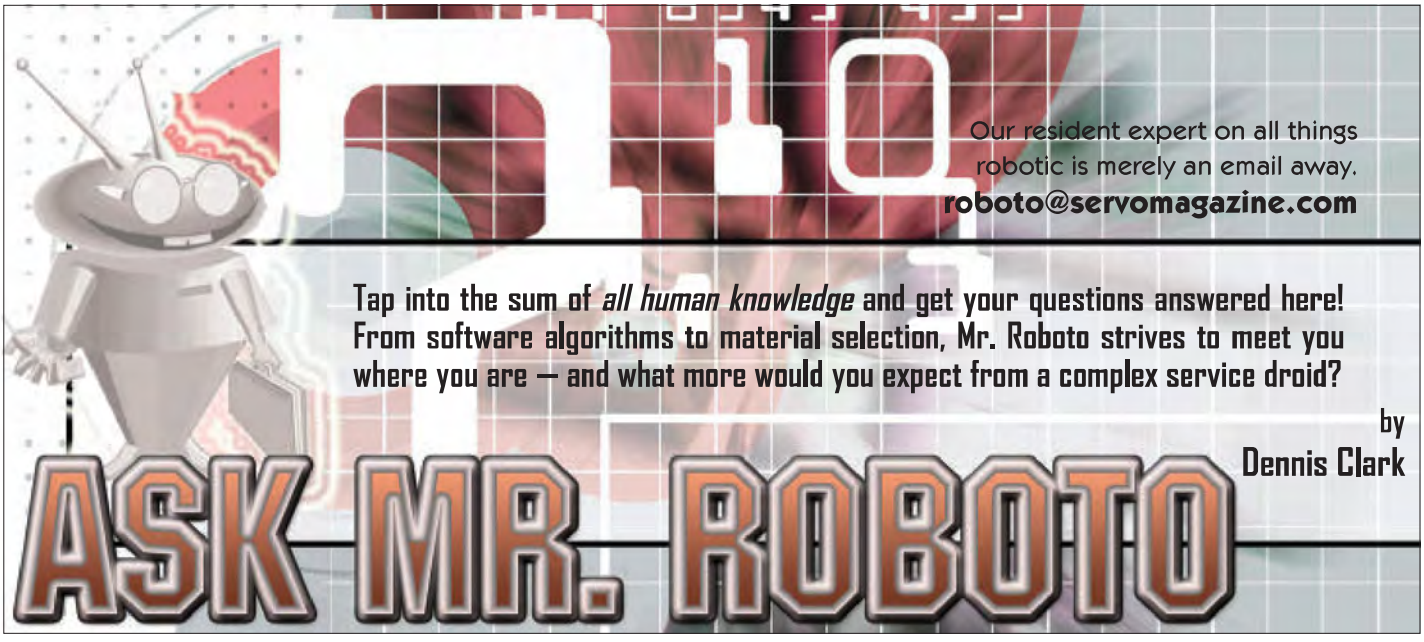

Background

This month's column can be called either "No good deed goes unpunished" or "Curiosity killed the cat." Last month, we looked at controlling an iRobot Roomba platform with an Arduino UNO. Questions came in about problems with my Roomba program, and I was asked if I could make the code more robust and useful. I never shy away from a challenge, so here you go! This month, we'll look at ways of programming that will allow multiple tasks to run (apparently) at the same time.

On Code Reliability:

Between last month and this month, I have discovered many interesting things. Most obvious is that there is more than one kind of Roomba controller! Some are faster or more tolerant of the way you send it SCI commands; others, not so much.

My old Roomba 410 is either faster or more tolerant of lots of SCI commands coming in. My much newer Roomba 440 is not so tolerant. Code that ran great on the Roomba "Red" ran very badly with resets, ignored commands, and "went into the weeds." on my Roomba 440. I found that if I delayed for about 5 ms after each command, that everything was happy! Okay. So be it. That delay isn't too bad if everything runs well when we use it.

Remember from last month I said that my Roomba Red would not run straight? I took the wheel assemblies apart and cleaned the wheel encoders, and that straightened everything right up (pun intended)!

On More Useful Code:

Last month's code was little more than a giant "if/then" statement that did a little bit of robot-ish stuff. This month, I'm going to talk about finite state machines in embedded processors like we use for many of our robots. I'm not going to go into CS-101 programming course language about state machines, so suffice it to say that a state machine allows a program to move from one known condition — or *state* — to another one in a predictable manner.

"Huh!" you say. "So what?" Indeed, that sounds like any program you write, doesn't it? The beauty of a state machine, however, is that your code will *hold that state* while you tell your microcontroller to go off and do something else. (I'll bet that got your attention.)

A long time ago (you know, like back in the 1990s), the robotics pioneer Rodney Brooks wrote about something called robot *subsumption architectures* to create what looked like intelligent behavior in robots. Each element in a subsumption architecture is a behavior that may or may not use sensors for input, and which does use motors (or similar actuators) for output. Higher-level behaviors take control of (subsume) output actuators, even if lower priority behaviors are active. Each behavior is implemented in what Brooks called *modified finite state machines*.

A *modified* state machine in Brooksian terms is a finite state machine (FSM) that uses timers as an input, as well as sensors and states. In short, using a subsumption architecture can make your robot look like it is "alive" and intelligently responding to its surroundings.

In my opinion, what is really cool about this type of behavioral programming is that your robot will start doing things that you did not tell it to do because of the states left and then returned to in your subsumption architecture behaviors. This is called *emergent behavior* and it is what makes our machines look alive.

To learn more about how simple behaviors can look like actual intelligence, I really recommend that you read Valentino Braitenberg's *Vehicles, Experiments in Synthetic Psychology*. This book is a very easy read and will keep you busy for hours

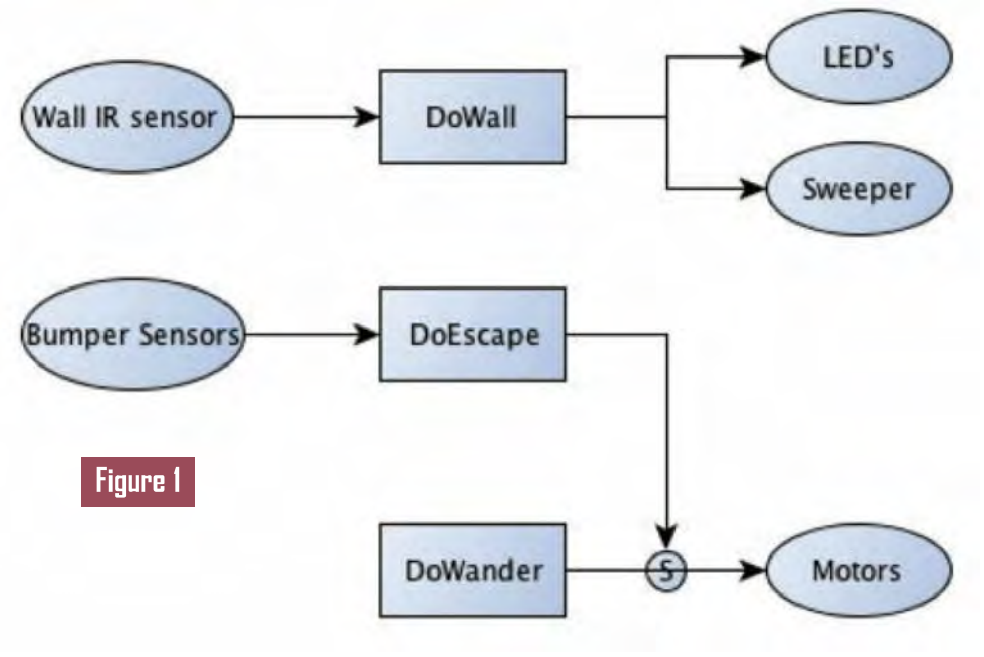

Figure 1

implementing behaviors in your robots. (This way, the next time you hear someone say "Braitenberg Type 2" robot, you'll know exactly what they are talking about!)

I am going to discuss basic subsumption architecture in this article. There are variants on this theme. A behavioral module may have inputs that are suppressed and replaced with other inputs. Or, it may have outputs which can be inhibited to allow outputs from other modules to be used. We will keep this discussion focused on suppressors and timers.

If you are interested in knowing more, there are some books to check out in the **sidebar**. Okay, I've introduced you to new terms and concepts perhaps. Now, you need to know how to design and implement them, so read on.

Designing a Subsumption Architecture

This is going to be a very simple introduction to behavioral programming, involving only a few behaviors. The beauty of the system, however, is that you can layer on more behaviors as you get them to work without modifying existing ones. This means that your behaviors are modular and can be modified without worrying about other behaviors that are currently working fine. Your emergent behaviors might change, however ...

Our new Roomba code will have three defined behaviors to start with:

1. Wander
2. Escape
3. "I see a wall"

The first two behaviors control the drive motors that move the Roomba around. The third one just lights up LEDs and turns on the sweeper motor to tell us that it saw a wall with the wall sensor.

Figure 1 shows what the subsumption network looks like using these three behaviors.

In **Figure 1**, each of the square boxes are behaviors. The *DoWall* behavior stands by itself and has a sensor input and output actuators that are LEDs and the sweeper brush. That behavior isn't all that interesting, but this is how it is represented.

The next two behaviors have subsumption characteristics. The *DoWander* behavior has no sensor input, but it does run the motors. As you can see in **Listing 1**, DoWander just makes the robot go forward — not interesting, but it does something.

The *DoEscape* behavior has a higher priority than *DoWander*, and

Listing 1: DoWanders behavior code.

```
void DoWander(void)
// Define a wander algorithm, or just go straight.
{
  if (priority < wander)
  {
    Serial.println("Wander");
    priority = wander;
    GoForward();
  }
}
```

Listing 2: RoombaBehave1 main Arduino loop.

```
/*
*  The nice thing about state machines is that they allow your
*  main loop to be really simple.
*/
void loop(void)
{
  UpdateSensors();
  DoWall();
  DoWander();
  DoAvoid();
  DoEscape();
  DoPanic();
}
```

uses the bumper sensors as inputs. The output of *DoEscape*, however, goes to the little circle labeled "S." The S stands for suppresses. Therefore, *DoEscape* suppresses the output of *DoWander* and takes over the motors. When *DoEscape* is done with its task, it will return control to *DoWander* or whatever other behavior has the highest currently active priority.

In general, higher priority behaviors are at the top of the **diagram**. That doesn't have to be true, however. All you need to do is look for the behavior that points to the suppressor that has the final say to see which one has the highest priority. When you have all of your behaviors coded, all you need to do in your code's main loop is call everything, and each behavior will handle itself. **Listing 2** shows what our Roomba behavioral program's main Arduino loop looks like. Simple, isn't it?

If you look at **Listing 2** again, you'll note that I have more behaviors there than what this article is talking about. The code (which you can find at the article link as RoombaBehave1.zip) has other behaviors *stubbed* in that can be added later. I am leaving these as an exercise for the reader. (Maybe I'll add some details in a later column, we'll see ...)

I have placed these behaviors in ascending order of priority with the lowest priority called first. Each of these behaviors are *modified finite state machines* (MFSM). There is a standard way to represent FSMs in the programming world.

The *DoWall* behavior is a simple call and has no real state machine; it is always active on each pass through the main loop. Likewise, the *UpdateSensors()* is just a function call that happens each pass through the main loop so that all the behaviors have access to the most recent sensor data. Our first real state machine is *DoWander*.

Figure 2 shows the state machine for that behavior. It is very simple. The first state is the "idle" state where it looks to see if it can even run. The second state goes active on the next pass of the function if the behavior is of a high enough priority to run. As you can see from **Listing 1**, however, this is little more than an if/then statement. Our first real FSM is the *DoEscape* behavior, which tries to get our robot out of a predicament when it runs into something.

Figure 3 shows the FSM diagram for *DoEscape*. Every state that uses the word "Done" is using a timer to decide when the state is finished. This timer is NOT the *delay()* function; that would block the function call until the delay is finished. Instead — as you can see in **Listing 3** — we use the Arduino background milliseconds counter to determine if we are done. If not done, the function exits, leaving the state machine in the same state until the next call of the behavior function.

In the C language, a *switch/case* construct is the simplest way to implement an FSM. Note the timer function *millis()*; I use this as my background timer to know when certain states are ready to change.

Subsumption, behaviors, finite state machines, oh my! This all sounds so technical, but as you can see, the

SELECTED READING

"Cambrian Intelligence, The Early History of the New AI" by Rodney A. Brooks, MIT Press, 1999.

"Mobile Robots, Inspiration to Implementation" by Jones and Flynn, A K Peters LTD, 1993.

"Vehicles, Experiments in Synthetic Psychology" by Valentino Braitenberg, MIT Press 1986.

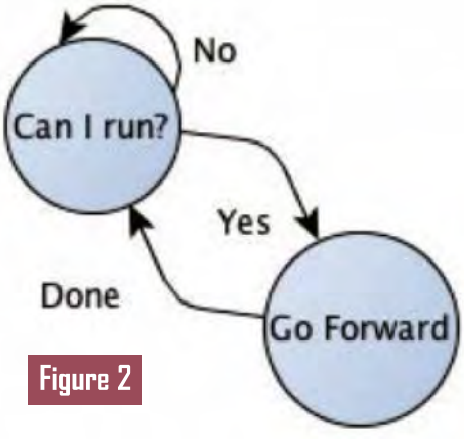

Figure 2

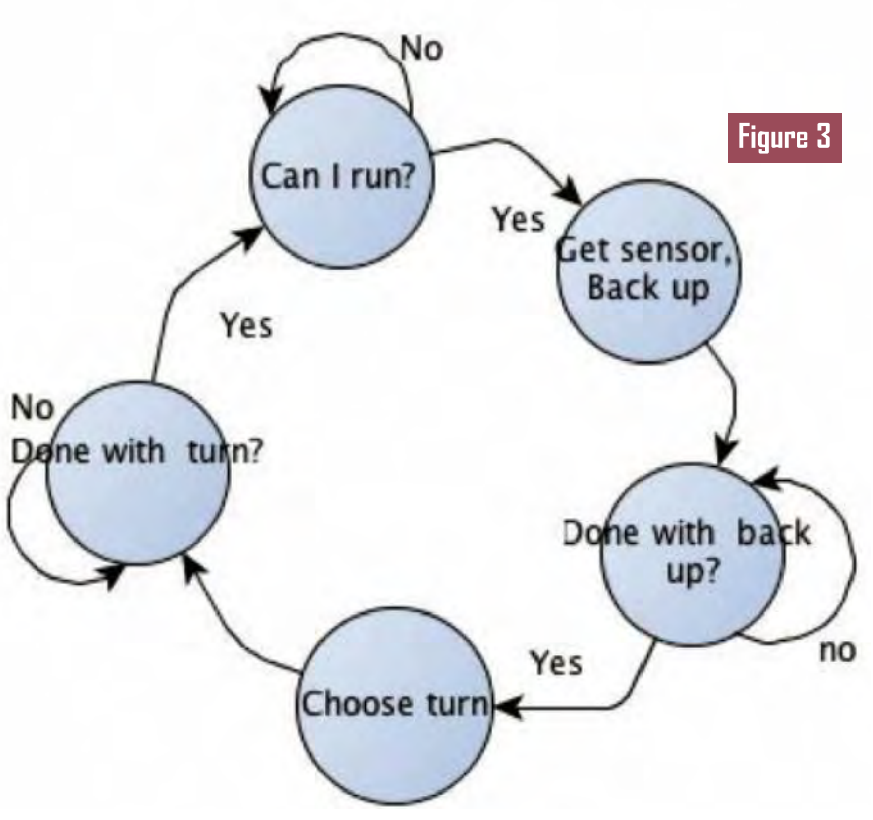

Figure 3

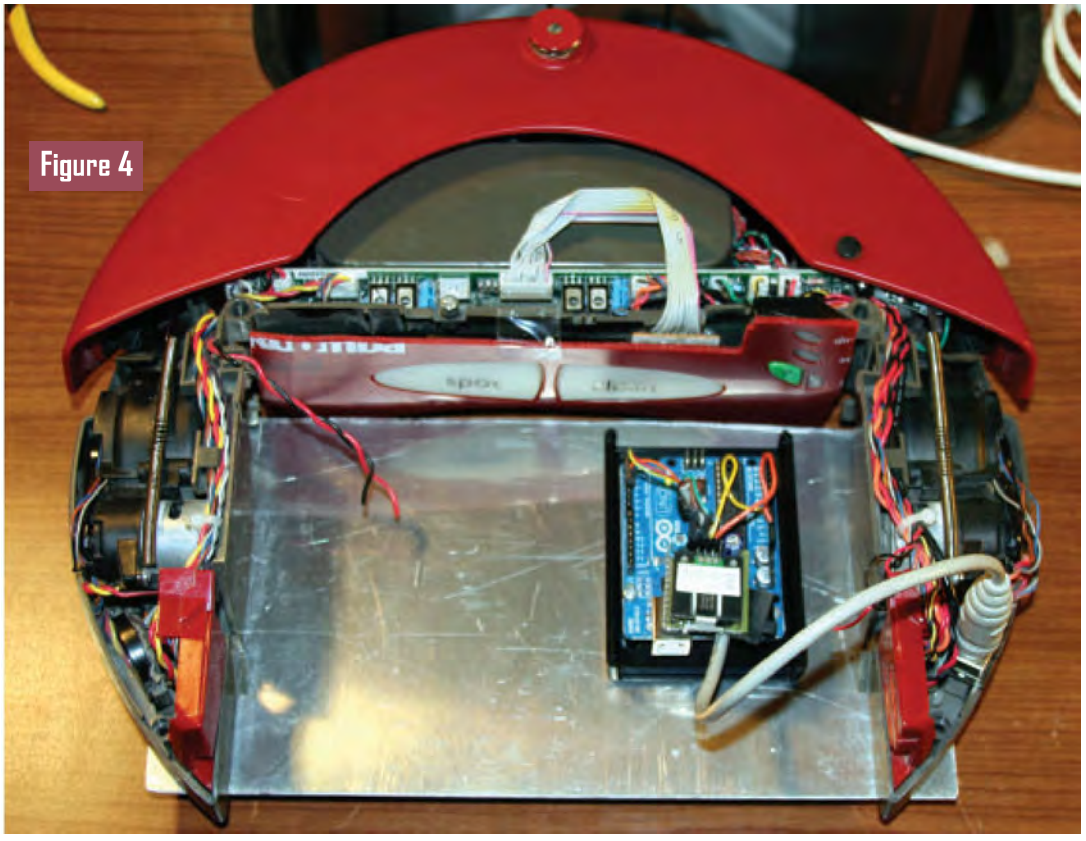

Figure 4

code isn't that complex. This is not to say that it is easy to write! You will spend a great deal of time tweaking and tuning your timing, and choosing your state steps before your behaviors work properly.

The FSM is a basic cornerstone of embedded programming, so if you are inclined to get a job in automation or embedded engineering, put some time in on these concepts — you'll need them!

Back to the Hardware

While working with my Roomba, I got tired of my taped-on dead weights falling off and my Roomba stopping every time it changed directions. So, I

Listing 3: DoEscape behavior.

```
void DoEscape(void)
// Define a bumper response escape behavior
{
  static uint32_t ticker = 0;
  static uint8_t state = 0;
  static uint8_t dir = 0;

  if ((mBumpers == 0) && (state ==0))
  // Don't do anything if nothing to do
  {
    return;
  }
  if (priority <= escape)
  {
    priority = escape;
    switch(state)
    {
      case 0: // record bumper, backup
        dir = mBumpers;
        ticker = millis() + 1000;  // 1 second
        GoBackward();
        GoSong();
        state = 1;
        Serial.print("Bumper dir: ");
        Serial.println(dir,HEX);
        break;

      case 1:             // wait for backup
                          // done
        if (ticker < millis())
        {
          state = 2;
          Serial.println("Do turn");
        }
        break;

      case 2:             // choose turn
                          // direction
        if (dir == RIGHT)
        {
          Serial.println("spin left");
          SpinLeft();
        }
        else if (dir == LEFT)
        {
          Serial.println("spin right");
          SpinRight();
        }
        else
        {
          Serial.println("ahead");
          SpinRight();
        }
         ticker = millis() + 1000;
         state = 3;
        break;

      case 3:             // wait for turn done
        if (ticker < millis())
        {
          state = 0;
          dir = 0;
          priority = idle;
          Serial.println("All done");
        }
        break;
    }
  }
}
```

added a "cargo bay" made of a simple aluminum plate. **Figure 4** shows my new robot with a handy rear deck bolted on.

If you are like me and really don't know how to use a CAD drawing package (yet) to make these simple mechanical plates, **Figure 5** shows my handy-dandy template made from cereal box paperboard and a sharp leather awl that I used to poke screw-hole locations in the paperboard.

It works great and allows me to fiddle with details by just bending the paperboard and using scissors to modify the plates. It is easy to achieve good results with these simple tools.

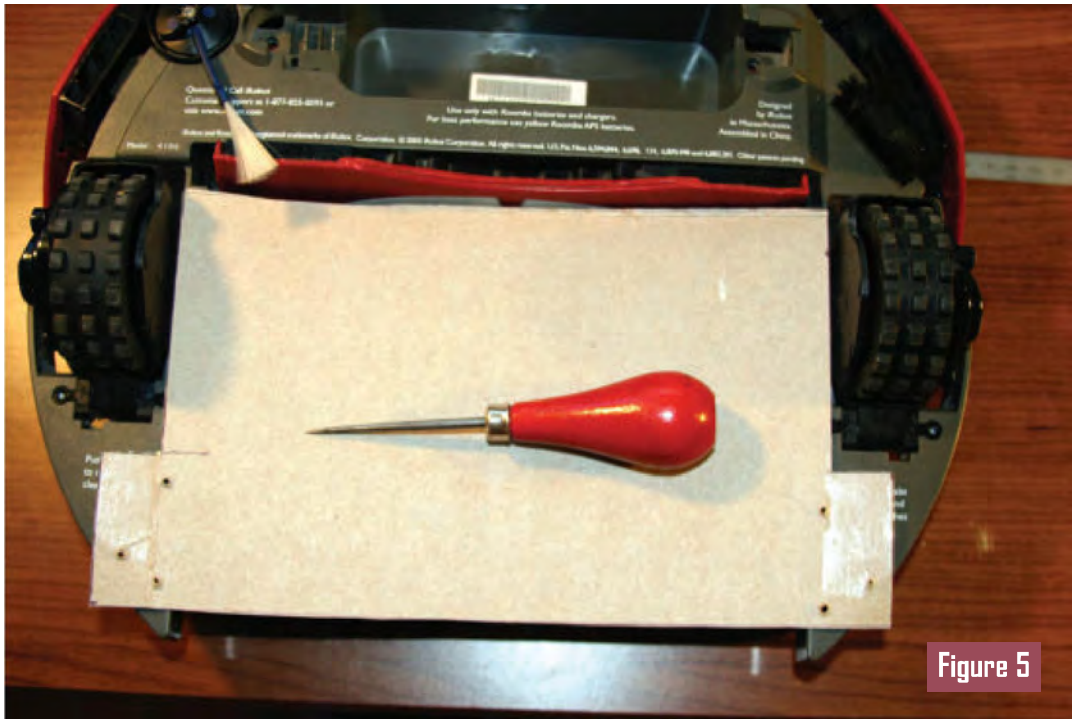

Figure 5

Well, that's all for another month. I love hearing from you, so remember, if you have a robot question, please drop me an email at roboto@servomagazine.com and I'll do my best to answer them. Until next month, keep on building robots! SV