# SUBSUMPTION
for the SR04 and jBot Robots

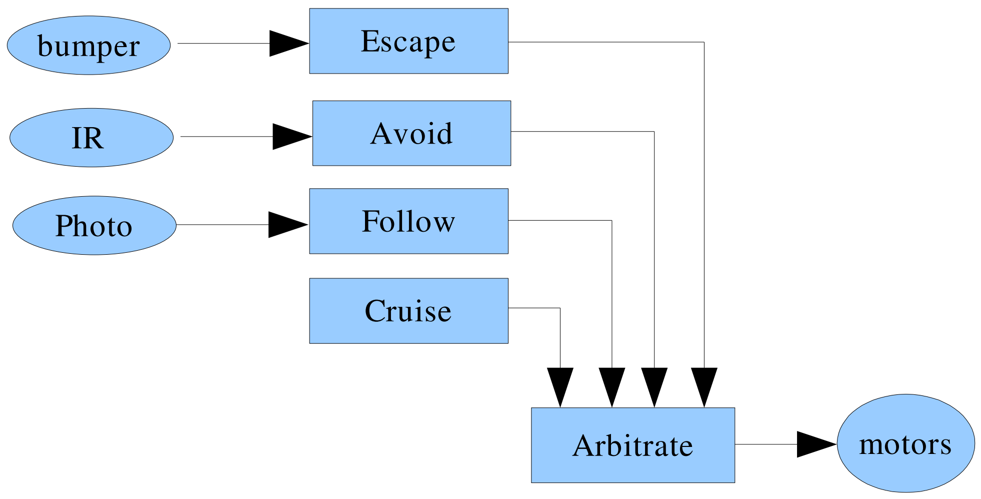


Figure 1. Subsumption diagram (Flynn&Jones)

David P. Anderson
Department of Geological Sciences
Southern Methodist University
www.geology.smu.edu/~dpa-www/myrobots.html

Motivation

This article is a compilation of a series of email postings on the Dallas Personal Robotics Group list server concerning the topic of robot programming, and particularly the technique developed by Rodney Brooks and his colleagues at M.I.T., known as subsumption.

It's not intended as an exhaustive study of the topic, but rather as an overview and context for some particular practical examples from my own robots. The two used here are the SR04 robot and jBot, although the LEGOBot runs very similar software. If you'd like to examine videos of the behaviors of these robots, the MyRobots web page has links for these and a few other robots.

Overview

A good starting point might be to read Chapter 9, "Robot Programming," of Flynn & Jones' *Mobile Robots* first edition, and Chapter 4, "Arbitration," of Jones' *Robot Programming: A Practical Guide*. I believe the DPRG library has copies of both. For a more general discussion, see *Cambrian Intelligence*, an important collection of papers that Brooks published through M.I.T. Press.

Following on from questions and feedback from the DPRG list server, this article begins by examining several different subsumption arbitration schemes from published literature and practical examples.

I. Basic primitive tasks

Most automation basic primitive tasks take on something like the following generic form:

```
void task()
{
    while (1) {

        read_sensor();
        calculate_output();
        output_control();

        delay(ms);
    end
}
```

This is an endless loop that executes a certain number of times per second, as determined by the "ms" parameter. Most of my current robots have a cycle time of 20 Hz, so ms = 50.

For most tasks, the execution time of the read/calculate/output is insignificant, measured in microseconds or less, compared to the time spent in the delay(), measured in milliseconds or greater, as the following ASCII cartoon attempts to illustrate:

```
 r/c/o             r/c/o             r/c/o             r/c/o
  __                __                __                __
  ||                ||                ||                ||
  ||                ||                ||                ||
  ||      delay     ||     delay      ||    delay       ||    delay
__||________________||________________||________________||________________etc

time -->

(not to scale)
```

The delay is, of course, where the other tasks run. This can be implemented with a simple cooperative multitasking kernel, or as a series of Finite State Machines Augmented with timers (AFSM), examples of each are included in the section on sample behaviors.

Of course some tasks take lots of CPU cycles, like the Kalman filter for a two wheel balancer or IMU, image processing, sonar arrays, etc. My experience is that these sorts of sensors and their associated filters are often best offloaded to separate processors, depending on the horsepower of your particular robot controller.

SR04

II. Subsumption and Behaviors

For robot tasks used in a subsumption architecture, there is an additional requirement: an active/inactive flag and its associated threshold:

```
void subsumption_task()
{
    while (1) {
        read_sensor();
        calculate_output();
        output_control();

        if (output > threshold) flag = TRUE;
        else flag = FALSE;

         delay(ms);
    end
}
```

(The word "threshold" is used loosely here, as the active/inactive test may be more complex than a simple greater-than.)

The flag signals whether the task wishes to control the robot for this cycle (50 milliseconds on my robots) or is happy with things the way they are. Every subsumption task must have a flag that is sometimes true and sometimes false except the default task, which we'll get to later, which is always true (hence "default").

As the delay() described in section I allows other tasks to execute, the flag described here allows those other executing tasks to control the robot, through the mechanism of subsumption.

III. Arbitration

Subsumption tasks are arranged by priority, from lowest to highest, as determined by the robot builder for a particular problem set. An arbitrator() selects the output of the highest priority task whose flag is true to pass on to the physical subsystem (motors or whatever) for this pass through the loop, this 20th of a second.

Low priority tasks can only control the robot when all higher priority tasks' flags are false. When any higher priority task sets its flag to true, that tasks' output controls the robot for the next 50 milliseconds, and all lower priority tasks asserting their arbitration flags are said to be “subsumed.”

A. *Mobile Robots* arbitration example

Jones and Flynn use the cooperative multi-tasking available in IC for their example. They define a group of subsumption tasks and start them up at initialization like this:

```
void main()
{
    start_process(motor_driver());        // control motor(s) speed
    start_process(cruise());              // accelerate straight ahead
    start_process(follow());              // turn toward bright light
    start_process(avoid());               // turn away from IR reflections
    start_process(escape());              // bumper collision recovery
    start_process(arbitrate());           // send highest priority to motors
}
```

These 6 tasks run asynchronously. The four behaviors, cruise, follow, avoid, and escape, signal the arbitrate process with their output flags, which passes along the commands of the arbitration winner to the motor_driver() task. See ***Mobile Robots*** for implementation details of the particular tasks.

Here is Flynn and Jones' arbitration code from ***Mobile Robots*** p264, for a scheme with these 4 tasks/layers/behaviors (pick your own terminology...):

```
void arbitrate()
{
    while (1) {
        if (cruise_output_flag == 1)
            {motor_input = cruise_output; }
        if (follow_output_flag == 1)
            {motor_input = follow_output; }
        if (avoid_output_flag == 1)
            {motor_input = avoid_output; }
        if (escape_output_flag == 1)
            {motor_input = escape_output; }
        sleep(tick);
    }
}
```

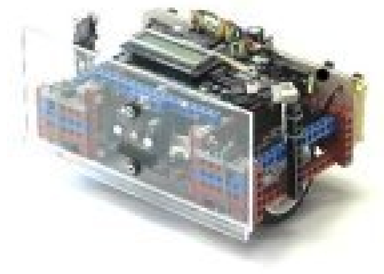
In this case the tasks are listed from lowest priority, cruise, to highest priority, escape.   The lowest priority, cruise, is the default behavior and its flag is always == 1, so this loop always begins with the motor_input set to cruise_output. If no higher priority tasks are asserting their flags, then that is the output delivered to the motors when the motor task runs during the sleep(tick) period of delay.

Note that this technique for arbitration might not work in an interrupt driven environment when the motor updates are happening asynchronously from the *arbitrate()* loop.   Randy Dumse adds:

“...a simple substitution scheme needs to be sure the substitutes are substituted before making it to the outputs, such as might happen if an interrupt routine asynchronously grabbed the results before every behavior had its chance to substitute.”

In the above example from Flynn and Jones, the use of the cooperative multi-tasking capability of the IC language provides for atomic execution of the task itself, as all other tasks run only during *arbitrate()*'s sleep(tick) period.

These behaviors and their arbitration scheme are the same as those illustrated schematically in the figure 1, at the top of the first page of this article.



LEGOBot

B. *Robot Programming* arbitration example.

Joseph Jones seems to prefer schematics over code, and his excellent *Robot Programming* book includes a number of diagrams of subsumption processes and their variations.

Here is one of Jones' examples from *Robot Programming*, fig 4.5 pp82-83, which includes a number of interesting features. (Note that the following “ASCII-ART” illustration probably only makes sense when viewed with a mono-spaced font.)

J.Jones' Robot Programming arbitration example:

```
                            +--------------+
    sensor --+------------->|              |
             |              |  Behavior A  |------------+
             |       +----->|  ID = B01    |            |
             |       |      +--------------+            |
             |       |                                  |
             |       |      +--------------+            |
    sensor -----+---------->|              |            |
             |  |    |      |  Behavior B  |----------+ |
             |  |    +----->|  ID = B02    |          | |
             |  |    |      +--------------+          | |
             |  |    |                                | |
             |  |    |      +--------------+          | |
             |  +---------->|              |          | |
             |       |      |  Behavior C  |--------+ | |
             |       |      |  ID = B03    |        | | |
             |       |      +--------------+        | | |
             |       |                              | | |
             |       |      +--------------+        | | |
             +------------->|              |        | | |
                     |      |  Behavior D  |------+ | | |
             +------------->|  ID = B04    |      | | | |
             |       |      +--------------+      | | | |
             |       |                            | | | |
             |       |      +--------------+      | | | |
    sensor --+------------->|              |      | | | |
                     |      |  Behavior E  |----+ | | | |
                     +----->|  ID = B05    |    | | | | |
                     |      +--------------+    | | | | |
                     |                          V V V V V
                     |     Winner = B02       +----------- +     +-----------+
                     +------------------------| Arbitrate  |---->| Actuators |
                                              +----------- +     +-----------+
```

Notes:

1. Jones' illustrates multiple behaviors making use of the same sensors, and reading more than one sensor.

2. He also includes a feedback line from *Arbitrate* back to the tasks. This is essentially a global variable that contains the ID of the task that “won” this round of arbitration. It can be used by the individual tasks to determine if they have been subsumed.

3. See *Robot Programming* Chapter 8, “Implementation” for more details.

C. SR04 Robot arbitration example

The **SR04** robot uses a home-brew cooperative multi-tasking kernel in addition to a single large sensor loop running at 20 Hz, to control the robot. A group of concurrent tasks, including the sensor loop, user I/O, odometry calculations, and a sound synthesizer are defined, and are started up by *main()*, like this:

```
/* ------------------------------------------------------------------------ */

main()
{
    srat_init();                          // SR04 initialize system and tasks

    create_process(trace,0,64);           // CPU load monitor
    create_process(play_note,0,128);      // FM synthesis
    create_process(sing_songs,0,256);     // music and audible warnings

    create_process(sensors,0,512);        // main sensor loop
    create_process(odometers,0,256);      // calculate robot position
    create_process(user_interface,0,512); // buttons and LCD
    create_process(reset_servos,0,64);    // retract motion detector and gripper

    scheduler();                          // start multitasking.
    printf("should never get here\n");    // scheduler() never returns.
}
```

create_process() adds a process to the multitasking queue. Its arguments are the function, initial parameter, and stack size. The process we are interested in here is the *sensors()* task, which is the primary subsumption loop controlling the robot. Here is the code and some comments for the main sensor loop on **SR04:**

```
/* ------------------------------------------------------------------ */

void sensors()                      // SR04 20 Hz sensor loop
{
    INT32 t;
    mseconds(&t);                   // set local 32 bit timer t to now

    while(1) {                      // endless loop
        if (srat_system & ARBITRATE) {

                speedctrl();         // PID
                slewspeed_task();    // velocity profiler
                prowl_task();        // seek navigation coordinates
                bumper_task();       // ballastic collision recovery
                photo_task();        // seek/avoid light
                ir_task();           // seek/avoid IR reflections
                sonar_task();        // seek/avoid sonar reflections
                motion_task();       // motion detector
                xlate_task();        // rotate and scan
                behaviors_task();    // scan for soda can profiles
                passive_task();      // jump toward/away from movement
                seek_task();         // seek/avoid IR beacon
                boundary_task();     // detect virtual walls with odometry
                feelers();           // gripper grasping reflex
                arbitrate();         // send highest priority to motors
         }
         tsleep(&t,50);              // suspend in multitasking queue
    }                                // for the remainder of 50 ms, then loop
}
/* ------------------------------------------------------------------ */
```

```
// SR04 tasks use a C struct named LAYER for their output commands:

typedef struct layer LAYER;

struct layer {
        int cmd;        /* assertion command */
        int arg;        /* assertion argument */
        int state;      /* layer state, used by AFSM tasks */
};

// SR04 has 9 subsumption behaviors, so define 9 LAYERS:

LAYER bump,motion,ir,boundary,sonar,photo,xlate,prowl,stop;

// layers[] is an array of pointers to the LAYER struct for each task:

#define LAYERS 9
LAYER *layers[LAYERS] =
        {&bump,&motion,&ir,&boundary,&sonar,&photo,&xlate,&prowl,&stop};

/* -------------------------------------------------------------- */
// The arbitration flags on SR04 are implemented as bit positions in a
// 16 bit flag word (hence only 16 possible layers), defined as follows:

#define BUMP    1
#define MOTION  2
#define IR      4
#define BOUNDARY 8
#define SONAR 0x10
#define PHOTO 0x20
#define XLATE 0x40
#define PROWL 0x80
#define DEFAULT 0x100

// These bits definition apply to a set of global masks:

int srat_flags;                 /* bit = layer asserting */
int srat_avoid;                 /* bit = invert response */
int srat_enable;                /* bit = enable this layer */
int srat_suppress;              /* bit = ignore this layer */

/* -------------------------------------------------------------- */
/* SR04 arbitration code */

int arbitrate()             // call this from sensors() loop at 20 Hz
{
        int i,j;
        i = 0;
        if (srat_system & ARBITRATE) {

                j = ~srat_suppress & srat_flags;  // suppress

                while ((j & 1) == 0) {            // subsume
                        j /= 2;
                        i++;
                        if (i == LAYERS-1) break; // default
                }

                motor_cmd(layers[i]);             // execute
        }
}
```

Note that *arbitrate()* is not an endless loop, but rather is called within the robot's *sensors()* loop, which runs at 20 Hz. The output goes to *motor_cmd()*, which takes a pointer to the winning layer and calculates motor velocities for the two wheel motors controlled by the speed control subsystem.

By passing in the actual velocity and rotation arguments, the tasks are able to generate velocity profiles such that the behaviors blend one into another for smooth and graceful maneuvering:

```
/* ---------------------------------------------------------------- */
/* Left and Right motor commands for SR04 */

int top_speed;                  // global, top speed set by user
int bot_speed;                  // global, current requested robot speed

int motor_cmd(LAYER *l)
{
        int extern Lvel, Rvel;  // Left and Right requested motor velocity
                                // these are inputs to the PID controller
        bot_speed = l->cmd;     // current requested velocity */

        Lvel = bot_speed + l->arg;   // left motor = velocity + rotation
        clip(Lvel,100,-100);        // don't overflow +- 100% full speed

        Rvel = bot_speed - l->arg;   // right motor = velocity - rotation
        clip(Rvel,100,-100);        // don't overflow +- 100% full speed
}

/* ---------------------------------------------------------------- */
```

Notes on SR04's arbitration:

1. There are nine behavior tasks on the SR04 robot. Each has an associated LAYER structure for its I/O. Pointers to these structures are grouped into an array, *layers[]*, for processing by the arbitrate task.

2. The tasks called from the SR04 *sensors()* loop each generate three outputs: a flag bit, a command, and an argument. For the platform motion control, the command is the requested velocity of the center of the robot, and the argument is the rotational velocity about the center of the robot, that is: speed and steering.

3. Tasks are enabled by setting their bit in the srat_enable global. The tasks signal the arbitrator by setting or clearing their bit in the srat_flags global. Task behavior can be inverted by setting a bit in the srat_invert global, -- for example, seeking dark rather than light. Behaviors can be suppressed by setting the appropriate bit in the srat_suppress global.

4. The tsleep() function is a special form of msleep() that takes into account the execution time of the entire loop in calculating how long to delay. So the repetition rate, (and sample rate for any included tasks) remains constant.

5. Those familiar with SR04 from the DPRG CanCan competition might notice that there is no "can collecting" layer. SR04's can collecting is an *emergent* behavior formed from a particular combination of the these tasks and their modes, and a separate feeler-actuated gripper that grabs anything it touches.

The arbitration scheme used on **jBot** is similar to the one described here for **SR04**, with a only few modifications and simplifications.

D. jBot arbitration example

jBot uses a Motorola 68332 processor running in a Mini Robominds controller. This is a full 32bit processor with the CPU32 core instruction set. The processor is more complex than the HC11 used on **SR04** and **LEGObot** but as a consequence the tasks are generally simpler to construct, and more powerful and flexible in their configuration and execution. Here's **jBot**'s startup sequence:

```
/* ---------------------------------------------------------------------- */

int     main()                  // jBot startup
{
        system_init();          // cpu, tpu, sysclock, lcd, led, sci, interrupts
        jbot_init();           // behavior inits

        fprintf(stdout,"%s\nHowdy.\n",VERSION);       // stdout = serial port
        fprintf(stderr,"%s\tHowdy.\n",VERSION);       // stderr = LCD

        create_task(trace,0,64);           // monitor CPU load
        create_task(uio,0,1024);           // user I/O
        create_task(green_led,0,64);       // flash heartbeat

        create_task(sensors,0,1024);       // main sensor loop
        create_task(odometers,0,1024);     // calculate robot location
        create_task(sonar_task,0,512);    // sonar array driver

        create_task(delayed_init,0,128);  // inits that must wait on GPS or IMU

        scheduler();                       // let's go!
        printf("Should never get here\n");

}

/* ---------------------------------------------------------------------- */
```

As was the case with the **SR04** robot, the process of interest for this article is the *sensors()* loop started by *main()*. Here is **jBot**'s *sensors()* loop:

```
/* ---------------------------------------------------------------------- */

void sensors ()            // jBot 20 Hz main sensor loop
{
    int t = mseconds();    // set timer

    while (1) {            // endless loop

        knob_task();       // on-board knob for testing
        radio_task();      // drive from R/C receiver
        xlate_task();      // rotate and scan
        prowl_task();      // seek navigation coordinates
        escape_task();     // ballistic sequences
        obstacle_task();   // sonar avoid
        perimeter_task();  // sonar perimeter following
        navmode_task();    // monitor/modify navigation mode
        arbitrator();      // pass highest priority to motors

        t = tsleep(t,50);  // suspend for the rest of 50 ms.
    }
}

/* ---------------------------------------------------------------- */
```

jBot's arbitration differs from SR04 mainly in that there is an indirect mapping between tasks and their priorities, so that different over-all priority schemes can be used to solve different problems, as Jones suggests in Fig 8.5, pp223-225 of *Robot Programming*.

The arbitration flag has been moved into the LAYER structure, and the suppress_flags are not implemented for this robot. **jBot's** arbitration also implements Jones' “arbitration winner” signal, which maintains a pointer to the highest priority task asserting its flag in the global named *this_layer*.

```
/* ---------------------------------------------------------------- */
/* jBot arbitration struct */

typedef struct layer LAYER;      // C struct for subsumption task output

struct layer {
        int cmd;                 // assertion command
        int arg;                 // assertion argument
        int flag;                // subsumption flag
};

LAYER xlate, user, radio, prowl, escape, obstacle, perimeter;

#define JOB1_SIZE 8
LAYER *job1[JOB1_SIZE] =
    {&user,&radio,&escape,&xlate,&obstacle,&perimeter,&prowl,&stop};

LAYER stop;                     // the default layer
LAYER *this_layer = &stop;      // output, layer chosen by arbitrator()

LAYER **job;                    // pointer to job priority list
int job_size;                   // number of tasks in priority list



/* ---------------------------------------------------------------- */
/* jBot's arbitration code */

int arbitrate;              // global flag to enable subsumption
int halt;                   // global flag to halt robot

int arbitrator()            // jBot arbitration code
{
    int i = 0;

    if (arbitrate) {
        for (i = 0; i < job_size-1; i++) { // step through tasks

            if (job[i]->flag) break;       // subsume
        }
        this_layer = job[i];               // global output winner

        motor_cmd(this_layer);             // send command to motors
    }
}

/* ---------------------------------------------------------------- */
```

job1 is set as the active job at startup time by code called during the robot's behavior initializations:

```
/* ---------------------------------------------------------------- */

int job1_init()                 // make job1 the active job
{
        job = &job1[0];         // global job priority list pointer
        job_size = JOB1_SIZE;   // number of tasks in job1 list
        return 0;
}

/* ---------------------------------------------------------------- */
```

Notes on jBot's arbitration:

1. Most of the notes for **SR04**'s arbitration apply to **jBot** as well.

2. Several of **jBot**'s tasks read its IMU sensor, it's Sonar Array, and it's GPS data stream. These and the quadrature encoders on the two drive motors are currently **jBot**'s only sensors.

3. At initialization the robot also starts up a series of tasks not documented in this article, which run in parallel on the 68332's Timer Processor Unit (TPU). These include quadrature decode for the motor encoders, decoders for the R/C signals, sequencing and timing for a 4-element sonar array and user interface, and serial communications for a GPS and a 9 DOF Inertial Measurement Unit.

4. Some information on the implementation of the IMU subsystem and its integration into the robot's navigation algorithms is available from jBot's IMU Odometry, also linked from jBot's webpage.

5. **jBot** has a 4-element sonar array that needs more complex calculations and sequencing than the two sonar on **SR04**. That is the job of the *sonar_task()*. The actual behaviors for this task are the *obstacle(), perimeter()* and *escape()* behaviors.

6. *sonar_task()* will probably be moved eventually to an off-board processor to allow expansion from 4 to 6 sensors in the sonar array, for full 90 degree coverage (current coverage is 60 degrees).

Notes on multitasking:

The use of a cooperative multi-tasking kernel to implement control code on these robots is a matter of convenience and not of necessity. All the same functions can be implemented without it. The only absolute requirement is for a 32 bit real-time clock running at 1000 Hz, which can be accessed by all the tasks. For SR04 and jBot, that clock is named *sysclock* and is driven by a 1000 Hz interrupt.

Having said that, a multitasking OS is an awfully handy thing to have when implementing robot control code. The Flynn/Jones examples use Interactive C, which is available for the Motorola 68HC11 processor from Newton Labs. **SR04** and **LEGOBot** use a home-brew executive based on the article "A Minimalist Multitasking Executive" from *Circuit Cellar* Magazine. **jBot** also uses a home-brew executive for the M68332 which will be published on-line by the DPRG in the near future. Fellow robot builder Duane Gustavus uses the powerful RTEMS OS to create his subsumption robot code, running on a Mini Robomind controller like that used by **jBot**.

Notes on ballistic behaviors:

Joseph Jones uses the term "ballistic" in ***Robot Programming*** to define a class of behaviors that, once initiated, use internal timers rather than external sensors to generate outputs.

The most familiar of these for many robot builders is the classic bumper behavior, where a bumper press causes the robot to back up for a while, rotate away from the collision for a while, and drive forward for a while, before releasing control of the robot.

These are called ballistic behaviors in order to make the distinction that, like a shot fired from a cannon, they continue to completion "on their own," as it were, once initiated. Jones distinguishes these behaviors from basic closed loop or feedback tasks, which he calls "servo" behaviors.

Ballistic behaviors in a certain sense break the subsumption paradigm, because they cannot easily be subsumed. They are however, in my experience, the exception rather than the rule, in terms of how often they are actually needed to control the robot. Jones advises using ballistic behaviors sparingly.

Three attributes of ballistic behaviors that may not be immediately apparent:

1. Ballistic behaviors cannot be subsumed: they must run to completion.
2. Ballistic behaviors are therefore usually assigned highest priority.
3. Ballistic behaviors can, however, be aborted.

In fact, abort is really the only reasonable response of a ballistic task when subsumed. For the **SR04** robot, the only task that can subsume a ballistic bumper behavior, which is the highest priority behavior, is another ballistic bumper behavior.

So if, in the course of a ballistic bumper behavior, the **SR04** has another collision, the current ballistic behavior is aborted, and a new behavior is initiated.

For robots with more than one ballistic behavior at more than one priority level, the "arbitration winner" global defined by Jones can be used by each ballistic behavior to see if it has been subsumed, and to abort itself in that case.

Note that in the case of a ballistic behavior implemented as an augmented finite state machine, as illustrated in the next section, abort just means resetting the task's state variable to 0.

jBot uses ballistic behaviors to escape from entrapment.

IV. Example Subsumption Behaviors

Here are some sample subsumption tasks based on the arbitration schemes described in the previous sections. Included are example implementations for concurrent behavior tasks in a multitasking environment and also for inclusion in a single larger sensor loop. A sample ballistic behavior is implemented both for multitasking and as a stand-alone Augmented Finite State Machine (AFSM, an FSM augmented with a timer), suitable for inclusion in a large sensor loop like those used by **jBot** and **SR04**.

These are coding examples and not the actual code running on those robots, which is more complex, but the principles are the same. These examples use **jBot's** arbitrator and implement the “continuous message” model of subsumption, as described below and illustrated graphically in section VI.

A. A Photo behavior

This behavior uses a pair of Cadmium Sulfide photo cells to seek toward or away from light. This is one of the easiest behaviors to get working. The sensors are used as the upper half of a pair of voltage dividers which are read by two A/D converter channels commonly found on robot micro-controllers, including the Motorola HC11 used on **SR04** and **jBot's** MRM board.

The photocells are mounted on the rear of the robot pointing forward and aimed slightly left and right of center. Here's a picture of a pair of photocells mounted on the motion detector of **SR04**.

```
int photo_task()
{
    extern LAYER photo;                  // C structure for task output

    int detect = read_analog(LEFT) - read_analog(RIGHT) + PHOTO_OFFSET_ERROR;

    if (photo_avoid == TRUE) detect = -detect;

    if (abs(detect) > PHOTO_DEADZONE) {  // if one side significantly brighter

         photo.cmd = top_speed;          // request top speed

         if (detect > 0)                 // turn toward brighter side
              photo.arg = TURN_LEFT;
         else photo.arg = TURN_RIGHT;

         photo.flag = TRUE;              // signal arbitrater we want control

    } else photo.flag = FALSE;           // else we are in dead-zone with light
                                         // more-or-less directly ahead,
                                         // so release control
}
```

This task can be called from a single large sensor loop. It can also be used as a standalone concurrent task like the IC examples from ***Mobile Robots*** by calling it from an endless loop with the appropriate delay:

```
void photo_behavior()      // photo behavior as an IC or concurrent task
{
    while (1) {
        photo_task();      // read/compute/output and set subsumption flag
        msleep(50);        // suspend in multi-tasking queue for 50 ms.
    }
}
```

Notes on photo_task():

1. The PHOTO_OFFSET_ERROR constant can be used to match the response of the two photo cells, so that the robot actually drives toward the light and not offset left or right of it.

2. The DEADZONE constant keeps the robot from “swimming” towards the light by defining a dead-band around zero when the light is roughly straight ahead. Note that the photo layer only releases control when the light is in this dead-band.

3. The resulting behavior is that this layer sends outputs to turn left and right at top speed towards the brightest light, unless that light is in front of it.

4. The behavior can seek instead toward the “darkest” light by negating the detect value after the sensors are read. Together these are often called *photovore* and *photoavoid* behaviors. (Also *moth* and *cockroach* behaviors, for obvious reasons.)

B. An IR avoidance behavior.

This IR behavior uses a pair of IR LEDs as emitters and a pair of Radio Shack IR detector modules mounted on the front of the robot, again pointed left and right, as detectors. Here is a picture of the IR emitters and detectors mounted in a shadow box attached to the front bumper of **SR04**. The sensors themselves are driven by a 40kHz signal modulated at 100 Hz by an interrupt routine.

```
int ir_task()
{
    extern LAYER ir;                        // C structure for task output

    int detect = read_ir_sensors();         // read sensors
    if (detect == LEFT) {                   // if reflection on the left
            ir.cmd = HALF_SPEED;            // request slow down
            ir.arg = RIGHT_TURN;            // and turn right
            ir.flag = TRUE;                 // signal arbitrater we want control
    } else {
        if (detect == RIGHT) {              // else if reflection on the right
            ir.cmd = HALF_SPEED;            // request slow down
            ir.arg = LEFT_TURN;             // and turn left
            ir.flag = TRUE;                 // tell arbitrater we want control
        } else {
            if (detect == BOTH) {           // else if reflection left and right
                ir.cmd = ZERO_SPEED;        // request slow to zero
                ir.arg = keep_turning();    // keep turning same direction
                ir.flag = TRUE;             // signal arbitrater we want control
            } else {
               ir.flag = FALSE;             // else no detection, release control
            }
        }
    }
}
```

Using the IR task in a multitasking implementation is similar to the photo task:

```
void ir_behavior()
{
    while (1) {           // endless loop
        ir_task()         // read/computer/output and set subsumption flag
        msleep(50);       // suspend in multitasking queue for 50 ms.
    }
}
```

Notes on ir_task():

1. The IR emitters and detectors together create a pattern of two overlapping lobes of reflections in front of the robot. Objects which reflect infrared light will trigger left, right, or both detectors when within the ranges of those two lobes, out to about 18 inches away, depending on the reflectivity of the object's surface.

2. This behavior slows the robot anytime it detects an IR reflection, and slows the robot to a stop if the detection is directly ahead, allowing only rotations around the robot's center until one or the other of the detectors is cleared. As a practical matter, this means the robot only can accelerate to top speed in the absence of IR detections.

3. Here is a blow-by-blow account of the arbitration cycle for this behavior as it avoids an obstacle on the right.

The IR avoidance task detects an IR reflection on the right sensor, so it outputs a command to slow down and turn left, and sets its subsumption flag to TRUE in order to signal the arbitrator that it wants to control the robot. Assume it is the highest priority flag at that moment, so the arbitrator sends the IR task's commands to the motors to slow down and turn left, and that is what the motors do (ignoring any PID slewing, etc) for the next 50 ms, until the next time through the subsumption loop. But that's all, just for the next 50 milliseconds.

Now, 1/20 second later as the loop executes again, the robot has hardly moved at all, and so the IR avoidance task still detects an IR reflection on the right, and again outputs a command to slow down and turn left, and again sets its subsumption flag to active to signal the arbitrator. Again it is the highest priority layer signaling and the arbitrator again passes its command to the motors to slow down and turn left, and that's what they do for the next 50 milliseconds. But only for the next 50 milliseconds.

This processes continues each time through the subsumption loop, with the IR avoidance winning the priority contest in little, 50 millisecond chunks, and passing its command to turn left to the motors each time, 20 times a second.

After, lets say, 2 seconds (40 times through the loop, 40 turn left commands to the motors) the robot has finally turned left far enough that the next time through loop, the IR avoidance detector no longer "sees" a detection on the right. So on that pass through the loop, the IR avoidance behavior's flag becomes FALSE (no detection). And some other, lower priority, behavior gets to pass its commands to the motors.

The output from the IR avoidance behavior goes away as soon as the detection goes away, (or within 50 ms thereof). By contrast, the output from the photo behavior only goes away when the robot is headed directly towards the brightest light.

SR04 navigating and grabbing things.

C. A Collision recovery ballistic behavior.

Here is simplified code for a bumper behavior. This behavior is illustrated two ways; first as a multi-tasking behavior, and then as an Augmented Finite State Machine (AFSM). The sensors in this case are left and right micro switches mounted behind the front bumper, which is pivoted in the center. Here is a diagram of the bumper on **SR04**.

jBot has “virtual bumpers” derived from it's wheel odometry and Inertial Measurement Unit, but the behaviors, once triggered, are very similar to those outlined here.

```
void bumper_behavior()      /* Collision recovery as concurrent task */
{
    extern LAYER bump;      // C structure for behavior output
    int bumper;             // local to hold bumper switches status

    while (1) {             // endless loop

        bumper = read_bumper_switches();   // read the bumper switches

        if (bumper) {                     // if any switches are closed

                                          // Ballistic segment 1
            bump.cmd = BACKUP_SLOW;       // request reverse low speed
            bump.arg = 0;                 // straight back
            bump.flag = TRUE;             // signal arbitrater
            msleep(1000);                  // suspend and back up for 1 second

                                          // Ballistic segment 2
            bump.cmd = HALF_SPEED;        // then request forward ½ speed
            if (bump == LEFT)             // and turn away from the bump
                 bump.arg = RIGHT_TURN;
            else bump.arg = LEFT_TURN;
            msleep(500);                   // suspend and turn for 1/2 second

                                          // Ballistic segment 3
            bump.cmd = top_speed;         // request full speed
            bump.arg = 0;                 // straight forward
            bump.flag = TRUE;             // signal arbitrater
            msleep(250);                 // suspend and forward for 1/4 second

                                          // Ballistic segments complete
            bump.flag = FALSE;            // then reset arbitration flag and loop

       } else {                           // else if no bumps,
            bump.flag = FALSE;           // reset flag and
            msleep(10);                 // loop at 100Hz, looking for bumps
       }
    }
}
```

This behavior is very similar to the *escape()* behavior described by Flynn and Jones in their “Robot Programming” chapter of ***Mobile Robots.*** Note that during the backup, turn, and forward sequences, the behavior is suspended in the multitasking queue by the msleep() calls, and does not test the switches again until the end of the entire ballistic behavior. Termination of the behavior depends on the timers.

(This simple implementation example does not allow for the behavior to interrupt or reset itself, but a practical bumper behavior might need that capability.)

D. Collision recovery as an AFSM task

Here is the same collision recovery behavior coded as an Augmented Finite State Machine, suitable for inclusion in a large sensors() loop like those used by **SR04** and **jBot.**

```
/* persistent local variables for the AFSM */

int bumper,             // bumper switches status
    bumper_state,       // AFSM variable
    bumper_timer;       // ballistic behavior timer

int bumper_task()       // Collision recovery behavior as an AFSM
{
    extern LAYER bumper;                  // C structure for task output
    extern int sysclock;                  // 32 bit 1000 Hz real time clock

    if (sysclock > bumper_timer) {        // if timer has expired

       if (bumper_state == 0) {           // state 0 == looking for bumps

            bumper = read_bumper_switches;  // so read the switches

            if (bumper) {                  // if any switches are closed
                                           // Ballistic segment 1
                bumper.cmd = BACKUP_SLOW;  // request backup slow
                bumper.arg = 0;            // straight back
                bumper.flag = TRUE;        // signal arbitrater

                bumper_timer = sysclock + 1000;  // set timer to now + 1000 ms
                bumper_state = 1;          // set state=1, backing up
                                           // and exit
            } else {                       // else, no bumper detections
                bumper.flag = FALSE;       // so reset arbitration flag and exit
            }
       } else
       if (bumper_state == 1) {            // ballistic segment 1 has completed
                                           // Ballistic segment 2
                bumper.cmd = HALF_SPEED;   // request ½ speed forward
                if (bumper == LEFT)        // turn away from bump
                     bumper.arg = RIGHT_TURN;
                else bumper.arg = LEFT_TURN;

                bumper_timer = sysclock + 500; // set timer to now + 500 ms
                bumper_state = 2;         // set state=2, turning away. and exit
       } else
       if (bumper_state == 2) {           // ballistic segment 2 has complete
                                          // Ballistic segment 3
                bumper.cmd = top_speed;   // request top speed
                bumper.arg = 0;           // straight forward
                bumper_timer = sysclock + 250;  // set timer to now + 250 ms
                bumper_state = 3;          // set state=3, drive forward, and exit

       } else {                            // ballistic segment 3 has completed
                bumper.flag = FALSE;       // reset arbitration flag to false,
                bumper_state = 0;          // and reset behavior state to 0
       }
    }
}
```

Notes on bumper_task():

The following cycle-by-cycle description of the arbitration loop applies to both the multi-tasking and the AFSM collision recovery ballistic behaviors. It describes the response of the task to a collision with the right bumper.

When the bumper gets a collision, on the right for our example, it closes the right bumper switch. The current status of the switches is returned by the *read_bumper_switches()* sensor code, and placed in the variable *bumper*. If *bumper* is non-zero, one of the bumper switches has been pressed.

The bumper task reads the bumper switches once each time around the arbitration loop, or 20 times per second. If no switches are pressed then the variable *bumper* is 0, and the task just exits. That's what happens most of the time.

In our case the *bumper* variable is not zero, and that triggers the start of a ballistic behavior.

The task first requests the robot's motors to backup at half speed, sets its subsumption flag=TRUE to signal the arbitrator, and sets an internal TIMER to, let's say one second, i.e., 1000 ms.

Assuming this behavior is the highest priority asserting a flag, the command to backup will be passed by the arbitrator to the motors. And for the next 50 ms, (ignoring any PID slewing, etc) that's what the motors will do.

Now, 1/20 second later when the loop executes again, the task is no longer reading and testing the sensors, but rather is reading and testing the timer.

That is why it is a ballistic behavior. Its termination depends on an internal timer, rather than the absence of an external sensor detection. The original switch closure has long disappeared before the ballistic behavior completes.

For the multitasking implementation, this means the task is suspended in the multitasking queue for 1000 ms. For the AFSM task, it means that each time it is called from the *sensors()* loop it returns immediately because its timer has not yet timed out, for 1000 ms. Consequently the task leaves the output command as backup and the subsumption flag as TRUE. The arbitrator passes the command each time to the motors, and the robot continues to backup.

This continues for another 18 times though the subsumption loop (20 loops = 1 sec). On the 21st time through the loop, the timer has expired, and the task sequences to the next state, which is a turn left command. It leaves the subsumption flag TRUE, requests the motors to turn left at half speed, and sets the internal TIMER to half second.

For the next half second, 10 times through the loop, the task is again waiting on the timer, while its command is passed each time by the arbitrator to the motors. When it times out on the 11th pass through the loop, the task sequences to the third and final segment of the behavior, which is a short drive straight forward (an attempt to get around whatever it is we collided with). The task requests top speed straight ahead, leaves the subsumption flag TRUE, and sets the timer to quarter second.

So for the next quarter second, 5 times through the loop, the task waits on the timer while passing its commands to the motors with each loop. On the 6th time through the loop, it times out, and the collision recovery ballistic behavior is complete.

The task then sets its subsumption flag FALSE and goes back to reading the bumper switches each time around the loop, allowing lower priority tasks to control the robot.

Note that the multitasking implementation of this behavior can scan the switches asynchronously from the rest of the arbitration loop, allowing it to loop at 100 Hz when testing the switches.

V. The Default Behavior

The default behavior is the lowest priority behavior in the subsumption arbitration scheme. The default behavior from the Flynn/Jones book is *cruise()*. This task requests full speed straight ahead from the motors, and its subsumption flag is always TRUE. Because it is the lowest priority behavior, that is what the robot does in the absence of any other higher priority motor commands.

A. The cruise() behavior

```
int cruise_task()
{
         extern LAYER cruise;

         cruise.cmd = top_speed;          // request top speed
         cruise.arg = 0;                  // straight ahead
         cruise.flag = TRUE;              // always

}
```

A small variation on this produces an interesting behavior. The addition of the *invert* flags illustrated for SR04 allows the task to drive to top speed OR to zero, depending on the state of the invert flag. An enable/disable flag for the behavior is also useful:

```
int cruise_task()
{
         extern LAYER cruise;             // C structure for output

         if (cruise_enable) {             // if enabled
             if (cruise_invert)           // and inverted
                  cruise.cmd = 0;         // request speed 0
             else cruise.cmd = top_speed; // else request top speed
             cruise.arg = 0;              // straight ahead
             cruise.flag = TRUE;          // always
         } else {
             cruise.flag = FALSE;         // unless disabled
         }
}
```

This can also be called from an endless loop with a delay for a multitasking implementation.

```
void cruise_behavior()
{
         while (1) {
             cruise_task();
             msleep(50);
         }
}
```

Notes on cruise():

1. The cruise behavior normally accelerates the robot to full speed in the absence of other behaviors.

2. Inverting the response by driving the robot to zero produces an interesting behavior also. The robot tends to move in response to sensor detections, but otherwise comes to a rest. So it will wander around the space until it "finds its spot" and there it will stop. Lighting changes in the room or people or objects moving into its IR detection range will cause it to move and to seek a new resting place. Sort of like a pet.

B. Navigation Behavior

The **SR04, jBot,** and **LEGObot** robots each run a concurrent task that calculates the robot's own position in X,Y inches relative to where it was last reset, using data from the robot's wheel encoders. (**jBot** also uses a 3 axis Inertial Measurement Unit and a GPS). These coordinates are calculated in floating point at a rate of 10 Hz on the two HC11-powered robots and 20 Hz on **jBot.**

Using the coordinates of the robot and a bit of trig, the angle and distance to any other arbitrary coordinates can be determined. The function *locate_target()* listed at the end of this article takes a pair of target coordinates and returns *distance* in inches and *heading_error* in degrees.

With the *heading_error* value we can write a simple navigation behavior, very similar to the photo attraction behavior, that will steer the robot towards a coordinate location.

```
int navigation_task()
{
    extern LAYER navigate;              // C structure for output
    int distance, heading_error;        // values returned from virtual sensor

    locate_target(target,&distance,&heading_error);    // read "sensor"

    if (abs(heading_error) > NAV_DEADZONE) {   // is heading error large enough?

        navigate.cmd = top_speed;      // yes, request top speed

        if (heading_error < 0)         // turn toward target
             navigate.arg = TURN_LEFT;
        else navigate.arg = TURN_RIGHT;

        navigate.flag = TRUE;          // signal arbitrater
    } else {
        navigate.flag = FALSE;        // else target is ahead, reset flag
    }
}
```

As with the photo task, this task only releases the subsumption flag when the target is more or less directly ahead, in the navigation dead-zone. I usually make NAV_DEADZONE 5 or 10 degrees wide. As with the photo behavior, this behavior should have low priority.

We have data for navigation that we don't have for the photo behavior: the actual distance to the target. This can be used for all sorts of things, including decelerating and stopping when the robot arrives at the target.

C. The default prowl() behavior

SR04, **jBot**, and the **LEGOBot** each use a default behavior named *prowl()*, which incorporates elements of both the *cruise()* and the *navigate()* behaviors, depending on the state of a global *target_active* flag. When the flag is TRUE, *prowl()* steers the robot towards an X,Y coordinate location. Otherwise it accelerates the robot straight ahead to top speed or decelerates to 0, depending on the prowl_invert flag.

Here's a simplified implementation of the prowl() task that also includes the ability to slow down and stop the robot when it reaches a coordinate target.

```
#define TARGET_RADIUS 10         // error radius around target in inches
#define DOWN_RAMP 36             // slow down within 36 inches of target
#define MINSPEED_AT_TARGET 5     // don't go slower than 5 magic speed units.

int prowl_enable, prowl_invert;  // prowl mode flags

int prowl_task()                            // call this from 20 Hz sensor loop
{
    extern LAYER prowl;                     // C structure for output
    extern int target_active;               // global, reset when target acquired
    int distance, heading_error;            // values returned from virtual sensor

    if (prowl_enable) prowl.flag = TRUE; else prowl.flag = FALSE;

    if (target_active == 0) {               // do cruise() behavior
        if (prowl_invert) prowl.cmd = 0;    // decelerate
        else prowl.cmd = top_speed;         // or accelerate
        prowl.arg = 0;                      // straight ahead

    } else {                                // else do navigate() behavior

        locate_target(target,&distance,&heading_error);    // read the "sensor"

        if (distance < TARGET_RADIUS) {     // arrived at target?
            prowl.cmd = 0;                  // yes, stop
            prowl.arg = 0;                  // stop turning
            target_active = 0;              // signal that target is acquired

        } else {                            // else, still looking for target
            if (distance < DOWN_RAMP) {     // slow down when getting close

                prowl.cmd =((distance*top_speed)/DOWN_RAMP);
                if (prowl.cmd < MINSPEED_AT_TARGET)
                    prowl.cmd = MINSPEED_AT_TARGET;

            } else {
                    prowl.cmd = top_speed;  // distance > 36", request top speed
            }
            if (abs(heading_error) > NAV_DEADZONE) { // heading error large?
                if (heading_error < 0)        // yes, steer toward target
                     prowl.arg = TURN_LEFT;
                else prowl.arg = TURN_RIGHT;
            } else {
                prowl.arg = 0;   // else target is in dead-zone, straight ahead
            }
        }
    }
}
```

D. Priorities

The selection of priority levels for the individual subsumption tasks is highly dependent on the particular problem set that the robot is attempting to solve. For the examples in this article the main problem to solve is autonomous navigation. A priority scheme suitable for navigation might not be suitable for, let us say, soda can collection.

One useful method for assigning priorities for autonomous navigation problems is to arrange the tasks in order of sensing distance from the robot. That way the highest priority behaviors are those having sensor detections closest to the robot, and the lowest priority behaviors are those with the most distant detections.

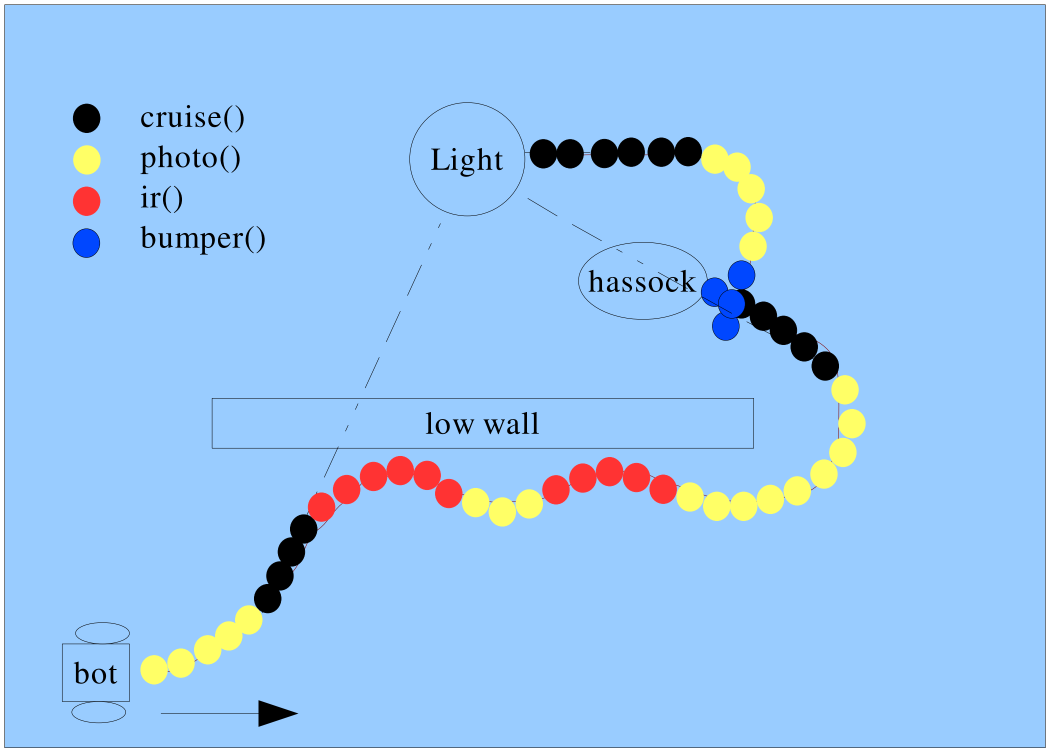
Highest priority

bumper()	detections actually touching the robot
feelers()	detections within 2 inches of robot
ir()	detections within 18 inches of robot
sonar()	detections within 32 feet of robot
navigation()	“detections” within 2^31 inches of robot
photo()	detections within 93 million miles of robot (the Sun!)
cruise()	detections at infinity (always)

Lowest priority

VI. Navigating through a sample space

The figure below illustrates the path of a robot moving through a space, using our version of the four-behavior subsumption model from Flynn/Jones, illustrated in the first schematic at the top of this article. The colored circles represent which behavior is controlling the robot for that part of its path*: photo() is yellow, prowl() cruise mode is black, ir() is red, and bumper() is blue.*



A. Navigation while seeking a bright light: an analysis

1. The room has a ceiling light toward the back and a low wall between it and the robot. There is also a black leather hassock which does not reflect IR light very well. The robot begins in the lower left corner facing across the room.

2. The default *prowl()* behavior in *cruise()* mode requests full speed, straight ahead. The photo*()* behavior requests full speed and left turn, towards the bright ceiling light. The ir*()* and *bumper()* have no detections and their subsumption flags are FALSE, so the *photo()* command is passed to the motors. [yellow]

3. The robot arcs forward and left until the ceiling light is directly in front of the robot, and the *photo(*) flag becomes FALSE. At that point the *prowl()* command *cruise()* mode takes over and drives the robot straight toward the light. [black]

4. When the robot gets within about 18 inches of the low wall, the IR detectors begin to see it. What it does next depends on the angle at which it approaches the wall. In the drawing above, the left sensor sees the wall first and begins to request that the robot turn right. The *ir()* behavior has higher priority than the *photo()*, so the arbitrator passes the turn right commands to the motors, and *photo()*, which now wants to turn left, towards the light, is subsumed along with the *prowl()* output. [red]

5. After the *ir()* behavior clears the wall, it releases control of the robot and the photo behavior, which has been wanting to turn left, takes control of the robot and turns it back towards the wall, where it is again seen by the IR. [yellow then red]

6. In this fashion the robot “swims” along the wall, with the photo pulling it toward the light and the IR pushing it away from the wall, in a wall-following behavior not implemented by the programmer directly. It “emerges” from the two behaviors and their interaction with the light and wall. These are called *emergent* behaviors. [red]

7. When the robot reaches the end of the wall the *ir()* behavior no longer prevents the *photo()* behavior from turning the robot far enough left to face the light. The *photo()* behavior then releases control and the *prowl()* behavior drives again straight towards the light. [yellow then black]

8. Between the robot and the light there is a black leather hassock that does not reflect IR well, and because the *ir()* behavior does not see it and subsume *prowl()*, the robot has a collision on its left bumper with the hassock. That triggers a backup-turnright-goforward ballistic behavior that allows the robot to drive clear of the hassock before releasing its control. [blue]

9. When the ballistic behavior completes, it releases control and the *photo()* behavior, which is now requesting a left turn, steers the robot left towards the light and then releases control to the *prowl()* behavior when the light is directly ahead. [yellow, then black]

10. What happens next depends on the robot's ability to maneuver but the robot will probably circle endlessly underneath the ceiling light.

B. Navigation while seeking a target waypoint coordinate.

If the *photo()* behavior in the above walk-through is replaced by the navigate() behavior seeking a coordinate target at the location of the ceiling light, the behavior and path of the robot will be very much the same as described above. If the *prowl()* navigate behavior is used instead, the robot will also slow down and stop upon reaching the target location.

C. Actual Navigation Examples

Here are some videos of the robots seeking on a target coordinate with various obstacles in the way.

1. **LEGOBot** seeking on a target 8 feet away and back, with a chair in the way, running only *prowl()* and *bumper()* behaviors:

<http://www.geology.smu.edu/~dpa-www/robo/lego/lego-06.mpg>

2. **LEGOBot** seeking a target 8 feet away and back, with a cardboard box in the way, running *prowl()*, *bumper()*, and *ir()* avoid behaviors:

<http://www.geology.smu.edu/~dpa-www/robo/lego/lego-02.mpg>

3. **SR04** maneuvering in the attic of the Heroy building at SMU. It is seeking a coordinate 24 feet away and back to the origin, with a bunch of junk and attic detritus in the way. It is running *prowl()*, *ir()*, *sonar()*, and *bumper()* behaviors:

<http://www.geology.smu.edu/~dpa-www/robots/mpeg/sr04_ob3.mpg>

4. **SR04** seeks four coordinate waypoints in the shape of a square, with obstacles and moving feet in the way:

<http://www.geology.smu.edu/~dpa-www/robots/mpeg/sr04_sq_001x.mpg>

5. **SR04** collects empty soda cans and returns them to 0,0 while seeking on three target coordinate locations in the shape of a T.

<http://www.geology.smu.edu/~dpa-www/robots/mpeg/cancan.mpg>

6. **jBot** seeking a pair of coordinates 20 feet apart over rough terrain:

<http://www.geology.smu.edu/~dpa-www/robo/jbot/jbot_rough_01.mpg>

7. **jBot** seeking a pair of coordinates 100 feet apart, with gardens and concrete benches in the way:

<http://www.geology.smu.edu/~dpa-www/robo/jbot/jbot_gardenx.mpg>

8. **jBot** seeking a coordinate target 1500 feet through the woods and back to the origin:

<http://www.geology.smu.edu/~dpa-www/robo/jbot/jbot_hatrick2_2.mpg>

9. **jBot** seeking a coordinate target half mile away, with a large institutional building, campus, and parking lots in the way:

<http://www.geology.smu.edu/~dpa-www/robo/jbot/jbot2/jbot_ti2.mpg>

VII. Some Observations

My hope is that enough information is available here so that others in the DPRG and elsewhere who have asked about the control code for these robots will be able to implement similar software on their own creations.

This tutorial has concentrated on navigation tasks but the same principles apply for arbitrators used to control other robot functions and their behaviors, such as grippers and other physical I/O devices.

Steve Rainwater of Robots.net, who encouraged me to write this article, is working with several others to create a DPRG database of GPL'd robot code that has been developed by the robotics community, and hopefully some of the code for these robots will make it into that repository as well.

23 Mar 2007
Denton, Texas
dpa

Appendix: Locate A Target

A function for determining distance and heading to an arbitrary coordinate location from the robot's location in Cartesian space where 0,0 is the place at which the robot was last reset.

```
/* ------------------------------------------------------------------- */
/* locate_target() uses these global variables */
/* ------------------------------------------------------------------- */

float X_target;                 /* X lateral relative target position */
float Y_target;                 /* Y vertical relative target position */
float target_bearing;           /* bearing in radians from current position */

int target_distance;            /* distance in inches from position */
int heading_error;              /* heading error in degrees */
int last_target_distance;       /* history */

/* ------------------------------------------------------------------- */
/*  calculate distance and bearing to target X,Y
        inputs are X_target, X_pos, and Y_target, Y_pos
        output is target_distance, heading_error
*/

void locate_target()
{
        float x,y;

        x = X_target - X_pos;
        y = Y_target - Y_pos;

        last_target_distance = target_distance;
        target_distance = (int)(sqrt((x*x)+(y*y)));

        if (x > 0.00001) target_bearing = 90-(atan(y/x)*RADS);
        else if (x < -0.00001) target_bearing = -90-(atan(y/x)*RADS);

        heading_error = (int)((target_bearing - (theta*RADS)));
        heading_error = heading_error%360;

        if (heading_error > 180) heading_error -= 360;
        else if (heading_error < -180) heading_error += 360;

}

/* ------------------------------------------------------------------- */
```