42

TIPS, TRICKS, AND TIDBITS FOR THE ROBOT EXPERIMENTER

Most every book has a "straggler" chapter that really doesn't fit with the rest. Well, this is the straggler chapter for *Robot Builder's Bonanza*. It contains various odds-and-ends discussions about robot building, including some of my own personal methodologies, rants, and observations.

But First…

All robots are different because their creators have different tasks in mind for their creations to accomplish. A robot designed to find empty soda cans is going to be radically different from one made to roam around a warehouse sniffing out the smoke and flames of a fire.

Consider that a true robot is a machine that not only *acts independently* within an environment but *reacts independently* of that environment. In describing what a robot is it's often easier to first consider what it *isn't:*

> Your car is a machine, but it's *not* a robot. Unless you outfit it with special gizmos, it has no way of driving itself (okay, so "Q" can make a self-running car for James Bond). It requires you to control it, to steer the wheels and operate the gas and brake pedals, and to roll down the window to talk to the nice police officer.

Your refrigerator is a machine, but it's *not* a robot. It may have automatic circuitry that can react to an environment (increase the cold inside if it gets hot outside), but it cannot load or unload its own food, so it still needs you for its most basic function.

Your dishwasher is a machine, but it's *not* a robot. Like the refrigerator, the dishwasher is not self-loading, may not adjust itself in response to how dirty the dishes are, and cannot be reprogrammed to accommodate changes in the soap you use, nor can it detect that you've loaded it with $100-a-plate porcelain—so go easy on the rinse cycle, thank you very much.

Other machines around your home and office are the same. Consider your telephone answering machine, your copier, or even your personal computer. All need *you* to make them work and accomplish their basic tasks.

A real robot, on the other hand, doesn't need you to fulfill its chores. A robot is programmed ahead of time to perform some job, and it goes about doing it. Here, the distinction between a robot and an automatic machine becomes a little blurry because both can run almost indefinitely without human intervention (not counting wear and tear and the availability of power). However, most automatic machines lack the means to interact with their environment and to change that environment if necessary. This feature is often found in more complex robots.

Beyond this broad distinction, the semantics of what is and is not a robot isn't a major concern of this book. The main point is this: Once the robot is properly programmed, it should not need your assistance to complete its basic task(s), barring any unforeseen obstacles or a mechanical failure.

"What Does My Robot Do?": A Design Approach

Before you can build a robot you must decide what you want the robot to do. That seems obvious, but you'd be surprised how many first-time robot makers neglect this important step. By reducing the tasks to a simple list, you can more easily design the size, shape, and capabilities of your robot. Let's create an imaginary homebuilt robot named *RoBuddy,* for "Robotic Buddy," and go through the steps of planning its design. We'll start from the standpoint of the jobs it is meant to do. For the sake of simplicity, we'll design RoBuddy so that it's an "entertainment" 'bot—it's for fun and games and is not built for handling radioactive waste or picking up after your dog Spot.

I've found that one of the first things people ask me about my robots is, "So, what does it do?" That's not always an easy question to answer because the function of a robot can't always be summarized in a quick sentence. Yet most people don't have the patience to listen to a complex explanation. Such is the quandary of the robot builder!

AN ITINERARY OF FUNCTIONS

One of the best shortcuts to explaining what a robot can do is to simply give the darned thing a vacuum cleaner. That way, when you don't feel like repeating the whole litany of

capabilities, you can merely say, "it cleans the floors." That's almost always guaranteed to elicit a positive response. So this is *Basic Requirement #1:* RoBuddy must be equipped with a vacuum cleaner. And since RoBuddy is designed to be self-powered from batteries, the vacuum cleaner needs to run under battery power too. Fortunately, auto parts stores carry a number of 12-volt portable vacuum cleaners from which you can choose.

Like the family dog that performs tricks for guests, a robot that mimics some activity amusing to humans is a great source of entertainment. One of the most useful—and effective—activities is pouring and serving drinks. That takes at least one arm and gripper, preferably two, and the arms must be strong and powerful enough to lift at least 12 ounces of beverage. We now have *Basic Requirement #2:* RoBuddy must be equipped with at least one appendage that has a gripper designed for drinking glasses and soda cans.

The RoBuddy must also have some kind of mobility so that at the very least it can move around and vacuum the floor. There are a number of ways to provide locomotion to a robot, and these were described in earlier chapters. But for the sake of description, let's assume we use the common two-wheel-drive approach, which consists of two motorized wheels counterbalanced by one or two nonpowered casters. That's *Basic Requirement #3:* RoBuddy must have two drive motors and two wheels for moving across the floor.

Since RoBuddy flits about your house all on its own accord, it has to be able to detect obstacles so it can avoid them. Obviously, then, the robot must be endowed with some kind of obstacle detection devices. We're up to *Basic Requirement #4:* RoBuddy must be equipped with passive and active sensors to detect and avoid objects in its path.

Serving drinks, vacuuming the floor, and avoiding obstacles requires an extensive degree of intelligence and is beyond the convenient capability of "hard-wired" discrete circuits consisting of some resistors, a few capacitors, and a handful of transistors. A better approach is to use a computer, which is capable of being programmed and reprogrammed at will. This computer is connected to the vacuum cleaner, arm and gripper, sensors, and drive motors. Finally, then, this is *Basic Requirement #5:* RoBuddy must be equipped with a computer to control the robot's actions.

These five basic requirements may or may not be important to you or applicable to all your robot creations. However, they give you an idea of how you should outline the functions of your robot and match them with a hardware requirement.

ADDITIONAL FEATURES

Depending on your time, budget, and construction skill, you may wish to endow your robot(s) with a number of other useful features, such as:

Sound output perhaps combining speech, sound effects, and music.
Variable speed motors so your robot can get from room to room in a hurry but slow down when it's around people, pets, and furniture.
Set-and-forget motor control, so the "brains on board" that is controlling your 'bot needn't spend all its processing power just running the drive motors.
Distance sensors for the drive motors so the robot knows how far it has traveled ("odometry").
Infrared and ultrasonic sensors to keep the robot from hitting things.
Contact bumper switches on the robot so it knows when it's hit something and to stop immediately.

LCD panels, indicator lights, or multidigit displays to show current operating status.
Tilt switches, gyroscopes, or accelerometers to indicate when the robot has fallen over, or is about to.
Voice input, for voice command, voice recognition, and other neat-o things.
Teaching pendant and remote control so you can move a joystick to control the drive motors and record basic movements.

Of course, we discussed all of these in previous chapters. Review the table of contents or index to locate the relevant text on these subjects.

Reality versus Fantasy

In building robots it's important to separate the reality from the fantasy. Fantasy is a *Star Wars* R2-D2 robot projecting a hologram of a beautiful princess. Reality is a homebrew robot that scares the dog as it rolls down the hallway—and probably hits the walls as it goes. Fantasy is a giant killer robot that walks on two legs and shoots a death ray from a visor in its head. Reality is foot-tall "trash can" robot that pours your houseguests a Diet Coke. Okay, so it spills a little every now and then . . . now you know why a robot equipped with a vacuum cleaner comes in handy!

It's easy to get caught up in the romance of designing and building a robot. But it's important to be wary of impossible plans. Don't attempt to give your robot features and capabilities that are beyond your technical expertise, budget, or both (and let's not also forget the limits of modern science). In attempting to do so, you run the risk of becoming frustrated with your inability to make the contraption work, and you miss out on an otherwise rewarding endeavor.

When designing your automaton, you may find it helpful to put the notes away and let them gel in your brain for a week. Quite often, when you review your original design, you will realize that some of the features and capabilities are mere wishful thinking, and beyond the scope of your time, finances, or skills. Make it a point to refine, alter, and adjust the design of the robot before, and even during, construction.

Understanding and Using Robot "Behaviors"

A current trend in the field of robot building is "behavior-based robotics," where you program a robot to act in some predictable way based on both internal programming and external input. For example, if the battery of your robot becomes weak, it can be programmed with a "find energy" behavior that will signal the robot to return to its battery charger. Behaviors are a convenient way to describe the core functionality of robots—a kind of "component" architecture to define what a robot will do given a certain set of conditions.

The concept of behavior-based robotics has been around since the 1980s and was developed as a way to simplify the brain-numbing computational requirements of artificial intelligence systems popular at the time. Behavior-based robotics is a favorite at the Massachusetts Institute of Technology, and Professor Rodney Brooks, a renowned leader in the field of robot intelligence, is one of its major proponents.

Since the introduction of behavior-based robotics, the idea has been discussed in countless books, papers, and magazine articles, and has even found its way into commercial products. The LEGO Mindstorms robots, which are based on original work done at MIT, use behavior principles. See Appendix A, "Further Reading," for books that contain useful information on behavior-based robotics.

WHEN A BEHAVIOR IS JUST A SIMPLE ACTION

Since the introduction of behavior-based robotics, numerous writers have applied the term *behavior* to cover a wide variety of things—to the point that *everything* a robot does becomes a "behavior." The result is that robot builders can become convinced their creations are really exhibiting human- or animal-like reactions, when all they are doing is carrying out basic instructions from a computer or simple electronic circuit. Delusions aside, this has the larger effect of distracting you from focusing on other useful approaches for dealing with robots.

Let me explain by way of an analogy. Suppose you see a magic show so many times that you end up believing the disappearing lady is really gone. Not so. It's an optical and psychological trick every time. Sometimes a robot displays a simple *action* as the result of rudimentary programming, and by calling everything it does a "behavior" we lose a clearer view of how the machine is really operating.

The following sections contain a brief discourse on behavior-based robotics, and my personal views on clarifying terms so we can get the most out of the behavior concept.

WALL FOLLOWING: A COMMON "BEHAVIOR?"

One common example of behavior-based robotics is the "wall follower," which is typically a robot that always turns in an arc, waiting to hit a wall. A sensor on the front of the robot detects the wall collision. When the sensor is triggered, the robot will back away from the wall, go forward a set amount, then repeat the whole process all over again.

This is a perfect example of how the term *behavior* has been misplaced: the *true* behavior of the robot is not to follow a *wall* but simply to turn in circles until it hits something. When a collision occurs, the robot moves to clear the obstacle, then continues to turn in a circle once again. In the absence of the wall—a reasonable change in environment—the robot would not exhibit its namesake "behavior." Or conversely, if there were additional objects in the room, the robot would treat them as "walls" too. In that case, the robot might be considered useless, misprogrammed, or worse.

If "wall following" is not a true behavior, then what is it? I won't presume to come up with an industry-standard term. The important thing to remember is that a *true behavior* is independent, or nearly so, of the robot's typical physical environment. That's *Rule Number One* to keep in mind.

Note that *environment* is not the same as a *condition.* A condition is a light shining on the robot that it might move toward or away from; an environment is a room or other area

that may or may not have certain attributes. Conditions contribute to the function of the robot, just like batteries or other electric power contribute to the robot's ability to move its motors. Conversely, environments can be ever changing and in many ways unmanageable. Environments consist of physical parameters under which the robot may or may not operate at any given time.

Robotic behaviors are most useful when they encapsulate multiple variables, particularly those are in response to external input (senses). This is *Rule Number Two* of true behavior-based robotics. The more the robot is able to integrate and differentiate between different input (senses), irrespective of its environment—and still carry out its proper programming—the more it can demonstrate its true behaviors.

THE "WALT DISNEY EFFECT"

It is tempting to endow robots with human- or animal-like emotions and traits, such as hunger (battery power) or affinity/love (a beacon or an operator clicking a "clicker"). But in my opinion these aren't behaviors at all. They are anthropomorphic qualities that merely *appear* to result in a human-type response simply because we want them to.

In other words, it's completely made up. Imagine this in the extreme: Is a robot "suicidal" if it has a tendency to drive off the workbench and break as it hits the floor? Or is it that your workbench is too small and crowded, and your concrete floor is too hard? Emotions such as love are extremely complex; as a robot builder, it's easy to get confused about what your creation can really do and feel.

In his seminal book *Vehicles,* Valentino Braitenberg gives us a study of synthetic psychology on which fictional "vehicles" demonstrate certain behavioral traits. For example, Braitenberg's Vehicle 2 has two motors and two sensors (say, light sensors). By connecting the sensors to the motors in different ways the robot is said to exhibit "emotions," or at the least actions we humans may *interpret* as quasi-intelligent or human-like emotional responses. In one configuration, the robot may steer toward the light source, exhibiting "love." In another configuration, the robot may steer away, exhibiting "fear."

Obviously, the robot is feeling neither of these emotions, nor does Braitenberg suggest this. Instead, he gives us vehicles that are fictional representations of human-like traits. It's important not to get caught up in Disneyesque anthropomorphism. A good portion of behavior-based robotics centers around *human interpretation* of the robot's mechanical actions. We interpret those actions as intelligent, or even as cognition. This is valid up to a point, but consider that only we ourselves experience our own intelligence and cognition (that is, we are "self-aware"); a robot does not. Human-like machine intelligence and emotions are in the eye of a human beholder, not in the brain of the robot. This, however, may change in the future as new computing models are discovered, invented, and explored.

ROBOTIC FUNCTIONS AND ERROR CORRECTION

When creating behaviors for your robots, keep in mind the *function* that you wish to accomplish. Then, consider how that function is negatively affected by variables in the robot's likely environments. For practical reasons (budget, construction skill), you must consider at least some of the limitations of the robot's environment in order to make it reliably demonstrate a given behavior. A line-following robot, which is relatively easy to build

and program, will not exhibit its line-following behavior without a line. By itself, such a robot would merely be demonstrating a simple action. But by adding *error correction*—to compensate for unknown or unexpected changes in environment—the line-following robot begins to demonstrate a useful behavior. This behavior extends beyond the robot's immediate environmental limits. The machine's ability to go into a secondary, error-correcting state to find a line to follow is part of what makes a valid line-following behavior—even more so if in the absence of a line to follow the robot can eventually make its own.

Error correction is *Rule Number Three* of behavior-based robotics. Without error correction, robots operating in restrictive environments are more likely to exhibit simple, even stupid, actions in response to a single stimulus. Consider the basic "wall-following" robot again: it *requires* a room with walls—and, at that, walls that are closer together than its turning radius. Outside or in a larger room, the robot "behaves" completely differently, yet its programming is *exactly* the same. The problem of the wall-following robot could be fixed either by adding error correction or by renaming the base behavior to more accurately describe what it physically is doing.

ANALYZING SENSOR DATA TO DEFINE BEHAVIORS

By definition, behavior-based robotics is reactive, so it requires some sort of external input by which a behavior can be triggered. Without input (a light sensor, ultrasonic detector, bumper switch, etc.) the robot merely plays out a preprogrammed set of moves—simple actions, like a player piano. More complex behaviors becomes possible if the following capabilities are added:

The ability to analyze the data from an analog, as opposed to a digital, sensor. The output of an analog sensor provides more information than the simple on/off state of a digital sensor. Let's call this *sensor parametrics.*

The ability to analyze the data from multiple sensors, either several sensors of the same type (a gang of light-sensitive resistors, for example) or sensors of different types (a light sensor and an ultrasonic sensor). This is commonly referred to as *sensor fusion.*

Let's consider sensor parametrics first. Suppose your robot has a temperature sensor connected to its onboard computer. Temperature sensors are analog devices; their output is proportional to the temperature. You use this feature to determine a set or range of preprogrammed actions, depending on the specified temperature. This set of actions constitutes a behavior or, if the actions are distinct at different temperatures, a variety of behaviors. Similarly, a photophilic robot that can discern the brightest light among many lights also exhibits sensor analysis from parametric data.

Sensor fusion analyzes the output of several sensors. Your robot initiates the appropriate behavioral response as a result. For example, your robot may be programmed to follow the brightest light but also detect obstructions in its path. When an obstruction is encountered, the robot is programmed to go around it, then continue—perhaps from a new direction—toward the light source.

Sensor fusion helps provide error correction and allows a robot to continue exhibiting its behavior (one might call it the robot's "prime directive") even in the face of unpredictable environmental variables. The variety, sophistication, and accuracy of the sensors

determine how well the robot will perform in any given circumstance. Obviously, it's not practical—economic or otherwise—to ensure that your robot will work flawlessly under all environments and conditions. But the more you give your robots the ability to overcome common and reasonable environmental variables (such as socks on the floor), the better it will display the behavior you want.

THE ROLE OF SUBSUMPTION ARCHITECTURE

Subsumption architecture isn't an odd style of building. Rather, it's a technique devised by Dr. Brooks at MIT that has become a common approach for dealing with the complexities of sensor fusion and artificial machine intelligence. With subsumption, sensor inputs are prioritized. Higher-priority sensors override lower-priority ones. In the typical subsumption model, the robot may not even be aware that a low-priority sensor was triggered.

More complex hybrid systems may employ a form of simple subsumption along with more traditional artificial intelligence programming. The robot's computer may evaluate the relative merits of low-priority sensors and use this information to intuit a unique course of action, perhaps one in which direct programming for the combination of input variables does not yet exist. In some cases, the output of a low-priority sensor may moderate the interpretation of a high-priority one.

As an example, a fire-fighting robot may have both a smoke detector and a flame detector. The smoke detector is likely to sense smoke before any fire can be identified, since smoke so easily permeates a structure. Therefore, the smoke sensor will likely be given a lower priority to the flame detector, since it is so easily triggered. But consider that flames can exist without a destructive fire (e.g., a fireplace and candlelight, both of which do not emit much smoke under normal circumstances). Rather than have the robot totally ignore its other sensors when the high-priority flame detector is triggered, the robot instead integrates the output of both flame and smoke sensors to determine what is, and isn't, a fire that needs to be put out.

Multiple Robot Interaction

An exciting field of research is the interaction of several robots working together. Rather than build one big, powerful robot that does everything, multirobot scenarios combine the strengths of two or more smaller, simpler machines to achieve synergy: the whole is greater than the parts. Anyone who has seen the now-classic science fiction film *Silent Running* knows what three diminutive robots (named Huey, Luey, and Dewey, by the way) can do!

Robot "tag teams" are common in college and university robot labs, where groups of robot researchers compete with their robots as the players of a game (robo-soccer is popular). Each robot in the competition has a specific job, and the goal is to have them work together. There are three common types of robot-to-robot interaction:

Peer-to-peer. Each robot is considered equal, though each one may have a different job to do, based on predefined programming The workload may also be divided based on physical proximity to the work, and whether the other robots in the group are busy doing other things.

Queen/drone. One robot serves as the leader, and one or more additional robots serve as worker drones. Each drone takes it's work orders directly from the queen and may interact only peripherally with the other drone 'bots.
Convoys. Combining the first two types, the leader of the convoy is the "queen" robot, and the other robots act as peers among themselves. In convoy fashion, each robot may rely on the one just ahead for important information. This approach is useful when the "queen" is not capable, for computer processing reasons or otherwise, to control a large number of fairly mindless drones.

Why all the fuss with multiple robots? First and foremost because it's generally easier and cheaper to build many small and simple robots than a single big and complex one. Second, the mechanical failure of one robot can be compensated for by the remaining good robots. In many instances, the "queen" or leader robot is no different than the others, it just plays a coordinating role. In this way, should the leader 'bot go down for the count, any other robot can easily take its place. And third, work tends to get done faster with more hands helping.

Dealing with Failures

Few robots work perfectly when you flip the switch the first time. Failure is common in robot building and should be expected. As you learn from these failures you will build better robots. Failure can occur at the onset when you first try a new design, or it can occur at any point thereafter, as the robot breaks down for one reason or another.

MECHANICAL FAILURE

Mechanical problems are perhaps the most common failure. A design you developed just doesn't work well, usually because the materials or the joining methods you used were not strong enough. Avoid overbuilding your robots (that tends to make them too expensive and heavy), but at the same time strive to make them physically strong. Of course, "strong" is relative: a lightweight, scarab-sized robot needn't have the muscle to tote a two-year-old on a tricycle. At the very least, however, your robot construction should support its own weight, including batteries.

When possible, avoid "slap-together" construction, such as using electrical or duct tape. These methods are acceptable for quick prototypes but are unreliable for long-term testing. When gluing parts in your robot, select a glue that is suitable for the materials you are using. Epoxy and hot-melt glues are among the most permanent. You may also have luck with cyanoacrylate (CA) glues, though the bond may become brittle and weak over time (a few years or more, depending on humidity and stress).

Tip:

Use the "pull test" to determine if your robot construction methods are sound. Once you have attached something to your robot—using glue, nuts and bolts, or whatever—give it a healthy tug. If it comes off, the construction isn't good enough. Look for a better way.

ELECTRICAL FAILURE

Electronics can be touchy, not to mention extremely frustrating, when they don't work right. Circuits that functioned properly in a solderless breadboard may no longer work once you've soldered the components in a permanent circuit, and vice versa. There are many reasons for this, including mistakes in wiring, odd capacitive effects, even variations in tolerances due to heat transfer. Here are some pointers:

If a circuit doesn't work from the get-go, review your wiring and make necessary repairs.

If the circuit fails after some period of use, the cause may be a short circuit or broken wire, or it could be a burned-out component. Example: if your motors draw too much current from the drive circuitry you run the risk of permanently damaging some semiconductors.

Certain electronic circuit construction techniques are better suited for an active, mobile robot. Wire-wrap is a fast way to build circuits, but its construction can invite problems. The long wire-wrap pins can bend and short out against one another. Loose wires can come off. Parasitic signals and stray capacitance can cause "marginal" circuits to work, then not work, and then work again. For an active robot it may be better to use a soldered circuit board, perhaps even a printed circuit board of your design (see Chapter 6, "Electronic Construction Techniques," for more information).

Some electrical problems may be caused by errors in programming, weak batteries, or unreliable sensors. For example, it is not uncommon for sensors to occasionally yield totally wacky results. This can be caused by design flaws inherent in the sensor itself, spurious data (noise from a motor, for example), or corrupted or out-of-range data. Ideally, the programming of your robot should anticipate occasional bad sensor readings and basically ignore them. A perfectly acceptable approach is to throw out any sensor reading that is outside the statistical model you have decided on (e.g., a sonar ping that says an object is 1048 feet away; the average robotic sonar system has a maximum range of about 35 feet).

PROGRAMMING FAILURE

As more and more robots use computers and microcontrollers as their "brains," programming errors are fast becoming one of the most common causes of failure. There are three basic kinds of programming "bugs." In all cases, you must review the program, find the error, and fix it:

Compile bug, caused by bad syntax. You can instantly recognize these because the program compiler or downloader will flag these mistakes and refuse to continue. You must fix the problem before you can transfer the program to the robot's microcontroller or computer.

Run-time bug, caused by a disallowed condition. A run-time bug isn't caught by the compiler. It occurs when the microcontroller or computer attempts to run the program. An example of a common run-time bug is the use of an out-of-bounds element in an array (for instance, trying to assign a value to the thirty-first element in a 30-element

array). Run-time bugs may also be caused by missing data, such as looking for data on the wrong input pin of a microcontroller.

Logic bug, caused by a program that simply doesn't work as anticipated. Logic bugs may be due to simple math errors (you meant to add, not subtract) or by mistakes in coding that cause a different behavior than you anticipated.

Task-Oriented Robot Control

As "workers," robots have a task to do. In many books on robotics theory and application, these tasks are considered "goals." Personally, I'm not big on the term *goal* because that suggests a human emotion involving desire. The robot you build will have no "desire" to fetch you a can of soda, but will merely do so because its programming tells it to. Instead, I prefer the term *task*—a defined job that the robot is expected to accomplish. A robot may be given multiple tasks at the same time, such as the following:

1. Get me a can of Dr. Pepper;
2. Avoid running into the wall while doing so;
3. Watch out for the cat and other ground-based obstacles;
4. And remember where you came from so you can bring the soda back to me.

These tasks form a hierarchy. Task 4 cannot be completed before task 1. Together, these two form the *primary directive tasks* (shades of *Star Trek* here—okay, I admit it: I'm a Trekker!). Tasks 2 and 3 may or may not occur; these are *error mode tasks.* Should they occur, they temporarily suspend the processing of the primary directive tasks.

PROGRAMMING FOR TASKS

From a programming standpoint, you can consider most any job you give a robot to look something like this:

```
Do Task X until
      on error Do Task Y
            repeat
            Task Y until no error
      resume Task X
Task X complete
```

X is the primary directive task, the thing the robot is expected to do. *Y* is a special function that gets the robot out of trouble should an error condition—of which there may be many—occurs. Most error modes will prevent the robot from accomplishing its primary directive task. Therefore, it is necessary to clear the error first before resuming the primary directive.

Note that it is entirely possible that the task will be completed without any kind of complication (no errors). In this case, the error condition is never raised, and the *Y* functionality

is not activated. The robot programming is likewise written so that when the error condition is cleared, it can resume its prime directive task.

MULTITASKING ERROR MODES FOR OPTIMAL FLEXIBILITY

For a real-world robot, errors are just as important a consideration as tasks. Your robot programming must deal with problems, both anticipated (walls, chairs, cats) and unanticipated (water on the kitchen floor, no sodas in the fridge). The more your robot can recognize error modes, the better it can get itself out of trouble. And once out of an error mode, the robot can be reasonably expected to complete its task.

How you program various tasks in your robot is up to you and the capabilities of your robot software platform. If your software supports multitasking (BasicX, OOPic, LEGO Mindstorms, and others), then try to use this feature whenever possible. By dealing with tasks as discrete units, you can better add and subtract functionality simply by including or removing tasks in your program.

Equally important, you can make your robot automatically enter an error mode task without specifically waiting for it in code. In non-multitasking procedural programming, your code is required to repeatedly check (poll) sensors and other devices that warn the robot of an error mode. If an error mode is detected, the program temporarily branches to a portion of the code written to handle it. Once the error is cleared, the program can resume execution where it left off.

With a multitasking program, each task runs simultaneously. Tasks devoted to error modes can temporarily take over the processing focus to ensure that the error is fixed before continuing. The transfer of execution within the program is all done automatically. To ensure that this transfer occurs in a logical and orderly manner, the program should give priorities to certain tasks. Higher-priority tasks are able to take over ("subsume," a word now in common parlance) other running tasks when necessary. Once a high-priority task is completed, control can resume with the lower-priority activities, if that's desired.

GETTING A PROGRAM'S ATTENTION VIA HARDWARE

Even in systems that lack multitasking capability it's still possible to write a robot control program that doesn't include a repeating loop that constantly scans (polls) the condition of sensors and other input. Two common ways of dealing with unpredictable external events are using a timer (software) interrupt or a hardware (physical connection) interrupt.

Timer interrupt A timer built into the computer or microcontroller runs in the background. At predefined intervals—most commonly when the timer overflows its count—the timer grabs the attention of the microprocessor, which in turn temporarily suspends the main program. The microprocessor runs a special timer interrupt program, which in the case of a task-based robot would poll the various sensors and other input looking for possible error modes. (Think of the timer as a heart beat; at every beat the microprocessor pauses to do something special.)

If *no* error is found, the microprocessor resumes the main program. If an error is found, the microprocessor runs the relevant section in code that deals with the error. Timer

interrupts can occur hundreds of times each second. That may seem like a lot in human terms, but it can be trivial to a microprocessor running at several million cycles per second.

Hardware interrupt A hardware interrupt is a mechanism by which to immediately get the attention of the microprocessor. It is a physical connection on the microprocessor that can in turn be attached to some sensor or other input device on the robot. With a hardware interrupt the microprocessor can spend 100 percent of its time on the main program and temporarily suspend it if, and *only* if, the hardware interrupt is triggered.

Hardware interrupts are used extensively in most computers, and their benefits are well established. Your PC has several hardware interrupts. For example, the keyboard is connected to a hardware interrupt, so when you press a key the processor immediately fetches the data and makes it available to whatever program is currently running. The standard PC architecture has room for 16 hardware interrupts, even though the microprocessor uses just one interrupt pin. The one pin is *multiplexed* to make 16 separate inputs. You can do something similar in your own robot designs.

Glass half-empty, half-full There are two basic ways to deal with error modes. One is to treat them as "exceptions" rather than the rule:

In the exception model, the program assumes no error mode and only stops to execute some code when an error is explicitly encountered. This is the case with a hardware interrupt.

In the opposite model, the program assumes the possibility of an error mode all the time and checks to see if its hunch is correct. This is the case with the timer interrupt.

The approach you use will depend on the hardware choices available to you. If you have both a timer and a hardware interrupt at your disposal, the hardware interrupt is probably the more straightforward method because it allows the microprocessor to be used more efficiently.