8
Implementation

Abstract notions about how to design software are all well and good, but at some point, you have to write actual code. This chapter presents one comprehensive example of how to implement a behavior-based robotic system. The system is a basic one, but it incorporates all the software elements needed by any behavior-based robot. These components include: a scheduler, a coherent method for constructing behaviors and specifying their priority, a scheme for connecting behavior outputs, and an arbiter. We begin by considering the robot's abilities and structure then show how its behavior-based system is implemented.

RoCK Specifications: The Goal of a New Machine

RoCK, an acronym for Robot Conversion Kit,[1] is a self-contained electronics and sensor package designed to convert any of a wide variety of RC cars into a robot. See **Figures 8.1** and **8.2**. The purpose of RoCK is to provide users with an inexpensive path to

[1]Ben Wirz and I submitted RoCK as our entry into *Circuit Cellar* Magazine's Design Logic 2001 contest. (We received a runner-up award.) For a more complete story of RoCK, including the electrical design, please refer to the series of four articles we wrote for the April through July 2002 issues of *Circuit Cellar*.

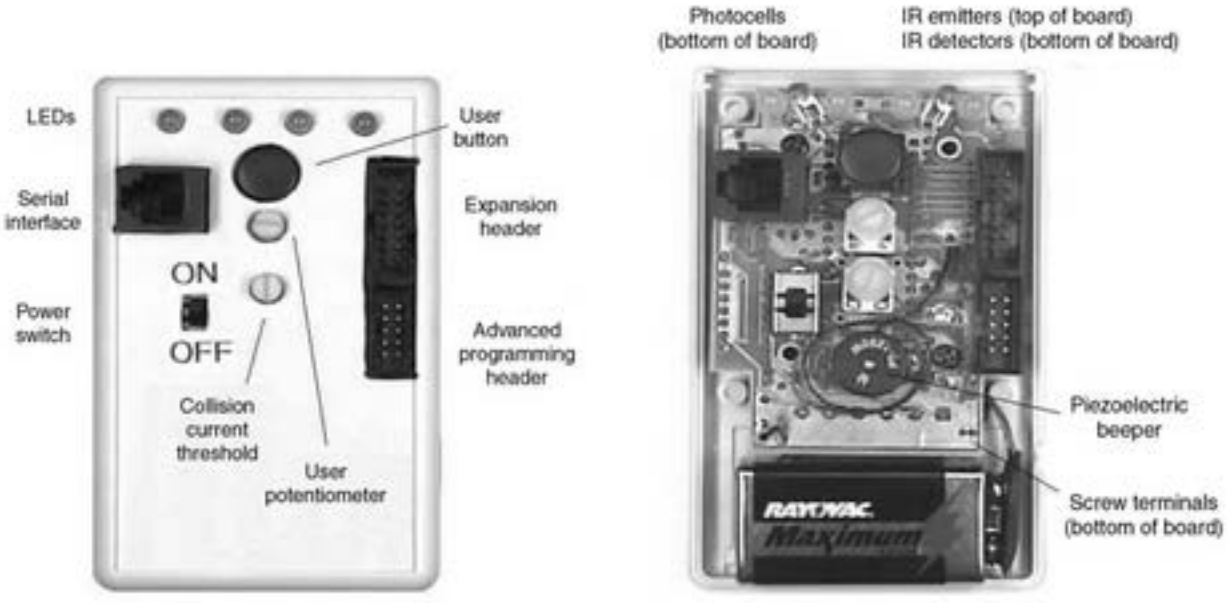

Figure 8.1

The RoCK electronics package is built into a standard plastic enclosure. RoCK's external appearance is shown on the left; on the right, the top has been removed, revealing the internal electronics board. Four LEDs in a row across the top illuminate to indicate RoCK's status. The user potentiometer allows the user to select which task RoCK will perform and to alter the value of behavior parameters.

building a robot and to demonstrate the power and versatility of behavior-based robotics.[2] RoCK comes complete with a number of built-in, user-selectable tasks. Many of the behaviors you have seen in BSim have rough analogs programmed into RoCK's firmware.

RoCK is based on Atmel's versatile and powerful AVR AT90S8535 microcontroller. Built into the 8535 are analog to digital converters (ADCs), a comparitor, and three timer/counters. The microcontroller also contains 8kB of FLASH program storage, 512B of SRAM, and 512B of software-programmable EEPROM. The chip uses the "Harvard" architecture with separate memory spaces for data and program storage. We programmed RoCK in the C language (plus a small amount assembly code) using the IICAVR cross compiler from ImageCraft.

[2]The robots that RoCK enables are in many ways reminiscent of the Braitenberg vehicles illustrated in Figure 1.10.

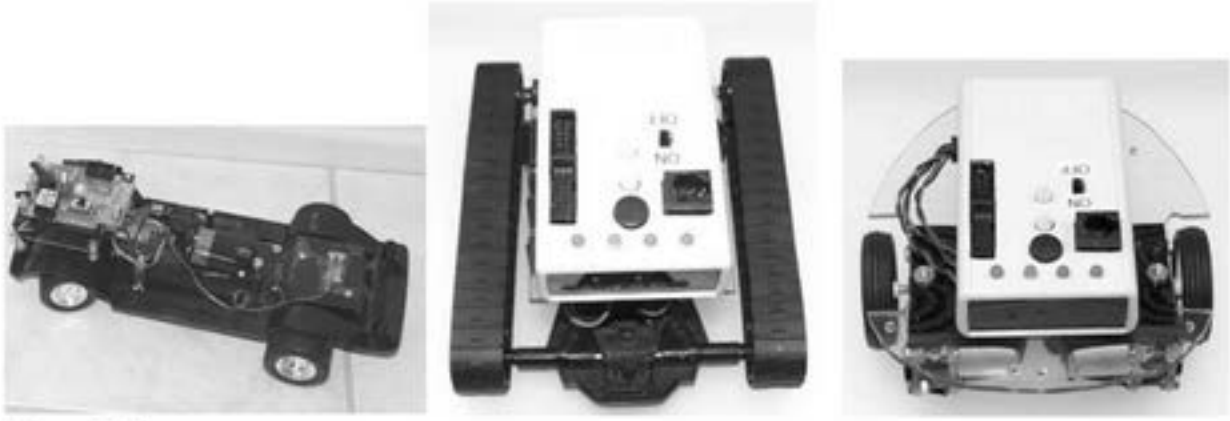

Figure 8.2

RoCK can be used to control a variety of mobility bases. On the left, RoCK's circuit board has been attached directly to a drive/steer base. The center photo shows the RoCK module attached to a differentially-steered tracked base. On the right, a home-brew chassis made from polycarbonate and motors scavenged from an old RC car forms RoCK's mobility base.

Burned into RoCK's FLASH-based firmware are 10 preprogrammed tasks plus two utility tasks. Users can program and store up to four additional tasks of their own. When a user presses the select button and twists the user potentiometer, RoCK can be commanded to perform any one of these 16 total tasks. RoCK's piezoelectric beeper can play user-programmed tunes of over 100 notes. Robot trajectories composed of more than 100 path steps can be stored and executed. RoCK has sensors to detect light, obstacles, collisions, and battery voltage. Connecting RoCK to a host computer enables the user to monitor the robot's sensors and to watch and set the robot's parameters. Despite its high level of functionality, RoCK remains an inexpensive device.

RoCK can be connected to a host computer for programming and monitoring, but the on-board user interface enables RoCK to run without being wired to a host computer. A power switch controls both the nine-volt logic supply incorporated into the RoCK module and the motor battery supply attached to the mobility base. A block diagram of RoCK's systems is shown in **Figure 8.3**.

The illumination pattern on RoCK's four LEDs indicates robot status. During task execution, LEDs generally display the state of the left and right obstacle detectors and the status of the colli-

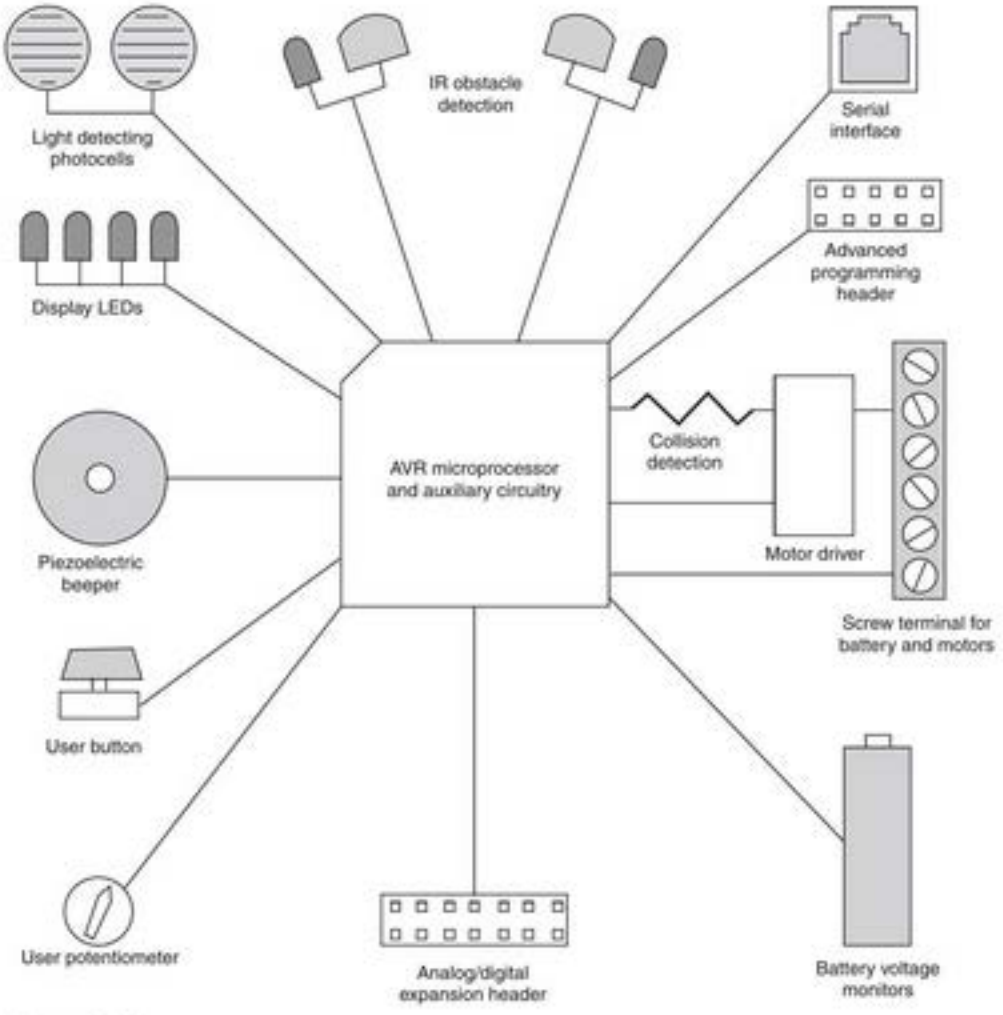

Figure 8.3

This functional diagram graphically depicts key elements of RoCK's hardware. RoCK's sensors include dual photocells, dual IR obstacle detectors, motor-stalled-based collision detection, and battery voltage monitors. The user interface consists of two inputs: the user button and user potentiometer, and two output devices: four status LEDs and a piezoelectric buzzer. RoCK can be connected to a host computer for purposes of monitoring and programming through a serial line and advanced programming header. RoCK can control two drive motors connected via the screw terminal.

sion detector. During task selection, the LED pattern indicates which of the built-in tasks the user has chosen.

The piezoelectric beeper can produce tones of arbitrary frequency, enabling RoCK to play built-in or user-programmed tunes.

Additionally, the state of certain sensors can be mapped into the frequency played by the beeper.

The user potentiometer enables the user to experiment with the parameters that control the robot's behaviors. For example, in a user task, the gain parameter in the control loop that enables the robot to follow a bright light can be mapped to the user potentiometer. The user can then adjust the robot's light-following response from sluggish to neurotic. The user button works in conjunction with the user potentiometer to select the built-in tasks the user wishes to run.

RoCK incorporates a dual-channel IR obstacle detection system. This system is composed of two series-connected emitters and two independently wired IR receivers. The receivers are sensitive to a 38-kHz modulation frequency. The emitter/detector pairs point diagonally outward from RoCK in such a way as to cover the area in front of the robot. Each receiver detects IR radiation reflected from nearby objects in the direction the detector is pointed, informing the robot of imminent collisions.

Dual diagonally mounted photocells measure the light level in front of RoCK. The difference between the light measured by the left and right photocells enables RoCK to home on a bright light source or speed to a dark corner.

As the robot operates, the logic and motor batteries discharge, decreasing the voltage of each battery. RoCK monitors the voltage of these two batteries, allowing the robot to take action if the voltage falls too low.

As we have seen in earlier chapters, an instrumented bumper provides an important means of collision detection. The mechanics of such a system are, however, rather complicated. RoCK avoids that complexity by using only a motor-stalled-based, one-bit collision detector. A calibration potentiometer is provided to adjust the trip point of this sensor to a value appropriate to the particular motors to which RoCK is attached.

Programming Specifications

A block diagram of RoCK's software architecture is presented in **Figure 8.4**. Using data provided by the task boxes on the left, the Task selector configures the behavior priorities used by the arbiter. The Task selector can also change the value of the various behavior parameters. The beeper is controlled in a way specified by the particular task. The User Tasks (the tasks programmed by the user) operate in a way identical to the other tasks, except that their specifications are stored in EEPROM, rather than flash memory. The scheduler runs one behavior after another and keeps the sensor values updated.

In developing RoCK, we first chose the set of tasks we wished the robot to perform and then created the primitive behaviors to support those tasks. (**Table 8.1** outlines RoCK's tasks.) In the code, we specify a particular task by choosing a list of behaviors, ordering their priorities, and choosing values for behavior-relat-

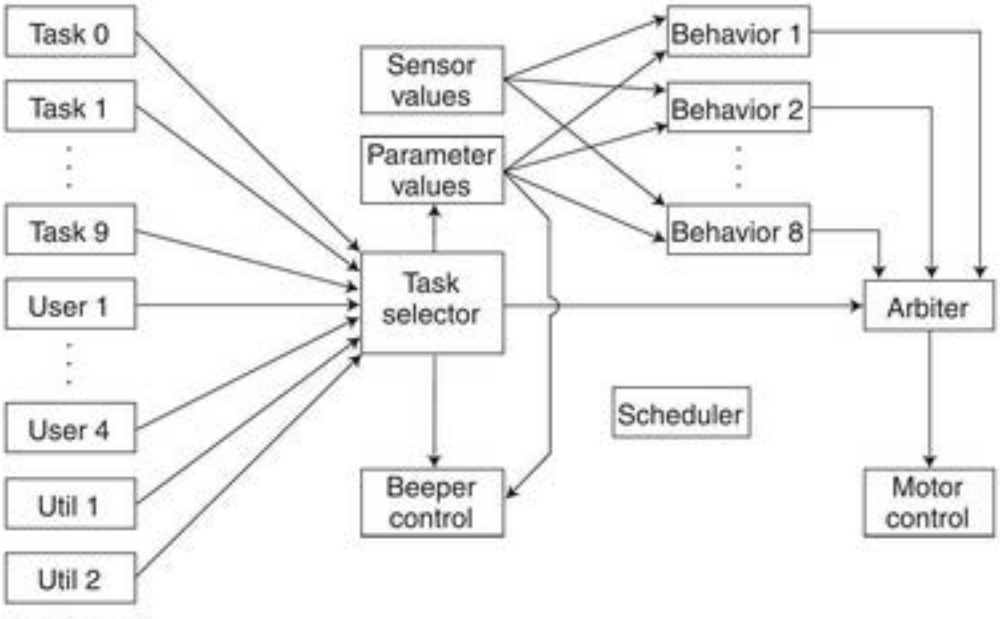

Figure 8.4

This simplified diagram shows the structure of RoCK's software. The diagram includes the familiar set of behaviors connected to an arbiter, but adds some functionality in the form of the Task selector. The Task selector makes it possible for the user to command the robot to run different tasks. The scheduler provides the "main loop" that in turn runs all the other pieces of software.

ed parameters. The reusability of these components enables us to store many tasks in the robot.

Table 8.1 These 16 named tasks can be selected via RoCK's on-board user interface. The left-most column indicates the number displayed in binary by RoCK's LEDs during task selection.

LED	Task	Description
0	Theremin	The Theremin task is the only task that does not make the robot move. This task uses the photocells and the beeper to simulate the Theremin, an early electronic musical instrument. The difference in the light level falling on the left and right photocells controls the frequency of sound emitted by the beeper.
1	Waltz	Waltz makes the robot move in a programmed pattern while a tune plays on the beeper. The user potentiometer controls robot speed. Like most tasks, Waltz specifies the Escape behavior as its highest priority behavior. This means that if the robot bumps into something while Waltzing, the robot will attempt to extricate itself before continuing with the dance.
2	Wimp	The Wimp task gives the robot a shy personality. The robot sits motionless until an object moves close enough to be sensed by the IR detectors. When this happens, the robot backs away.
3	Schizoid	Schizoid gives the robot a nervous personality. The robot cruises in a straight line, occasionally spinning or turning randomly. The interval between these events is controlled by a parameter. Increasing the parameter makes the robot seem calmer; decreasing the parameter makes the robot more frantic.
4	Pounce	The Pounce task has the robot sit in one spot until something comes near enough to trip the IR

continued on next page

Table 8.1 These 16 named tasks can be selected via RoCK's on-board user interface. The left-most column indicates the number displayed in binary by RoCK's LEDs during task selection. *(Continued)*

LED	Task	Description
4	Pounce	detectors. The robot then races full speed forward until it collides with the encroaching object.
5	Moth	The robot uses the difference in the left and right photocells to servo toward the brightest light. The robot will thus respond as if it were a moth homing on a flame.
6	Mouse	The Mouse task enables the robot to follow walls. Using its IR detectors, the robot cruises along a wall, going through doorways and turning away from inside corners.
7	Chicken	In this task, the robot uses its IR sensors to play chicken. The robot travels along at high speed, turning away just before colliding with objects it encounters.
8	Roach	When the room is dark, RoCK waits patiently. But when the light is switched on, RoCK races toward a dark spot. Safely concealed in the shadows, RoCK comes to a halt.
9	Remote	The Remote task allows direct control of the robot by the user when the robot is connected to the host computer. Remote is implemented using only the Joystick behavior. Thus the user is responsible for preventing collisions.
10	User 1	User-programmable task number 1
11	User 2	User-programmable task number 2
12	User 3	User-programmable task number 3
13	User 4	User-programmable task number 4
14	DiffSel	Helps the user correctly connect a differential drive base
15	SteerSel	Helps the user correctly connect a drive/steer base

RoCK's Behaviors

All twelve of RoCK's built-in tasks (numbers 0 through 9 plus 14 and 15, as listed in **Table 8.1**) and all four of RoCK's user-programmable tasks can be constructed from the set of eight primitive behaviors listed in **Table 8.2**.

Table 8.2 Each of RoCK's primitive behaviors can have a number of parameters that affect the behavior's output. Most parameters are unique to a particular behavior; the global speed parameter, however, is used by a number of behaviors.

Index	Behavior	Parameters
1	Dance	dance_tune_index, tempo, speed
2	IR_follow	speed, nine gain parameters (see text)
3	VL_follow	speed, nine gain parameters (see text)
4	Boston	time_between_events, event_duration,
5	Cruise	speed, turn_angle
6	Escape	backup_dist, spin_dist
7	Joystick	turn_angle, speed
8	Wire	

Dance

The Dance behavior causes the robot to move according to a stored program. A dance is a sequence of motion commands that specify a relative turn angle and the duration of motion at that angle. A motion command is stored in a single byte using the format: [tttfffff], where ttt is three bits of duration information and fffff is five bits of angle information. Dance has one parameter, dist_factor. For each motion step, Dance multiplies dist_factor by the duration bits to compute the length of time the robot holds the associate turn angle. One dance program is built into RoCK, but the user can program and store into EEPROM a second dance of over 100 steps.

IR_follow and VL_follow

The IR_follow and VL_follow behaviors are very similar in operation. The former computes robot motion using data from the left and right IR sensors; the latter uses information from the left and right photocells. These motion computations are made using a very general technique that provides an arbitrary linear mapping from a two-degree-of-freedom (DOF) input into a two DOF output. The technique also computes whether to request control of the robot. Both behaviors work in a manner fully analogous to the general linear transform we saw in the section, "Generalized Differential Response," in Chapter 5.

Boston

Periodically, the Boston behavior causes the robot to swerve randomly. Most of the time, Boston outputs no motion commands. Occasionally, however, Boston specifies an arc, rotation, or other motion. The parameter time_between_events specifies the number of seconds between such events. The event_duration parameter determines how long each motion event lasts. A random factor is included in the motion computation to make the behavior less regular.

Cruise

Cruise ignores all sensors and makes the robot move at a constant velocity. The behavior has one parameter pc_dir—the desired relative turn angle of the robot. Cruise is explained in more detail in subsection "Behavior Format" later in this chapter.

Escape

The Escape behavior attempts to extricate the robot after it has collided with an object. When the collision detector triggers, Escape responds by commanding the robot to back up and spin in place before releasing control. The duration of each of these steps is controlled by a parameter.

Joystick

Joystick allows the user directly to control the robot when the serial cable is connected. Robot speed and heading are obtained directly from the host computer.

Wire

The Wire behavior is a utility behavior included as a debugging tool for the user. Wire activates RoCK's LEDs and the mobility base's motors in a regular way that reveals to the user whether the mobility base is connected to RoCK in the correct way.

Beeper Control

Rather than build a second arbitration method for the beeper, we require that each task specify one of a small number of ways of controlling the beeper. The defined methods for controlling the beeper along with their descriptions are:

1. **Flash_tune**—The beeper plays a tune stored in flash memory.
2. **EE_tune**—The beeper plays a tune that the user composes and stores in EEPROM.
3. **Photocell difference**—The beeper behaves like a theremin—the beeper's output frequency is a function of the difference in light intensity sensed by the two photocells.
4. **Photocell sum**—Beeper frequency is a function of the total light intensity: the brighter the light, the higher the frequency.
5. **Bumper**—The beeper beeps in monotone when a collision occurs.
6. **IR_detect**—The beeper frequency depends on the IR detector: no sound for no detection, and different tones for detections by the left, the right, and both detectors.
7. **None**—The beeper is silent.

The Code

RoCK attempts to pack a great deal of functionality into a small code space. The robot has neither the room nor the need for an operating system. Given that we are flying solo, as it were, the first choice to make is the type of scheduler.

We think of all RoCK's software components (shown in **Figure 8.4**) as running in parallel. If this were actually the case, there would be no need for a scheduler. The scheduler creates the appearance of parallelism on a microprocessor whose operation is fundamentally serial.

RoCK implements a type of parallelism known as cooperative multitasking. Cooperative multitasking makes the scheduler's job especially easy. In this scheme, each parallel element (each behavior, the Arbiter, the utility functions that update sensor values, and so on) runs for a brief time, then returns control to the scheduler. The scheduler then calls the next element. Thus RoCK's scheduler consists of a list of subroutines that are called endlessly one after another.

The low overhead of a cooperative multitasking system makes it expeditious. On average, RoCK gives each behavior a chance to run over 500 times each second! One downside to cooperative multitasking is the discipline it requires of the programmer. Each element must be designed to compute for only a short time, then return control to the scheduler. A behavior that runs too long will freeze out all the other behaviors.

Scheduler

Here is RoCK's cooperative scheduler:

```
void main(void) {   // The scheduler is implemented by the main function
  // Decelerations
  extern unsigned char sensor[];
  extern unsigned char winner;        //Stores the ID the winning behavior
  extern unsigned char motor_bat_OK;
  // Initializations
  init_multi();
```

```
init_params_flash(); // Initialize default parameter values
init_serial();          // Connection with host
drive_config = EEPROMread(DRVSTR_CONFIG_ADDR); //Stored drive config
task_index = EEPROMread(TASK_INDEX_ADDR);
init_tasks(task_index);  // On startup pick the task to run
SEI();                   // Enable interrupts
// Tell the host where the sensor and parameter arrays are in RAM
//              EEPROM address        EEPROM data
EEPROMwrite(EE_SENSOR_ADDR_H, SENSOR_ADDR_H);
EEPROMwrite(EE_SENSOR_ADDR_L, SENSOR_ADDR_L);
EEPROMwrite(EE_PARAMS_ADDR_H, PARAMS_ADDR_H);
EEPROMwrite(EE_PARAMS_ADDR_L, PARAMS_ADDR_L);
wakeup();               // Play wakeup tune and show behavior selection
//The Main Loop — Read sensors, run behaviors, and arbitrate
while(1) {
 acquire_sensors();              // Read all analog and digital sensors
 p_arr[px_frob].u = user_pot;    // Move data to selected parameter
 eewriter();        // Maybe host wants to write to EEPROM
 choose_task ();  // Maybe user wants to pick a different task
 buzzer_beh(pb_select);  // The "behavior" that controls the buzzer
 // Give each behavior a chance to run:
 cruise();          // Move at a constant selected velocity
 joystick();        // Let the user control the robot
 IR_follow();       // IR_follow behavior
 vl_follow();       // Visible Light follow behavior
 escape();          // Escape from bump behavior
 boston();          // Randomly do something
 dance();           // Move to music
 wire();            // Maybe help user to connect the motors
 winner = arbitrate();  // Find highest priority beh that wants ctl
 // LED control
 if (choose_halt)        // If user is choosing a task
   leds(led_var);        //   Display the current choice
 else if (winner < WIRE_ID) {          // The normal situation
   led(0x01, (IR_detect & IR_LEFT ));  // Left IR sees an object
   led(0x08, (IR_detect & IR_RIGHT));  // Right IR sees an object
   led(0x02, !motor_bat_OK);           // Glow if motor bat too low
   led(0x04, bump);                    // Declared bump
```

```
    }
    move_dual(winner);                    // Dif drive or drive steer
}}
```

A lot is going on in this section of code, but the key item to note is the while-loop that contains calls to each of the primitive behaviors. Each iteration of this loop begins with a call to the subroutine that gathers sensor data. Subsequently, each behavior is called in turn and given a chance to run. The fact that the behaviors run in the order shown in the code has no bearing on behavior priority. After all the behaviors have been called, the arbiter runs and picks the winning behavior. The last statement in the loop is the call to move_dual. This routine sends the motion command computed by the winning behavior to the robot's motors. Nothing especially mysterious or magic is taking place.

Behavior Format

Next we'll take a look at the code that implements a couple of RoCK's behaviors. We begin with the famously simple Cruise behavior. Cruise is actually written using three macros, but to avoid confusion, let's look first at Cruise in its expanded form[3]:

```
void cruise(void) {
  unsigned char behavior_id = CRUISE_ID;
  _drive_angle(pc_dir, pg_speed, behavior_id);
  beh_ctl[behavior_id] = behavior_id; }
```

Cruise is composed of only four lines of code, but analyzing this little bit of software will take us through most of the complexity in RoCK's behavior-based implementation.

[3]In C, a macro allows us to substitute one string of text for another. When the code is compiled, the macro text is replaced (we say the macro is expanded) by the text in the macro definition. Macros are especially useful when a bit of code has to be repeated in many places with small variations at each repetition. Used in this way, macros simplify your code and reduce the likelihood of certain types of errors.

The first line creates a subroutine named cruise and declares that it has no arguments. This is expected because, as we saw in the scheduler, behaviors are implemented as subroutines. (We will designate as Cruise the behavior implemented by the cruise subroutine.)

The second line of the subroutine creates a local variable called behavior_id and assigns the value CRUISE_ID to that variable. Each behavior in RoCK must have its own unique identifier. In one of the files that compose the RoCK system, numbers are attached to behavior identifiers in this way:

```
// Behavior IDs
#define STOP_ID          0    // ID of default stop behavior
#define DANCE_ID         1    // Dance a canned dance
#define IR_FOLLOW_ID     2    // Use IRs to follow/avoid objects
#define VL_FOLLOW_ID     3    // Follow/avoid visible light
#define BOSTON_ID        4    // Do random motion at random time
#define CRUISE_ID        5    // Move at a constant speed/rotation
#define ESCAPE_ID        6    // Respond to collisions
#define JOYSTICK_ID      7    // Direct control by user
#define WIRE_ID          8    // Help the user connect motor wires
#define MAX_BEH          9    // One more than the highest behavior ID
```

Each #define statement introduces a new macro. The macro tells the compiler to replace the first text string after #define with the second string of text wherever the first string occurs in the user's program. The replacements occur before the code is processed further. Thus in the Cruise behavior, CRUISE_ID is replaced by 5 so that the local variable behavior_id is assigned the value 5. (Text following the "//" is marked as a comment and ignored by the compiler.)

So far so good, but why does each behavior need a local variable called behavior_id and why does each instantiation of that variable need a unique value? This is done because RoCK depends on these features to enable arbitration.

Every behavior computes motor control commands for the robot. In RoCK, motor control commands are stored in the form of a left

motor velocity and a right motor velocity. But clearly, behaviors must not write the value of the left and right motor velocity directly to the motor controller, because this would bypass arbitration and lead to contention between behaviors. Instead, each behavior that wants to control the motors writes its command to a particular slot in a pair of special arrays. The arrays are defined in this way:

```
int right_vel[MAX_BEH];          // Velocity array for right drive motor

int left_vel[MAX_BEH];           // Velocity array for left drive motor
```

And which slot of the right and left velocity arrays does a particular behavior write to? It writes to the slot specified by behavior_id. Each behavior has a unique local value for this variable, so that each behavior writes to a unique slot of the velocity arrays. Thus no behavior overwrites the commands of any other behavior.

The next line, line 3, of the Cruise behavior is:

```
_drive_angle(pc_dir, pg_speed, behavior_id);
```

This statement (ultimately) causes the left and right motor velocity commands to be written to the velocity arrays using behavior_id as the index. But where in this statement are the left and right velocities? There are many equivalent ways to specify the motion of a robot (see Appendix A). In this case, _drive_angle gives us a way to specify a relative heading and a velocity. The function interprets its first argument as an angle and the second as a velocity—both are parameters. The angle and velocity arguments are converted by the function into left and right wheel velocities and written into the corresponding velocity array. The angle argument, pc_dir, tells the function how tightly to turn the robot. When this argument is 0, the robot moves straight ahead—the velocities of the left and right wheel motors are both equal to pg_speed. When the angle argument is 180 degrees, the robot moves backward. When the argument is 90 degrees, the robot spins in place to the left; –90 makes the robot spin to the right, and so on.

The final statement in the Cruise behavior is:

```
beh_ctl[behavior_id] = behavior_id;
```

By this point, the behavior has computed motor commands for the robot. But is Cruise triggered or untriggered? Should the arbiter pass to the motor controller the velocity commands that Cruise has computed or not? By writing its own (non-zero) identification number to the beh_ctl array, Cruise informs the arbiter that the motor commands should be delivered. Had Cruise not wanted control, it would have executed the statement:

```
beh_ctl[behavior_id] = 0;
```

Were you to look at the file that instantiates RoCK's Cruise behavior, you would find it written not as shown above, but rather in this way:

```
void cruise(void) {
  behavior(CRUISE_ID);                    // Declare behavior_id
  drive_angle(pc_dir,pg_speed);
  CONTROL; }
```

This version operates exactly as described above, but here we have used three macros to hide the details of the implementation. The second line automatically creates local variable behavior_id and assigns to it the value CRUISE_ID. The third line is a macro that expands to the _drive_angle function. The final line becomes beh_ctl[behavior_id] = behavior_id;

We'll take a quick look at one more behavior, the Boston behavior. This behavior, implemented as an FSM, causes the robot to make a random motion at a random time.

```
void boston(void) {
  behavior(BOSTON_ID);
  static int angle;
```

```
  static unsigned char state = 0;       //The state of the FSM
  extern unsigned int  clock;
  static unsigned int  del_time;
  static unsigned int  b_time;
  switch(state) {
    // State 0 - Compute time until random event
    case 0:
      b_time = clock;                          // Starting time
      del_time = ((rand()>>10) * (unsigned int) pr_time);
      PUNT;                                    // Don't try to control robot
      state++;                                 // Go to next state
      break;
    // State 1 - Wait for event to start
    case 1:
      if (time_out(b_time, del_time)) {        //If done act, else remain
        b_time = clock;                        // New timeout time
        del_time = ((rand()>>7) * (unsigned int) pr_dur); //
        angle = rand()>>7;                     // Pick a random angle
        drive_angle(angle,pg_speed);           //Macro hides array details
        CONTROL;                               // Try to control robot
        state++;                               // Go to next state
      }
      break;
    // State 2 - Random heading until timeout
    default:
      if (time_out(b_time, del_time))
        state = 0;
      break;
    }
}
```

Boston has the same basic format as Cruise. The statement, behavior(BOSTON_ID), appears early in the subroutine to give Boston its unique identifier and declare the behavior_id local variable. Second, the code calls drive_angle in various places when it wants to send velocity commands to the motors. And third, PUNT and CONTROL are called to inform the arbiter of Boston's desires. (PUNT writes a zero into Boston's slot in the

beh_ctl array.) Boston uses C's switch function to implement an FSM. In state zero, Boston figures out how long to wait until the next random motion, then advances to state one. Boston waits in state one until it is time for the event to occur. At the proper moment, the random motion is commanded, the time when the motion should stop is computed, and control passes to the next state. Boston remains in the final state, state two, until the event finishes; then control goes back to state zero.

Here is one final example of a RoCK behavior—RoCK's version of Escape:

```
// Escape – A ballistic behavior executed when a collision
// is detected. The robot will backup and turn before releasing
// control. Parameters in units of ~mS *6 => max time = 256 *0.016
// => 4.096 seconds. That is, each unit of pe_x is 16 mS.
// Use motor current to determine a bump. Backup and spin if detected.
void escape(void) {
  behavior(ESCAPE_ID);                          // Declare behavior_id
  extern unsigned char sensor[];
  static unsigned char state;
  extern unsigned int  clock;
  static unsigned int  e_time;
  static unsigned int  duration;
  switch(state) {
    case 0: if (bump) {                         // There is now a collision
              e_time = clock;
              duration = ((unsigned int) pe_backup) << 4;
              drive(-pg_speed,-pg_speed);// Backup fast
              state++; }                        // Advance to next state[4]
            else
              PUNT;                             // No bump => release
            break;
    case 1: if (time_out(e_time, duration))  {
              e_time = clock;
```

[4]Why is there no CONTROL statement in this section of code? None is needed here because on the next iteration, the behavior will be in the next state (case 2) and this state does contain a CONTROL statement.

```
                duration = ((unsigned int) pe_spin) << 4;
                drive(-pg_speed,pg_speed);  // Spin in place
                state++; }
            CONTROL;
            break;
    default: if (time_out(e_time, duration))
                state = 0;
            CONTROL;
            break;
    }
}
```

Most structural aspects of this behavior should now be familiar. The RoCK robot does not have a differential bumper; thus RoCK can detect only that a collision has occurred, it does not know on which side of the robot the collision happened. Escape determines that a bump has occurred by examining the global variable bump (whose value is either zero or one). Also, RoCK lacks shaft encoders, meaning that it cannot determine how far it has moved. All that RoCK can know is that it commanded a certain velocity and that enough time has passed to allow the robot to move the desired distance. Escape therefore decides that it has backed up and spun far enough based on time parameters pe_backup and pe_spin.

Arbiter

RoCK refers to arbiter by the name arbitrate. There are two features peculiar to RoCK that make arbitrate a little more complex than it might otherwise be. RoCK is not programmed to do just do one task; it can perform any of 16 different tasks. A task may use some, but not other behaviors. Each task can prioritize the primitive behaviors that it does use in any way. A task can also set global parameters and parameters associated with the primitive behaviors in an arbitrary way. A second complication of RoCK, compared to robots that do just one task, is that the user's selection of a new task is treated outside the arbitration system.

RoCK's arbiter is defined in this way:

```
int arbitrate(void) {
  unsigned char pri_index;       // Check each index
  unsigned char beh_index;       // Get the behavior index stored there
  // Step through the behaviors in the beh_pri array
  if (choose_halt)          //Don't arbitrate if user is choosing a new task
    return 0;                       // This forces the STOP "behavior" to win
  else {
    for(pri_index = 0; pri_index < MAX_BEH; pri_index++) {
      beh_index = beh_pri[pri_index];    // Get slot # of next hst beh
      if (beh_ctl[beh_index])             // If this behavior wants control
        return beh_index;                 // ...return it as winner
    }
    return 0;                // No behavior wants control, so STOP
  }
}
```

Unless the user is in the process of selecting a new task, the "for" loop runs. This loop steps through the possible priorities. If there are *n* primitive behaviors, then there must necessarily be *n* different priorities. Were there fewer priorities than behaviors, then two or more behaviors could have the same priority—this would lead to an ambiguity.

Arbitrate uses two levels of indirection,[5] as shown in **Figure 8.5**. The beh_pri array contains the identification numbers of the primitive behaviors that implement a particular task; these numbers are ordered according to index number. That is, the identifier of the highest priority behavior is stored in beh_pri[0], the identifier of the second highest priority behavior is stored in beh_pri[1], and so on.

Arbitrate steps through the elements of beh_pri one at a time, assigning the identifier to the variable beh_index. Arbitrate then uses beh_index as an index for examining the beh_ctl array. If the value of beh_ctl[beh_index] is zero, then the behavior correspond-

[5]An indirect reference means that the value we are interested in is not stored in the referenced location. Rather, stored at that location is a pointer to another location, where the value (or in RoCK's case another pointer) is stored. Indirection can make programming structures more versatile.

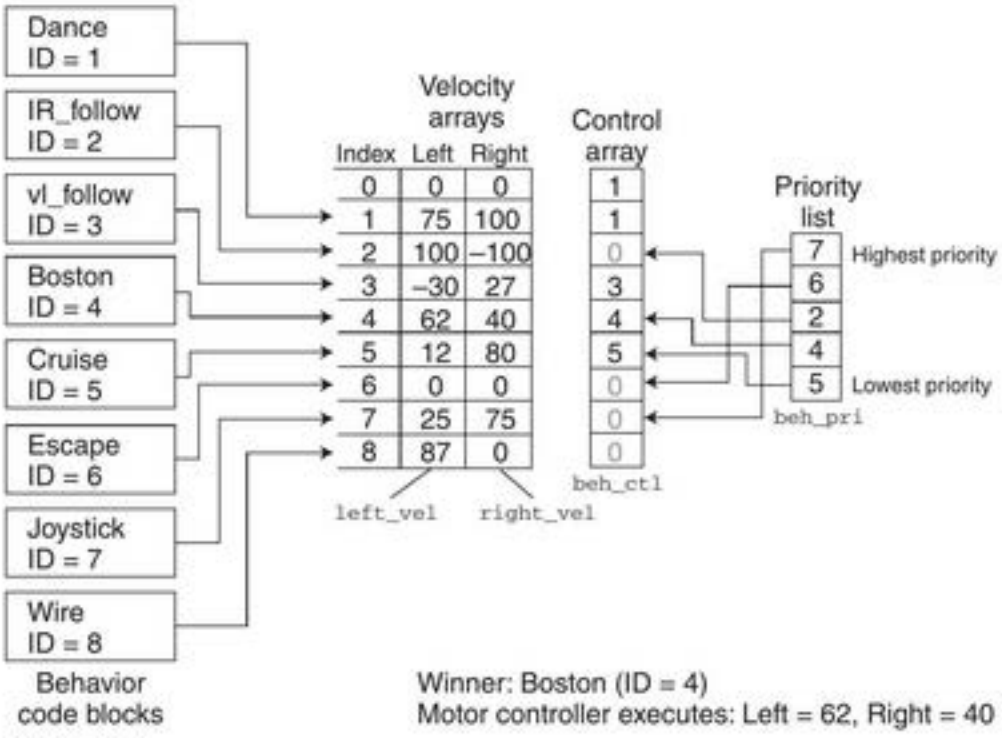

Figure 8.5

To instantiate a task, the identification numbers of the behaviors that compose the task are written into the beh_pri array. The behavior whose identification number is stored in index zero has the highest priority in this task; the behavior whose identification number is stored in index one of the beh_pri array has the second highest priority, and so on. The arbiter steps through the elements of the priority list using the numbers stored there to index into the control array, beh_ctl. Any behavior wanting control (any triggered behavior) writes its ID number into its slot of the control array. In the example, the Boston behavior has ID=4. Boston, wanting control, writes 4 into slot 4. No higher-priority behavior (Joystick, Escape, or IR_follow) is requesting control, so the arbiter ultimately checks the priority list slot containing the fourth highest priority. Finding a 4 stored in this slot, the arbiter notes that slot 4 of the control array is non-zero and thus declares Boston the arbitration winner.

ing to the identifier stored in beh_index does not want control and the "for" loop continues iterating. If the value of beh_ctl[beh_index] is non-zero, then the corresponding behavior does want control and the loop is exited with the index of the highest-priority behavior that does want control (the winning behavior) stored in beh_index. Arbitrate returns this value.

In RoCK, a small set of data represents a task. The set contains information for ordering the priorities of the primitive behaviors,

values for relevant behavior parameters, and a mapping from the user potentiometer to one of the parameters (this gives the user real-time control over one parameter). The code that specifies RoCK's built-in behaviors is contained within an initialization routine; data for instantiating user-programmed behaviors is burned into EEPROM. The following block of code from RoCK's startup routine illustrates how built-in tasks are instantiated:

```
// Schizoid
  case 3: beh_pri[0] = ESCAPE_ID;   // Highest priority beh
          beh_pri[1] = BOSTON_ID;
          beh_pri[2] = IR_FOLLOW_ID;
          beh_pri[3] = CRUISE_ID;        // Lowest priority beh
          pi_a = -64; pi_b = -64; pi_c = 64;
          pi_d = 0;   pi_e = 0;   pi_f = 64;
          pi_g = 1;   pi_h = 1;   pi_i = 0;
          px_frob = ig_speed;            // User pot controls robot speed
          pb_select  = BUZZ_PHO_DIF; // Theremin makes it sound weird
          pr_time = 32;                  // Parameters
          pr_dur = 2;
  break;
```

Seventeen assignment statements instantiate the Schizoid task. The highest priority primitive behavior in Schizoid is Escape. Assigning Escape_ID to the zeroth slot of the priority array beh_pri[0] gives this behavior the highest priority. The other behaviors, in order of priority, are Boston, IR_follow, and Cruise. Variables that begin with a 'p' and contain an underscore are behavior parameters. Schizoid uses the general linear transform method to control the robot's response to the IR sensors. The nine variables pi_a through pi_i are the matrix elements. The other parameter settings control other aspects of the task.

The advantage of building RoCK's system as outlined here is that additions and deletions are very simple—just storing some behavior identifiers into an array and writing some values into some variables creates an all-new task. Adding tasks, deleting tasks, and modifying tasks are all very simple—just as with BSim.