Keith L. Doty
Professor EE
Reid Harrison

University of Florida
Department of Electrical Engineering
EEL 5934 Intelligent Machines Design Lab

Robot Programming

CRITTER: AN EXAMPLE ROBOT

Let's consider a robot that was built at MIL several years ago. The robot, called Critter, has two actuators: a left and right drive motor. Critter has nine sensors. The sensors are described below.

- **photosensitive "nose":** Critter's nose is an array of three Cadmium Sulfide (CdS) photocells. The array points at the floor in front of Critter and can be used to detect light or dark areas on the floor.
- **photosensitive "eye":** Critter's eye is also an array of three CdS cells. These cells are aimed parallel with the ground.
- **puck sensor:** Critter can push small pucks around the floor. He has a microswitch near the floor that is tripped by the presence of a puck.
- **whiskers:** Critter has two touch-sensitive whiskers--one on each side facing forward.

This fairly simple robot will be used for several programming examples.

GATHERING SENSORY INFORMATION

IC makes it very easy to write processes that run in the background and perform useful tasks. Perhaps the most useful task these processes can perform in robot software is that of **automatically gathering sensory information.**

Consider the common IR proximity sensor. This sensor consists of an IR transmitter and an IR receiver. With the transmitter turned on, the sensor detects nearby obstacles. With the transmitter turned off, the sensor can still detect IR beacons. Why not build a circuit that allows the processor to switch the transmitter on and off? Then we can detect both beacons and obstacles.

A **sensory module** that runs in the background can make this type of sensor *very* easy to use. Consider the following code segment:

```
/* global variables */
int obstacle_sense, beacon_sense;

void sensor_module()
{
  while(1) {
    turn_transmitter_on();
    wait(50);              /* Wait 50 msec for transients to die */
    obstacle_sense = analog(0);

    turn_transmitter_off();
    wait(50);              /* Wait 50 msec for transients to die */
    beacon_sense = analog(0);
  }
}
```

```
void main()
{
  start_process(sensor_module());

  /* Put the rest of the program here */
}
```

As long as `sensor_module()` is running in the background, any other piece of code can tell what the robot is sensing by looking at the global variables `obstacle_sense` and `beacon_sense`. It is easy to see how a background process that "runs" the sensors can make the programmers life *much* easier in the long run. Even if your sensors are much simpler than this example, a sensory module can ease programming by translating expressions like `analog(5)` into descriptive variable names. Even more importantly, if you ever reroute your sensors in hardware, you need only to change one small piece of code.

PROGRAMMING BEHAVIORS

In this class, we are building agents to operate in the real world. When designing software for real robots, it is important to keep a tight sensing-acting loop. The robot should swiftly react to changes in the world it is sensing. A good way to build up functionality in your robot is to develop several simple **behavior modules.** Each **behavior** links some subset of the robot's sensors to some (or all) of its actuators.

The following program listing illustrates a complete program that contains a behavior module called `line_following_behavior()`. This module implements a line-following behavior on Critter, the robot described previously.

Program Listing 1: Simple Line-Following Behavior

```
/* GLOBALS */

/* Sensory Registers */
int left_nose, center_nose, right_nose;
int left_eye, center_eye, right_eye;
int puck_sense;

void sensor_module()  /* Read sensors into global variables at 10 Hz */
{
  while(1) {
    left_nose   = analog(0);
    center_nose = analog(1);
    right_nose  = analog(2);
    left_eye    = analog(3);
    center_eye  = analog(4);
    right_eye   = analog(5);
    puck_sense  = analog(6);
    wait(100);    /* Wait 100 msec */
  }
}
```

```
void line_following_behavior()
{
  while(1) {
    if (left_nose < center_nose && left_nose < right_nose) {
      motor(0,0);
      motor(1,100);
    }
    else if (right_nose < center_nose && right_nose < left_nose) {
      motor(0,100);
      motor(1,0);
    }
    else {
      motor(0,100);
      motor(1,100);
    }
  }
}

void main()
{
  beep();
  start_process(sensor_module());
  start_process(line_following_behavior());
}
```

With a minimal number of modifications, we could change the line-following behavior into a light-following behavior. We just change the sensory input from Critter's "nose" sensor array to Critter's "eye" sensor array. We also change the greater-than signs to less-than signs so that Critter will follow the brightest stimulus instead of the darkest stimulus.

Program Listing 2: Light-following behavior module

```
void light_following_behavior()
{
  while(1) {
    if (left_eye > center_eye && left_eye > right_eye) {
      motor(0,0);
      motor(1,100);
    }
    else if (right_eye > center_eye && right_eye > left_eye) {
      motor(0,100);
      motor(1,0);
    }
    else {
      motor(0,100);
      motor(1,100);
    }
  }
}
```

Now that we have developed a small repertoire of behavior modules, we can link them together to achieve more advanced performance. Suppose we want Critter to follow a dark line until he happens upon a puck. When he gets a puck, we want him to follow a light until he loses the puck. This functionality clearly involves more than one behavior. Notice that both behavior modules (line-following and light-following) output to Critter's drive motors. We need to develop a **behavior arbitration** scheme to decide which behavior module gets to control Critter's drive motors at a given instant in time.

The following listing is a complete program which implements the line-following and light-following behaviors, as well as an **arbitration module** which routes the motor commands from the line-following behavior module to the motors whenever the robot is not pushing a puck. Notice that the two behavior modules have been slightly altered. They do not directly control the motors. Instead, they output left and right motor commands as global variables. The arbitration module decides which commands reach the motors. In IC, the only way to pass information between processes is through global variables.

(Note: If two behavior modules output to *different* actuators, there may be no need to arbitrate between the behaviors. For example, you could have one set of behaviors drive the robot around while another set of behaviors drive motors that keep sensors pointed in appropriate directions. These two sets of behaviors could operate independently with no interference.)

Program Listing 3: Line-following and light-following behaviors with arbitration

```
/* GLOBALS */

/* Sensory Registers */
int left_nose, center_nose, right_nose;
int left_eye, center_eye, right_eye;
int puck_sense;

void sensor_module()  /* Read sensors into global variables at 10 Hz */
{
  while(1) {
    left_nose   = analog(0);
    center_nose = analog(1);
    right_nose  = analog(2);
    left_eye    = analog(3);
    center_eye  = analog(4);
    right_eye   = analog(5);
    puck_sense  = analog(6);
    wait(100);    /* Wait 100 msec */
  }
}


/* globals */
int line_follow_left, line_follow_right;

void line_following_behavior()
{
  while(1) {
    if (left_nose < center_nose && left_nose < right_nose) {
      line_follow_left = 0;
      line_follow_right = 100;
    }
    else if (right_nose < center_nose && right_nose < left_nose) {
      line_follow_left = 100;
      line_follow_right = 0;
    }
    else {
      line_follow_left = 100;
      line_follow_right = 100;
    }
  }
}

/* globals */
int light_follow_left, light_follow_right;

void light_following_behavior()
{
  while(1) {
    if (left_eye > center_eye && left_eye > right_eye) {
      light_follow_left = 0;
      light_follow_right = 100;
    }
    else if (right_eye > center_eye && right_eye > left_eye) {
      light_follow_left = 100;
      light_follow_right = 0;
    }
    else {
      light_follow_left = 100;
      light_follow_right = 100;
    }
  }
}

void behavior_arbitrate()
{
  while(1) {
    if (puck_sense < 127) {
      motor(0,line_follow_left);
      motor(1,line_follow_right);
    }
    else {
      motor(0,light_follow_left);
      motor(1,light_follow_right);
    }
  }
}

void main()
{
  beep();
  start_process(sensor_module());
  start_process(line_following_behavior());
  start_process(light_following_behavior());
  start_process(behavior_arbitrate());
}
```

See Figure 1 for a graphical representation of this program. It often helps to draw these box-and-arrow diagrams when writing behavior programs. Each box represents a module, or process. Each arrow represents some information that is passed. This information usually takes the form of a global variable.

You may notice that the line-following and light-following behaviors are both always actively running in the background, consuming processor time. This may strike many as a wasteful programming practice since only one behavior is "active" at a given time. However, building robot programs in **parallel** by using many multitasked processes pays off in the long run. Additional behaviors can easily be added in parallel with minimal modifications to the arbitration module. Also, if you are worried about wasting processor time just remember that few robots need to get new sensory information or update actuators at greater than about 10 Hz. We are using microprocessors that run at 2 MHz.

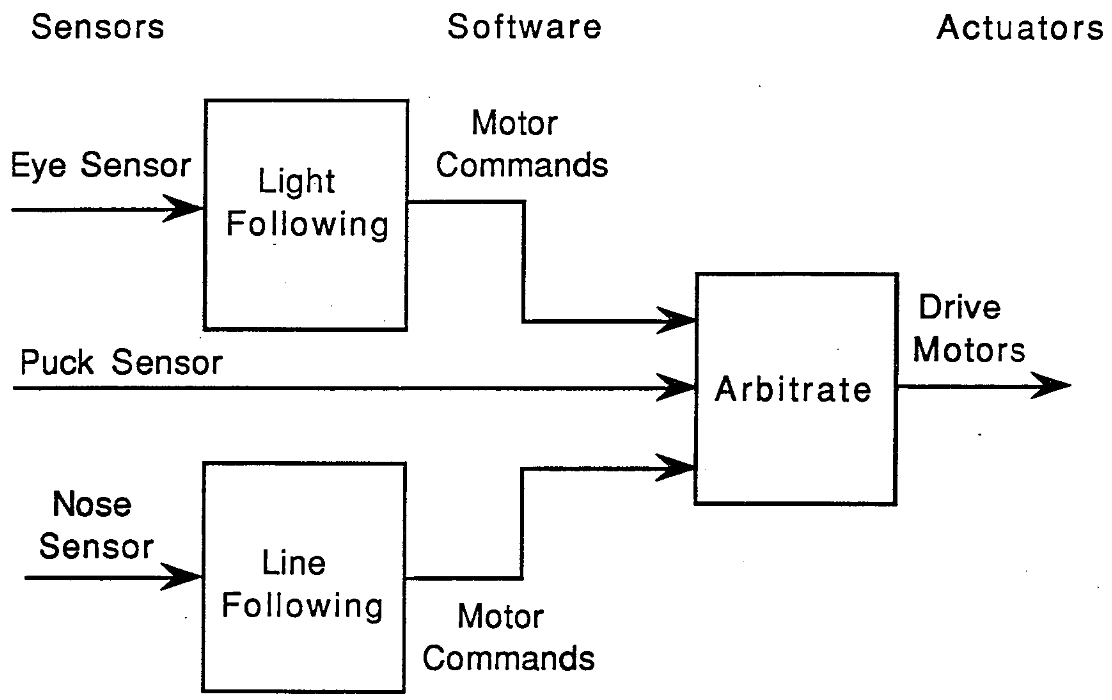

Figure 1 Graphical representation of behavior program.

University of Florida
EEL 5934

2 November, 1994
Robot Programming

Keith L. Doty
Professor EE
Reid Harrison

WHY `wait` IS BETTER THAN `sleep`

The IC library `lib_rw10.c` contains a function called `sleep(int msec)` that causes a delay of `msec` milliseconds. This function is not good to use in processes that will be multitasked, because it actively "counts time" to produce the delay. The following listing implements a superior function.

```
void wait(int milli_seconds)
{
  long timer_a;

  timer_a = mseconds() + (long) milli_seconds;
  while(timer_a > mseconds()) {
    defer();
  }
}
```

The `wait` function avoids a "busy wait" by looking at the system clock. If the clock has not reached a sufficient count, a `defer()` is issued. The `defer()` function tells the multitasking executive to skip to the next task without wasting any more time on this one. It lets multitasking programs make much more efficient use of time.

If you are using IC, you should add this command to your programs, or to your personal library. Notice that the argument passed to wait is the time in milliseconds, not seconds.

CRITTER

COMPUTER Motorola MC68HC11 (256B RAM, 512B EEROM)

LANGUAGE Assembly

SENSOR SUITE Six CdS light sensitive resistors. Two contact whiskers

No.	Sensor Type	Range	Precision	Function	Location
2	Contact whisker	100mm	NA	Object detection	Front
3	CdS Photoresistor	5mm	NA	Dark path follow	Front-down
3	CdS Photoresistor	NA	NA	Follow light above ambient	Front-forward
1	Microswitch	NA	NA	Disk contact detect	Front

ACTUATION Gearhead DC motors on each wheel.

BEHAVIORS

Follows black lines. Captures black disks. Seeks light with captured disk. Whiskers for obstacle detection. Backs up and turns a random angle when whiskers touched.

FUNCTION Anibot (Animal-Robot)

Follows dark trail, gathers disks found on trail, deposits them at a light source.