on the top shows what happens when you cover a photocell with your hand quickly. The figure on the bottom shows several taps on a force-sensitive resistor (FSR). You can see that the taps on the FSR happen much quicker than the covering of the photocell. There's a clear peak on each tap on the FSR. The readings start at zero (which is the threshold where you start and end looking for the peak) and end at zero. In the photocell graph on the left there is a peak, but the beginning and ending point is not as clear, so you need to set a threshold to determine which readings you care about and which you don't. With the photocell, you might only care when the light crosses the threshold, because the change is so gradual. On the other hand, because the change in the FSR readings is sudden, you might need to look for the peak.

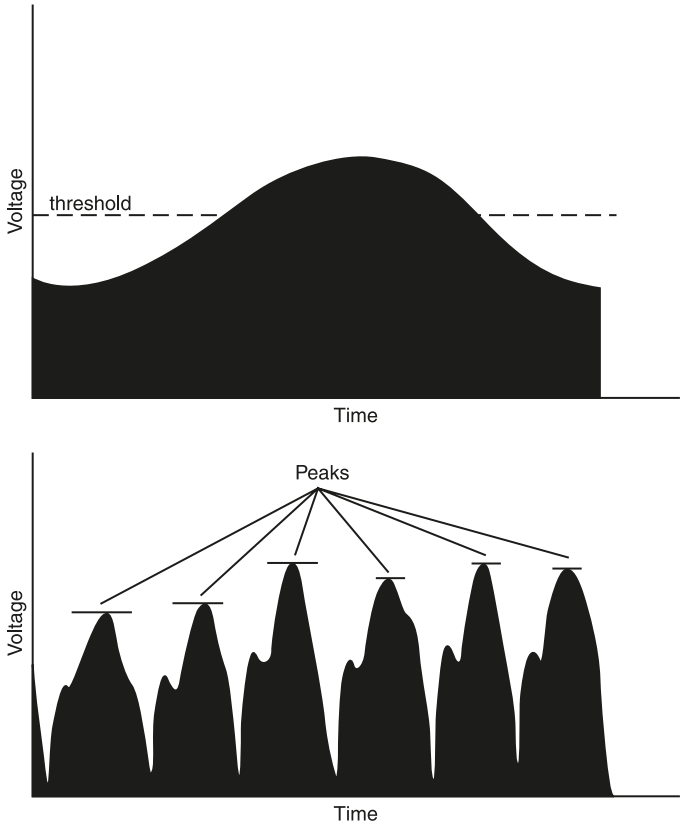

Figure 8.4
Analog sensor readings. On the top, a photocell being covered and uncovered quickly. On the bottom, several taps on a force-sensitive resistor.

In pseudocode, finding a peak looks like this:

```
Loop:
    Read sensor
    If sensorValue >= threshold then
        If sensorValue >= lastSensorValue then
            PeakValue = sensorValue
        End if
    Else
        If peakValue >= threshold then
            this is the final peak value; take action
        end if
        Set the peak value to 0 so we can start it rising next loop
    End if
End loop
```

You might have to set your threshold fairly high in order to eliminate false peaks like the ones you see at the beginning of each reading in the graph on the bottom in Figure 8.4.

Peak finding is useful when your sensor changes very fast, like the key on the keyboard mentioned above, but it's not as useful for sensors that change slowly, like a volume knob. Before you implement a peak-finding routine, decide whether your sensor lends itself to finding peaks first.

Here's the actual code:

PBASIC

```
PeakValue VAR Word
SensorValue VAR Word
LastSensorValue VAR Word
Threshold VAR Word
Noise var word

Threshold = 50      ' set your own value based on your sensors
PeakValue = 0     ' initialize peakValue
Noise = 5

Main:
    ' read sensor on pin 0:
    HIGH 0
    PAUSE 1
    RCTIME 0, 1, sensorValue

    ' check to see that it's above the threshold:
    IF sensorValue >= threshold + noise THEN
        ' if it's greater than the last reading,
        ' then make it our current peak:
        IF sensorValue >= lastSensorValue + noise THEN
            PeakValue = sensorValue
        ENDIF
    ELSE
        IF peakValue >= threshold THEN
            ' this is the final peak value; take action
            DEBUG "peak reading", DEC peakValue, 10, 13
        ENDIF

        ' reset peakValue, since we've finished with this peak:
        peakValue = 0
    ENDIF

    ' store the current sensor value for the next loop:
    lastSensorValue = sensorValue
GOTO main
```

MBasic

```
PeakValue var word
SensorValue var word
```

```
LastSensorValue var word
Threshold var word
noise var word

Threshold = 50    ' set your own value based on your sensors
Noise = 7
PeakValue = 0     ' initialize peakValue
Main:
    ' read sensor on pin 0:
    ADin 0, sensorValue

    ' check to see that it's above the threshold:
    If sensorValue >= threshold + Noise then
        ' if it's greater than the last reading,
        ' then make it our current peak:
        If sensorValue >= lastSensorValue + noise then
            PeakValue = sensorValue
        endif
    Else
        If peakValue >= threshold then
            ' this is the final peak value; take action
            debug ["peak reading", DEC peakValue, 10, 13]
        endif

        ' reset peakValue, since we've finished with this peak:
        peakValue = 0
    Endif

    ' store the current sensor value for the next loop:
    lastSensorValue = sensorValue
Goto main
```

PicBasic Pro

```
' Define ADCIN parameters
DEFINE  ADC_BITS        10            ' Set number of bits in result
DEFINE  ADC_CLOCK       3          ' Set clock source (3=rc)
DEFINE  ADC_SAMPLEUS    20         ' Set sampling time in uS

PeakValue var word
SensorValue var word
LastSensorValue var word
Threshold var word
Noise var word

Threshold = 50    ' set your own value based on your sensors
PeakValue = 0     ' initialize peakValue
```

```
noise = 5

' Set PORTA to all input
TRISA = %11111111
' Set up ADCON1
ADCON1 = %10000010

Main:
    ' read sensor on pin RA0:
    ADCin 0, sensorValue

    ' check to see that it's above the threshold:
    If sensorValue >= threshold + noise then
        ' if it's greater than the last reading,
        ' then make it our current peak:
        If sensorValue >= lastSensorValue + Noise then
            PeakValue = sensorValue
        endif
    Else
        If peakValue >= threshold then
            ' this is the final peak value; take action
            serout2 portc.6, 16468, [ "peak reading", DEC peakValue, 13,10]
        endif

        ' reset peakValue, since we've finished with this peak:
        peakValue = 0
    Endif

    ' store the current sensor value for the next loop:
    lastSensorValue = sensorValue
Goto main
```

BX-Basic

```
Dim PeakValue as integer
dim noise as integer
Dim SensorValue as integer
Dim LastSensorValue as integer
Dim Threshold as integer

Sub Main()
Threshold = 300       ' set your own value based on your sensors
noise = 7
PeakValue = 0     ' initialize peakValue

    do
        ' read sensor on pin 13:
```

```
        sensorValue = getADC(13)
        'debug.print cstr(sensorValue)
        ' check to see that it's above the threshold:
        If sensorValue >= threshold + Noise then

            ' if it's greater than the last reading,
            ' then make it our current peak:
            If sensorValue >= lastSensorValue + noise then
                PeakValue = sensorValue
            End if
        Else
            If peakValue >= threshold then
                ' this is the final peak value; take action
                debug.print "peak reading: "; cStr(peakValue)
            end if

            ' reset peakValue, since we've finished with this peak:
            peakValue = 0
        End if

        ' store the current sensor value for the next loop:
        lastSensorValue = sensorValue
    loop
end sub
```

Debouncing

If you were using a homemade switch, or any switch with loose contacts, you may have noticed when you ran the edge detection routine that every once in a while, the button count advanced by more than one when you pressed the button. What happened?

If you've ever been shocked by static electricity jumping from your hand to a metal doorknob on a cold, dry day, you know that it's possible for electricity to jump through air when two conductors are close to each other and when the charge is great enough. At the lowest physical level, a switch is just two mechanical contacts that can be brought into contact with each other or separated. When the contacts are brought close together, but not touching, the same phenomenon can occur. For the last few fractions of a second before the contacts make a solid connection, they bounce off each other and grind together. It's possible for them to make and break electrical contact several times in a few milliseconds, and current tries to leap the gap, sometimes succeeding and sometimes failing. Because the microcontroller is reading the switch several thousand times a second or more in our applications, it's not uncommon for it to detect these false contacts before the switch is properly closed. The microcontroller might read the press of a button something like in Figure 8.5.

During the time between the arrows, the switch is being closed. This may be less than a millisecond long, but the microcontroller can still take several readings during that time.

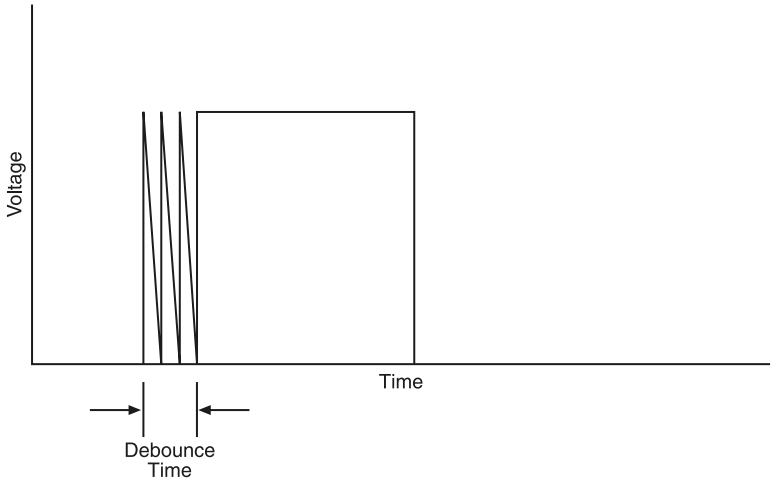

Figure 8.5
Readings from a switch on a time scale of milliseconds or microseconds.

This can mean that sometimes, even when you're using an edge detection routine like the one above, you might still get false readings. To prevent this, you can use a debounce routine. *Debouncing* a switch is the process of checking its reading two or more times over a very short interval (less than human reaction time) to ensure an accurate reading. It works like this:

```
If the switch is on
    Wait a fraction of a second
    If the switch is still on then
        Take action
    End if
End if
```

Here's a simple debounce routine:

PBASIC
MBasic

```
SwitchOnVar var byte
Input 7

Main:
    If in7 = 1 then
        Pause 10 ' 10 milliseconds; change the time to suit your needs
        If in7 = 1 then
            SwitchOnVar = 1
        Else
            SwitchOnVar = 0
        Endif
    Endif
Goto main
```

PicBasic Pro

```
SwitchOnVar var byte
Input portb.7

Main:
    If portb.7 = 1 then
        Pause 10 ' 10 milliseconds; change the time to suit your needs
        If portb.7 = 1 then
            SwitchOnVar = 1
        Else
            SwitchOnVar = 0
        Endif
    Endif
Goto main
```

BX-Basic

```
Dim SwitchOnVar as byte

Sub Main()
    do
        If getPin(12) = 1 then
            Call delay(0.01)' 10 ms; change the time to suit your needs
            If getPin(12) = 1 then
                SwitchOnVar = 1
            Else
                SwitchOnVar = 0
            End if
        End if
    Loop
End sub
```

Debounce routines can be used with analog sensors as well, though it's more common to check an analog reading to see if it's higher or lower than a given threshold or within an acceptable range than to check if it's identical to its past reading. Unlike digital sensors, which can have only two states (on or off), analog sensors can have multiple states. Sometimes the difference between one state and another may be imperceptible to humans, but readable by microprocessors.

Smoothing, Sampling, and Averaging

Much of what you do in programming physical computing projects is to figure out how to deal with the world's natural randomness and make it look smooth. A photoresistor read through an analog-to-digital converter, for example, will never give you a steady value. It always fluctuates with tiny changes in lighting that your eye automatically filters out. Because the microcontroller will read these changes that you don't see,[4] the changes get reflected in

[4] For more on this, see Donald Hoffman's *Visual Intelligence: How We Create What We See* (W.W.Norton & Co., 1998) or Tor Nørretranders' *The User Illusion* (Viking, 1999).

the output. Your sensors and their supporting circuitry will also introduce noise that is not actually in the environment. In practice, you often need to smooth out your readings.

Whenever you start to work with a new sensor, you should get in the habit of testing the sensor with a simple piece of code that just reads it and returns the values continually so that you can watch its behavior over time. The simple digital and analog input programs from Chapter 6 do this job nicely.

A quick way to smooth out your numbers is simply to divide each reading by the amount of fluctuation you typically see. For instance, if your numbers fluctuate within a range of five points when the sensor is at rest, just divide by 5 and the numbers will appear to be still as well. This will give you a smaller range of numbers, however, so your smoothness comes at the expense of resolution. By reducing the resolution to the minimum needed to begin with, you can reduce the jitter you get from noisy readings before it's a problem. For example, if your instrument only plays 12 notes, try dividing your analog range down to 1–12 right away. The process of scaling down your numbers was covered in Chapter 6.

Taking a number of recent readings into account is another approach that can help smooth your results. To do this, you keep an array of recent readings and have a variable to keep track of the last reading that you put into the array. Averaging the recent readings will keep your resolution high and will appear to smooth things out as long as the readings are fluctuating within a limited range.

```
make an array for recent readings
loop
    read sensor
    find the next place in the array
    if you are at the end of the array go the beginning
    insert the current reading to the array, replacing the oldest one
    total all the readings together in a repeat loop
    divide the total of all recent readings by the number of recent readings

end loop
```

If the fluctuations are caused by random noise, the best way to eliminate those freak readings is to take the median (the middle of a sorted set) rather than the average of recent readings. In this way, freakish readings are discarded without affecting normal readings as much. To do this, you have to sort the array of past readings and pick the middle item.

```
make an array for recent readings
loop
    read sensor
    find the next place in the array
    if you are at the end of the array go the beginning
    insert the current reading to the array, replacing the oldest one
    sort all the readings into a separate array by value
    take the middle reading of the sorted array

end loop
```

Any scheme that takes recent readings into account will make your system slower to respond because past readings are still weighing down the average as your sensor heads in a new direction.

Here's the actual code for both averaging and median filtering. Note that this code uses two big arrays, which take up a lot of memory, particularly for the Basic Stamp.

PBASIC

```
lastPositionInArray CON 2 ' keep a history of the past 3 readings
temp VAR Word
average VAR Word
median VAR Word
positionInPastArray VAR Byte
past VAR Word(lastPositionInArray)
sortedPast VAR Word(lastPositionInArray) 'we all have one

' variables for subroutines:
i VAR Byte
total VAR Word
j VAR Byte
position VAR Byte


positionInPastArray = lastPositionInArray

PAUSE 500 ' start program with a half-second delay

main:
    'get the reading:
    HIGH 8
    PAUSE 1
    RCTIME 8,1,temp


    'add the reading to an array of past readings,
    ' and reposition the oldest:
    positionInPastArray = positionInPastArray + 1
    IF positionInPastArray > lastPositionInArray THEN
        positionInPastArray = 0
    ENDIF
    past(positionInPastArray) = temp

    ' subroutine averageArray averages the array (see below):
    GOSUB averageArray

    ' subroutine medianArray() gets the middle of the array:
```

```
    GOSUB medianArray

    ' print the results:
    DEBUG " Average = ", DEC average, " Median = ", DEC median,  13
GOTO main

averageArray:
    'average the values in the array:
    total = 0
    FOR i = 0 TO lastPositionInArray
        total = total + past(i)
    NEXT
    temp = (lastPositionInArray + 1)
    average = total /temp
RETURN

medianArray:
    'simplest sorting routine; this could be faster
    FOR i = 0 TO lastPositionInArray
        position = 255  'same as -1 for a byte
        FOR j = 0 TO lastPositionInArray
            IF past(i) >= past(j) THEN
                position = position + 1
            ENDIF
        NEXT
        sortedPast(position) = past(i)
    NEXT

    ' get the middle element:
    median = sortedPast( (lastPositionInArray / 2))

    ' just for debugging purposes, print the sorted array:
    FOR i = 0 TO lastPositionInArray
        temp = sortedPast(i)
        DEBUG DEC temp, 32 ' 32 = ASCII space
    NEXT

RETURN
```

MBasic

```
lastPositionInArray CON 4 ' keep a history of the past 5 readings
temp VAR Word
average VAR Word
median VAR Word
positionInPastArray VAR Byte
past VAR Word(lastPositionInArray)
```

```
sortedPast VAR Word(lastPositionInArray) 'we all have one

' variables for subroutines:
i VAR Byte
total VAR Word
j VAR Byte
position VAR Byte


positionInPastArray = lastPositionInArray

PAUSE 500 ' start program with a half-second delay

main:
    'get the reading:
    ADIN 0, temp

    'add the reading to an array of past readings,
    ' and reposition the oldest:
    positionInPastArray = positionInPastArray + 1
    IF positionInPastArray > lastPositionInArray THEN
        positionInPastArray = 0
    ENDIF
    past(positionInPastArray) = temp

    ' subroutine averageArray averages the array (see below):
    GOSUB averageArray

    ' subroutine medianArray() gets the middle of the array:
    GOSUB medianArray

    ' print the results:
    DEBUG [" Average = ", DEC average, " Median = ", DEC median,  13]
GOTO main
averageArray:
    'average the values in the array:
    total = 0
    FOR i = 0 TO lastPositionInArray
        total = total + past(i)
    NEXT
    temp = (lastPositionInArray + 1)
    average = total /temp
RETURN

medianArray:
    'simplest sorting routine; this could be faster
```

```
    FOR i = 0 TO lastPositionInArray
        position = 255  'same as -1 for a byte
        FOR j = 0 TO lastPositionInArray
            IF past(i) >= past(j) THEN
                position = position + 1
            ENDIF
        NEXT
        sortedPast(position) = past(i)
    NEXT

    ' get the middle element:
    median = sortedPast( (lastPositionInArray / 2))

    ' just for debugging purposes, print the sorted array:
    FOR i = 0 TO lastPositionInArray
        temp = sortedPast(i)
        DEBUG [DEC temp, 32 ]' 32 = ASCII space
    NEXT

RETURN
```

PicBasic Pro

```
' Define ADCIN parameters
DEFINE ADC_BITS 10 ' Set number of bits in result
DEFINE ADC_CLOCK 3    ' Set clock source (3=rc)
DEFINE ADC_SAMPLEUS 50    ' Set sampling time in uS

TRISA = %11111111 ' Set PORTA to all input
ADCON1 = %10000010 ' Set PORTA analog and right justify result
Pause 500 ' Wait .5 second

lastPositionInArray CON 9 ' keep a history of the past ten readings
temp VAR Word
average VAR Word
median VAR Word
positionInPastArray VAR Byte

' variables for subroutines:
i VAR Byte
j VAR Byte
total VAR Word
position VAR Byte
past VAR Word(lastPositionInArray)
sortedPast VAR Word(lastPositionInArray) 'we all have one
```

```
positionInPastArray = lastPositionInArray

PAUSE 500 ' start program with a half-second delay

main:
    'get the reading:
    ADCIN 0, temp

    ' add the reading to an array of past readings,
    ' and reposition the oldest:
    positionInPastArray = positionInPastArray + 1
    IF positionInPastArray > lastPositionInArray THEN
        positionInPastArray = 0
    ENDIF
    past(positionInPastArray) = temp
    ' subroutine medianArray() gets the middle of the array:
    GOSUB medianArray
    ' subroutine averageArray averages the array (see below):
    GOSUB averageArray

    ' print the results:
    serout2 portc.6, 16468, [" Average = ", DEC average, " Median = ",
        DEC median, 10, 13]
    serout2 portc.6, 16468, [" Median = ", DEC median, 10, 13]
GOTO main

averageArray:
    ' average the values in the array:
    total = 0
    FOR i = 0 TO lastPositionInArray
        total = total + past(i)
    NEXT
    Temp = (lastPositionInArray + 1)
    average = total /temp
RETURN

medianArray:
    'simplest sorting routine; this could be faster
    FOR i = 0 TO lastPositionInArray
        position = 255 'same as -1
        FOR j = 0 TO lastPositionInArray
            IF past(i) >= past(j) THEN
                position = position + 1
            ENDIF
        NEXT
```

```
            sortedPast(position) = past(i)
        NEXT

        ' get the middle element:
        median = sortedPast(lastPositionInArray / 2)

        ' just for debugging purposes, print the sorted array:
        FOR i = 0 TO lastPositionInArray
        temp = sortedPast(i)
        serout2 portc.6, 16468, [DEC temp, 32] ' 32 = ASCII space
        NEXT

        serout2 portc.6, 16468, [10, 13]
    RETURN
```

BX-Basic

```
const lastPositionInArray as byte = 4 ' keep a history of the past 5 readings
dim past (0 to lastPositionInArray) as integer
dim sortedPast (0 to lastPositionInArray) as integer 'we all have one
dim average as integer
dim median as integer
dim positionInPastArray as byte

Sub Main()
    Debug.Print "start"
    positionInPastArray = lastPositionInArray
    call delay(0.5) ' start program with a half-second delay

    do

        ' add the reading to an array of past readings,
        ' and reposition the oldest:
        positionInPastArray = positionInPastArray + 1
        if positionInPastArray > lastPositionInArray then
            positionInPastArray = 0
        end if
        'replace the oldest reading with a new reading
        past(positionInPastArray) = getADC(13)

        ' subroutine averageArray() averages the array (see below):
        call averageArray()

        ' subroutine medianArray() sorts the array
        ' and takes the middle of the array:
        call medianArray()

        ' print the results:
```

```
        debug.print " Average = " ; cstr(average);
        debug.print " Median = " ; cstr(median)

    loop
End Sub

Sub averageArray()
    ' average the values in the array:
    dim i as byte
    dim total as integer
    total = 0
    for i = 0 to lastPositionInArray
        total = total + past(i)
    next
    average = total\cInt(lastPositionInArray+1)
end sub

sub medianArray()
    dim i as byte
    dim j as byte
    dim position as byte
    dim arrayElement as integer

'simplest sorting routine fine for short array; could be faster
    for i = 0 to lastPositionInArray
        position = 0
        for j = 0 to lastPositionInArray
            if past(i) > past(j) then
                position = position + 1
            end if
        next
        sortedPast(position) = past(i)
    next

    ' get the middle element:
    median = sortedPast((lastPositionInArray\2))

' just for debugging purposes, print the sorted array:
    for i = 0 to lastPositionInArray
        arrayElement = sortedPast(i)
        debug.print cstr(arrayElement); ",";
    next
end sub
```

Conclusion

The techniques above will help you in realizing a well-designed interaction scheme. Multitasking will give your system the ability to sense and respond at a human rhythm; edge detection, threshold setting, and peak finding will allow you to define discrete events, and debouncing and smoothing will allow you to smooth out noise and the fluctuations that are insignificant to human senses so that the system can sense change on a broader time scale. These techniques won't solve all of your interaction problems, but they'll give you a basis for defining the interaction in terms that humans can understand. When applying these techniques, keep in mind the principles of interaction discussed in the first section of this chapter.

One of the first rules of performance is that actors need to be given something to do, not told what to feel. From the action, they will find their own way to the emotion or the logic of the story. There's a similar principle at work to physical interaction design: participants need to be given something to do in an experience. If there's meaning to be had in the work, they'll interpret it themselves based on the action. If they aren't given logical sequences of action to follow or to discover, they won't engage with the work. Actions must be clearly indicated or suggested, and there must be a progression from one action to the next, whether it's action taken by the participant, by the system, or by both.

When planning your project, picture not only what it is you're making, but also the person who will be experiencing it. Picture her actions as an integrated part of the whole system, and balance the rhythm of the experience so that all participants are an active part of the whole, not just passive observers who occasionally trigger another prerecorded or computationally generated sequence. Don't tell them what to think; show them what to do.

Finally, it's important to watch how people use your project. When a user can't operate it, you'll be tempted to instruct him, and might even get impatient and argue with him. Don't do this. His interpretation of what's going on is based on his observations. If he doesn't understand part of the system, it's an indication that what you built for him to observe doesn't provide enough for his mental model of the system's workings to match yours. The best thing you can do at this point is to drop your theories about what works and watch the user carefully with an open mind. Consider conforming the interaction to his expectations, or consider what would make his model of the system match yours.