# Autonomous Mobile Robots

# Lecture 08: Reactive Control

Lecture is based on material from Robotic Explorations: A Hands-on Introduction to Engineering, Fred Martin, Prentice Hall, 2001.

## Outline

- Reactive vs. Algorithmic Control
- Multi-Tasking
- Subsumption Architecture
- Priority-Based Control Program
- How the Prioritization Algorithm Works
- Using Reactive Control
  - Robo-Miners
  - Collecting Soda Cans
  - Reactive Groucho
- Extending the Prioritization Framework

Copyright Prentice Hall, 2001

## Homework #8

- **Subsumption Architecture:** Read Brooks, R. A. and A. M. Flynn, "Fast, Cheap and Out of Control: A Robot Invasion of the Solar System," *Journal of the British Interplanetary Society*, October 1989, pp. 478--485. Web site: http://www.ai.mit.edu/people/brooks/papers/fast-cheap.pdf

Copyright Prentice Hall, 2001

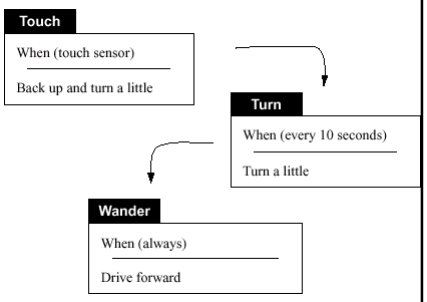

## Reactive vs. Algorithmic Control

- **Algorithmic Control:** robot's program is fundamentally a series of steps or actions to be taken in a predetermined order.
  - Most effective when the robot's world and its interactions with it are well-structured
    - Manipulator arms in factories typically use algorithmic control with great success
  - Loses its appeal when the robot must deal with unexpected situations
  - When it is extended to deal with error situations, the algorithmic method becomes a complicated tree of branching decisions that is hard to design and debug
- **Reactive Control:** robot's program is organized around a collection of separate mini-programs, all running at once and able to take control of the robot as they see fit
  - For a very simple, minimal *HandyBug* program, there might be
  - a **touch sensor process**, which monitored the robot's touch sensors and caused the robot to back up and turn when it hit something
  - a **periodic turn process**, that caused the robot to take a turn every now and then
  - and a **wander process**, that caused the robot simply to move
- Reactive control excels in complex situations with many unpredictable interactions

Copyright Prentice Hall, 2001

# Multi-Tasking

- **Touch process:** if a touch sensor is pressed, then its action will run, and it will back up the robot and turn a little
- **Turn process:** activates every ten seconds, causing the robot to make a small turn
- **Wander process:** is always active, and causes the robot to drive straight ahead
- Which process gets priority over the others?
- While all processes can be active at once, checking their conditions to determine whether or not they should do something, only one process can have control of the robot's outputs at at given time
- Fixed priority system: each process is assigned a priority level; at any given time, the process with highest priority gets control of the robot




Arrows indicate a path of priority among three processes. **Touch** has the highest priority, followed by **Turn** and then **Wander**. This ordering makes sense because while the robot's wandering about, if it hits something, the obstacle avoidance should take precedence over the wandering.

Copyright Prentice Hall, 2001


# Multi-Tasking

- IC comes with the ability to *multi-task*, or run several programs at once
- Ideal for reactive robot programming, where the robot has several "behaviors" running at the same time

```
void main () {
back_and_forth();
sensor_beep();
}
```

```
void back_and_forth () {
while (1) {
      fd(0); msleep(500L);
      bk(0); msleep(500L);
      }
}
void sensor_beep () {
while (1) {
      if (analog(0) < 100) beep
      }
}
```

- As it stands, `sensor_beep()` will never run, because the computer would be stuck in an infinite loop performing `back_and_forth()`

Copyright Prentice Hall, 2001

## Multi-Tasking

- IC's **`start_process()`** function, however, allows multiple programs to execute in tandem:

```
void main () {
start_process(back_and_forth());
start_process(sensor_beep());
}
```

- Now, both subroutines will execute at once; motor 0 will flash back and forth, and the beeper will trigger whenever analog sensor 0 falls below 100
- Note that `start_process()` can also be used from IC's command line on the host computer
- IC's multi-tasking capability makes it easy to implement reactive control for robot programming

Copyright Prentice Hall, 2001


## Subsumption Architecture

- Professor Rodney Brooks of the MIT Artificial Intelligence Laboratory is considered the father of the reactive robotics approach, which he calls "behavioral robotics." The central idea of Brooks' approach is that more sophisticated robot competencies should be built on top of simpler ones, an approach he calls the **subsumption architecture.**
- Instead of all robot inputs feeding into a sensory perception unit, which creates a "world model" of the robot's environment, which feeds into a planning module, Brooks has argued that robot perception and action should be closely linked, and that **complex behaviors can be built from the interactions of simple ones.**
- **Example:** a robot that must walk should first learn to stand. Then later behaviors can "exploit" earlier ones: a task which causes a legged robot to move its legs can make use of the knowledge embedded in the behavior that allows the robot to simply stand.
- Brooks has proposed that future unmanned interplanetary missions should be performed by hundreds—or thousands—of simple, insect-like robots that act in teams to accomplish work, rather than a large and complicated monolithic device. Individual robots could be considered expendable without jeopardizing the success of the entire mission, whereas if a single large robot had a failure, the mission would be over.
- Homework #6: Read: Rodney Brooks's paper on ***"Fast, Cheap, and Out of Control"***

Copyright Prentice Hall, 2001

## Priority-Based Control Program

- Creates priority-based, multi-tasking robot programs
- Example program:
  - **`priority.c`,** contains the prioritization code
  - **`plegobug.c`,** contains standard movement and touch sensor routines for the *HandyBug* robot; i.e., `forward()`, `backward()`, `left()`, `right()`, `left_touch()`, and `right_touch()`
  - **`lb1task.c`,** contains one robot behavior
- Generic touch sensor program: simple, clear, only deals with issues that concern the task
- Need a **prioritization structure to prevent conflicts**
- Assign each task a **process number** & **priority level:**
  - Process numbers identify task when issuing motor commands; allows prioritization routine to identify motor commands of each different task
  - Priority level allows tasks with higher priority to supersede tasks with lower priority

```
/* generic touch sensor pr
void touch () {
while (1) {
     if (left_touch()) {
     backward(); msleep(50
     right(); msleep(500L)
     }
     else if (right_touch(
        backward(); msleep
        left(); msleep(500
        }
}
}
```

Copyright Prentice Hall, 2001


## Priority-Based Control Program

### Rewritten Touch Sensor Task:

- `pid` = **process ID (process identifier)** argument is used in movement commands to indicate *which process is requesting which motor commands*
- Pair of commands used to **enable** and **disable** a process's functioning
- Separately (and in parallel), prioritization routine determines which process is enabled and has the highest priority, and then issues movement commands selected by that process to motors
- If a process has enabled itself and it has the highest priority of all enabled processes, then its movement commands get to run
- **Disable(pid):** issued after each set of movement commands that react to a touch sensor press
  - Process must disable itself when done so that other processes that have lower priority get a chance to operate
  - If not, then process would have control of the robot even if it were not issuing any movement commands (unless a process with higher priority were active)

```
/* touch sensor process */

void touch (int pid) {
while (1) {
     if (left_touch()) {
     enable(pid);
     backward(pid); msleep
     right(pid); msleep(50
     disable(pid);
     } else if (right_touc
        enable(pid);
        backward(pid); msl
        left(pid); msleep(
        disable(pid);
        }
     }
}
```

Copyright Prentice Hall, 2001

# Priority-Based Control Program

## Main Routine:

- Setting up the touch sensor task:

  (1) Process priority is assigned: touch sensor process is given **priority level 3** (zero priority is off; higher numbers are higher priority)

  (2) Process name is assigned: name is displayed by the prioritization program when the process is made active

  (3) Process itself is launched, and the process counter variable, `pid`, is incremented for the next process setup

- Launching prioritization program:

  – Before doing so, the process counter variable is stored in the program global `num_processes`

  – This is an efficiency measure for the *prioritize()* program

```
/* lb1task.c: main program
LEGObug and priority.c
one task: touch */

void main () {
     int pid= 0;
     /* touch sensor */
     process_priority[pid]
     process_name[pid]= "T
     start_process(touch(p
     /* motor arbitration
     num_processes= pid;
     start_process(priorit
}
```

Copyright Prentice Hall, 2001


# Priority-Based Control Program

## Try out on HandyBug:

- Load the file **lb1task.lis**, which loads the files **lb1task.c, plegobug.c,andpriority.c**
- In plegobug.c, motor and sensor wiring connections are defined: left motor in motor port 0, right motor in motor port 3, left touch sensor in digital port 7, and right touch sensor in digital port 8.
- With the full program running, notice that *HandyBug* just **sits there unless a touch sensor is pressed.** That's because when the touch sensors are not pressed, the touch() process is disabled, and since it's the only control task, therefore no control tasks are active. The prioritize() routine realizes that no tasks are active, and it shuts the motors off and displays the message "No tasks enabled."
- When a **touch sensor *is* pressed**, then the touch sensor task enables itself and issues motor commands to back up and turn away from the direction of contact. While the touch sensor task is active, prioritize() issues its motor commands to the motors, and displays the message "Running touch."
- *HandyBug* as an artificial creature: it can get out of the way if something hits its. *HandyBug* doesn't go anywhere on its own, but if something comes and bother it, it will move. The prioritization scheme we have been using thus far does not support one task taking advantage of another's capabilities; it only allows one task to override (or *suppress*, in Brooks' language) another. But it provides a good start in working with the types of ideas Brooks has developed.

Copyright Prentice Hall, 2001

# Priority-Based Control Program

## Add a Turn Task:

- Make a quick turn every 10 seconds
- The `periodic_turn()` routine is simple, with a tricky conditional expression in the `if` statement:
- Every ten seconds, the conditional fires and runs the code to make *HandyBug* turn: first it enables itself, then it turns right for a half a second, then it disables itself (allowing other processes to take over), and then it waits another half second.
- Last delay is necessary because the conditional expression will be true for an entire second every ten seconds. Without trailing delay, the `if` statement would fire again immediately, and *HandyBug* would end up turning for a whole second.
- Try out *HandyBug*'s dual behavior code—load **`lb2task.lis`,** which loads the new `lb2task.c` along with unchanged versions of `plegobug.c` and `priority.c` (unload lb1task.c first!)

```
/* lb2task.c: main program for LEGObug ar
two tasks: periodic_turn and touch */
void main () {
     int pid= 0;
     /* touch sensor */
     process_priority[pid]= 3;
     process_name[pid]= "Touch";
     start_process(touch(pid++));
     /* periodic turn */
     process_priority[pid]= 2;
     process_name[pid]= "Turn";
     start_process(periodic_turn(pid++));
     /* motor arbitration process */
     num_processes= pid;
     start_process(prioritize());
}
/* periodic turn: every 10 secs, turn a l
void periodic_turn (int pid) {
while (1) {
     if (((int)seconds() % 10) == 9) {
          enable(pid);
          right(pid); msleep(500L);
          disable(pid);
          msleep(500L);    }    }    }
```

Copyright Prentice Hall, 2001


# Priority-Based Control Program

## Type Coercion:

- `seconds()` routine reports the elapsed time as a floating point number; e.g., 53.374 sec. By prefacing the functional call with "`(int)`," this floating point value is converted to an integer value (e.g., 53).
- Necessary so that we may use the "%" operator, which is the arithmetic *modulus* function
- **"`(int)seconds() % 10`"** means, "take the elapsed system time, convert it to an integer, and report the remainder after dividing by 10"
- This provides a number from 0 to 9; the rest of the `if` statement simply compares this value with 9
- Thus, in the final second of every ten second period, the full expression yields a true, and the clause of the `if` statement runs
- **Reason for use:** IC does not have a modulus operator that works on floating point numbers. It does support floating point division, but not remainder. So the conversion to integer is used to circumvent this limitation.

```
if (((int)seconds() % 10) =
```

Copyright Prentice Hall, 2001

# Priority-Based Control Program

## Add a Wander Task:

- Simply drives straight ahead
- "Wandering" will happen because `periodic_turn()` will kick in every ten seconds, causing *HandyBug* to veer from a straight-line path.
- Notice: no loops are needed to work: it simply enables itself, and sets its movement command as drive forward.
- `wander()` is installed in the standard way with a priority of 1, the least among the three processes now installed.
- Load **`lb3task.lis`** to give it a try. Now *HandyBug* is a full-fledged explorer robot, able to roam about a room, a back up and turn away from any obstacles in its way—after hitting into them, of course!

```
/* lb3task.c main program for LEGObug and
three tasks: periodic_turn, touch and wan
void main () {
     int pid= 0;
     /* touch sensor */
     process_priority[pid]= 3;
     process_name[pid]= "Touch";
     start_process(touch(pid++));
     /* periodic turn */
     process_priority[pid]= 2;
     process_name[pid]= "Turn";
     start_process(periodic_turn(pid++));
     /* wander */
     process_priority[pid]= 1;
     process_name[pid]= "Wander";
     start_process(wander(pid++));
     /* motor arbitration process */
     num_processes= pid;
     start_process(prioritize());
}
/* wander process: just go forward */
void wander (int pid) {
     enable(pid);
     forward(pid);   }
```

Copyright Prentice Hall, 2001


# How the Prioritization Algorithm Works

## Basic Concept:

- Each process has a priority level, a pair of output values representing its left and right motor commands, an enable/disabled state indicator and a process name character string associated with each process for display to user
- The `prioritize()` process, which runs alongside all of the behavior processes cycles through the list of enabled processes, finds the one with the highest priority, and copies its motor output commands to the actual motors.
- Two global variables used by prioritization method:
  - **`num_processes`**, which hold the number of processes (to simply the search for the one with the highest priority)
  - **`active_process`**, which is dynamically set by the `prioritize()` process each time it chooses a behavior task to run

Five global arrays used to store behavior process state variables:

**`process_priority[]`** Stores each process's priority level

**`process_enable[]`** Indicates whether a process is enabled or disabled at any given point in time. When a process is enabled, its priority level is stored here; when a process is disabled, a zero is stored.

**`left_motor[]`** Holds a process's current left motor command

**`right_motor[]`** Holds a process's current right motor command

**`process_name[]`** Stores the process name

Copyright Prentice Hall, 2001

# How the Prioritization Algorithm Works

## Data Structures:

**`process_name[]`** holds the process names as set up by `main()`

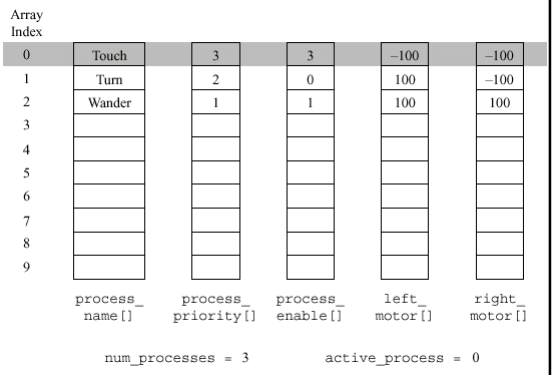
**`process_priority[]`** holds the fixed priority values as assigned to the processes in `main()`

**`process_enable[]`** holds dynamic enable values

– Touch is enabled, since its priority level has been copied into the `process_enable[]` array. Turn is disabled, and Wander is always enabled

**`Left_motor[]`** and **`right_motor[]`** hold values assigned by the behavior tasks

– Even though Turn is disabled, its entries are still present in the motor arrays from a previous time. No need to clear them out




**`Active_process`** assigned by `prioritize()`, is zero, indicating that the process with an index 0—the touch sensor process—is active.

Copyright Prentice Hall, 2001


# How the Prioritization Algorithm Works

## Code:

• **Enable:** copy process's priority level, which is stored in process_priority[], into "enabled" array

• **Disable:** zero stored in process's position in process_enable[] array

• **Prioritize:** Initialize local variable max=0, then scan `process_enable[]` array, looking for the process with the highest priority level.

• If `max` =0 after looking through all of the installed processes, then none of them is enabled. In this case, both motors are turned off, and the message "`No tasks enabled`" is displayed on the LCD screen.

• If `max>0`: then at least one process is enabled. The next step is to again search through the list of processes, and find one with a priority that matches the maximum value.

```
/* priority.c: arbitration program for
multi-behavior robot control */
/* define process tables */
int process_priority[10];
int process_enable[10];
char process_name[0][10];
/* define motor output tables */
int left_motor[10];
int right_motor[10];
/* globals */
int num_processes= 0;
int active_process= 0;

/* set process_enable entry to process'
void enable (int pid) {
     process_enable[pid]= process_prior
}
/* set process_enable entry to 0 */
void disable (int pid) {
     process_enable[pid]= 0;
}
```

Copyright Prentice Hall, 2001

## How the Prioritization Algorithm Works

### Code:

- If there is a "tie," with more than one process at the maximum priority, select the first one it encounters (could be re-written to randomly choose a process)
- Global variable `active_process` is set to hold the index of the highest priority process, and then the motor commands of this process are written out to the motor ports.
- Finally, the name of the active process is printed on the LCD display.
- At this point the loop repeats, and the process selection activity begins anew.

```
void prioritize () {
int max, i;
while (1) {
/* find process with maximum priority */
     max= 0;
     for (i=0; i< num_processes; i++)
          if (process_enable[i] > max) max= process_e
/* if no processes enabled, turn off motors */
     if (max == 0) {
          motor(LEFT_MOTOR_PORT, 0); motor(RIGHT_MOTO
          printf("No tasks enabled\n");
     } else {
/* get pid of active process */
/* if more than one at highest level, get the first o
          for (i=0; i< num_processes; i++)
               if (process_enable[i] == max) break;
          active_process= i;
/* set the motors based on the commands of this proce
          motor(LEFT_MOTOR_PORT, left_motor[active_pr
          motor(RIGHT_MOTOR_PORT, right_motor[active_
/* display name of active process */
          printf("Running %s\n", process_name[active_
          }  }  }
```

Copyright Prentice Hall, 2001


## How the Prioritization Algorithm Works

### HandyBug Movement and Touch Routines:

```
/*
plegobug.c: movement and touch sensor commands
for LEGObug with priority.c
*/
/* motor and sensor ports */
int LEFT_MOTOR_PORT= 0;
int RIGHT_MOTOR_PORT= 3;
int LEFT_TOUCH_PORT= 7;
int RIGHT_TOUCH_PORT= 8;
/* movement commands */
void forward (int pid) {
left_motor[pid]= 100;
right_motor[pid]= 100;
}
void backward (int pid) {
left_motor[pid]= -100;
right_motor[pid]= -100;
}
void right (int pid) {
left_motor[pid]= 100;
right_motor[pid]= -100;
}
```

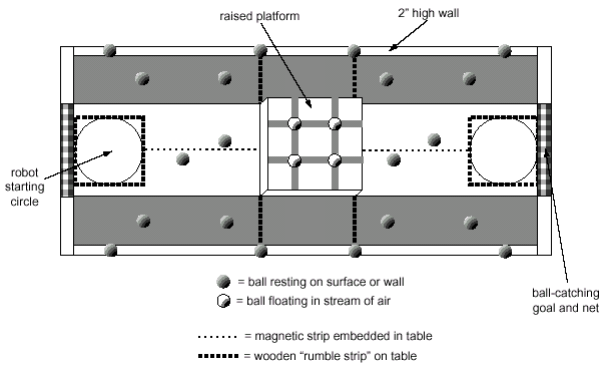
```
void left (int pid) {
left_motor[pid]= -100;
right_motor[pid]= 100;
}
void halt (int pid) {
left_motor[pid]= 0;
right_motor[pid]= 0;
}
/* sensor functions */
int left_touch () {
return digital(LEFT_TOUCH_PORT);
}
int right_touch () {
return digital(RIGHT_TOUCH_PORT)
}
```

Copyright Prentice Hall, 2001

# Using Reactive Control: Robo-Miners

- **Table Structure:**
  - had central raised platform with air hoses underneath
  - At start of game, balls were placed in each of the four air streams emanating from hoses
  - Additional balls placed at various other points on the table surface and edges

1995 MIT Robot Design contest




- **Navigating:**
  - Use downward-facing **reflective light sensors** on white/black areas on table, cross-hatched pattern on raised platform
  - Use **Hall-effect sensors** to detect magnetic stripping embedded in cable from starting circles to raised platform
  - Use **touch switches** to detect Wooden molding "rumble strips"
- **Scoring:** points awarded for collecting balls, depositing balls into one's goal area, and placing collected balls into the air streams

Copyright Prentice Hall, 2001


# Using Reactive Control: Robo-Miners

## The Arm

- Algorithmic Control Strategy
- **Structure:** a small car at the end of a lazy tongs arm.
- **Strategy:** the car would drive out of a nest, scoop up one ball, and drive backward into the nest while the lazy tongs held it straight. The nest would then rotate the car-arm assembly into position to grab the next ball, using shaft encoding to turn to predetermined "correct" angles.
- **Problems Encountered:**
  - the nest's base would rotate with respect to the playing surface, thus rendering subsequent rotations incorrect. This was solved by stretching LEGO tires as big rubber bands.
  - the robot was highly sensitive to correct initial placement
  - the ball-grabbing mechanism at the end of the car-arm extension didn't perform well
  - the overall mechanism was not terribly reliable
- Ultimately, the robot did quite poorly in the contest, as what seemed like an easy win ("just reach out and grab the balls; we know where they are") turned out to be difficult to perform in practice

Copyright Prentice Hall, 2001

## Using Reactive Control: Robo-Miners

### Fluffy the Sunshine Robot

- Reactive Control Strategy

• Designed in two or three *days* before the contest, was the result of a team of students discarding three weeks' of effort trying to get their algorithmic robot to work with any degree of reliability.

• **Strategy:** absurdly simple: wandered around the playing field, scooping up balls that it happened to run over. It had two touch sensors, and would back up and turn when either of them were triggered.

• On the one hand, *Fluffy* was **completely unpredictable:** one never knew which balls it might collect.

• On the other hand, it was **incredibly reliable**—because it was so simple, it *always* got at least a few balls.

• By the scoring method of the contest, simply collecting balls scored the least number of points; returning them to your goal or placing them in the air stream scored many more points.

• During the contest itself, even though all it could do was collect a few balls, *Fluffy* surprised everyone by ***tying for second place*** overall.

• Nearly all of the complex, algorithmic robots were at least as likely to fail completely and score no points as they were to score a sizable bounty, and *Fluffy* almost carried the day.

Copyright Prentice Hall, 2001


## Using Reactive Control: Robo-Miners

### Comments

• While *Fluffy* may have seemed like an isolated phenomenon to the casual observer, it was not.

• Just about every year of the MIT contest, there are one or two robots with a similar story behind them as *Fluffy*: students who became frustrated with repeated failures during the progress of their algorithmic robot design, and decided to rebuilt a simple robot from scratch.

• With almost no time remaining, the only approach that seems viable is the **reactive** one (though students are not consciously making a decision between algorithmic and reactive).

• The surprise fact is that a last-ditch reactive machine comes in second or third place nearly every year of the MIT contest. The culture of the MIT contest is heavily weighted toward algorithmic machines, so the successes of the reactive machines are typically blamed on luck—people don't remember that in previous years, reactive machines have also done well.

• Students in the MIT class tend to blame performance failures on particular component failures or unexpected circumstances, rather than re-evaluating their overall control strategies.

• It requires a new way of thinking to design a system that works properly only as the result of many small interactions rather than a master plan.

Copyright Prentice Hall, 2001

# Using Reactive Control: Soda Can Collector

• While a graduate student at the Artificial Intelligence Laboratory at MIT, Jonathan Connell created a robotic arm that collected soda cans, using the behavioral control principles of Rodney Brooks.

• Connell's arm controller was based on a collection of 15 independent behaviors that operated with six levels of priority:

– **Grab** (which closed the hand anytime something broke a light sensor beam between two fingers)

– **Excess** (which prevented the hand from squeezing too hard)

– **Extend Over** (which helped the arm move out and above a soda can)

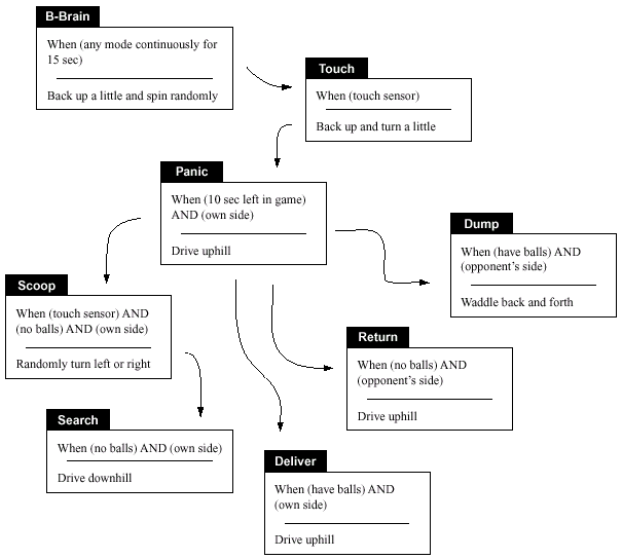
– **Home** (which brought the arm near to the robot's body)

• Designed in the early 1980's, when today's advanced 32–bit microprocessors were prohibitively expensive and complex, Connell's robot used a collection of simple 8–bit microprocessors — indeed, the same one used in the **Handy Board** — wired into a local network.

• Each microprocessor ran a couple of the behavior tasks, and if the robot needed computational power for new behaviors, Connell simply plugged in a new microprocessor board.

Copyright Prentice Hall, 2001


# Using Reactive Control: Reactive Groucho




Copyright Prentice Hall, 2001

## Using Reactive Control: Reactive Groucho

- Would it be possible to devise an effective reactive control program for *Groucho's* task?
- Eight separate behaviors at five priority levels.
- Main behaviors for getting work done:
  - **Search** activates when *Groucho* is on its own side and isn't carrying any balls, and drives the robot downhill—toward the ball trough, where balls should be waiting.
  - **Scoop** is intended to run after Search, when *Groucho* reaches the far wall on its side. Then Scoop triggers and turns *Groucho* so that it can pick up some balls.
  - **Deliver** should trigger next, causing *Groucho* to drive uphill.
  - When the robot drives over to the opponent's side **Dump** activates, releasing the balls.
  - Then **Return** can become active, when *Groucho* notices it is on the opponent's side and has no balls.

Copyright Prentice Hall, 2001


## Using Reactive Control: Reactive Groucho

- Three **supervisory processes** help make sure *Groucho* doesn't get stuck as it performs its task:
- **Panic** will drive the robot onto the opponent's side (if it isn't already there) when the contest round is about to expire, as a safety measure
- **Touch** makes sure that *Groucho* does not get wedged if it runs into a wall unexpectedly
- **B-Brain** monitors all of the other behaviors and executes an emergency "get unstuck" action if it notices that *Groucho* has been lodged in the same task for too long.
- **Exercise:**

  (a) This collection of behaviors for getting *Groucho* to perform its job has not been tested. Do you think it would work? Why or why not? What are some problems with the approach? How can they be remedied?

  (b) If you are not confident that the solution presented is viable, invent a different collection of behaviors that would be effective in getting *Groucho* do perform the ball collection and transportation task.

  (c) With a *Groucho*-style robot and a *Robo-Pong* playing field, implement your choice of the solution presented, the solution with your modifications, or your approach (whichever you have the most confidence in). What surprised you as you tested the solution?

Copyright Prentice Hall, 2001

## Extending the Prioritization Framework

• Two different extensions to the prioritization program

•**Dynamic Priorities.** The framework presented uses static robot task priorities that are established at start-up time and do not change. But there is no reason that a task's priority level could not be a dynamic quantity which varied depending upon various internal and external factors.

• Consider an example based on the *Groucho* robot task:

– Suppose that *Groucho* had several different strategies for searching for balls, some more conservative and others more aggressive and risky.

– At the start of the round, it would make sense to try the conservative approaches first, but if these failed, to switch to the more aggressive search modes.

– A supervisory task could monitor the *Groucho*'s performance, and elapsed time, and adjust other tasks' priority levels as it saw fit.

• **Exercise:** Describe other ways that *Groucho* could be improved with a dynamic task priority structure.

Copyright Prentice Hall, 2001


## Extending the Prioritization Framework

• **A-Brain, B-Brain.** In his famous book The Society of Mind, Marvin Minsky present a particular model of cognitive operation that he terms *A-Brain and B-Brain:*

– **A-Brains:** Suppose one part of the brain is directly connected to a creature's sensor and motor apparatus, and is responsible for performing sensory-motor tasks such as hand-eye coordination.

– **B-Brains:** Other parts of the brain have no direct connection to the sensory-motor apparatus, and are only connected to various A-Brains. B-Brains are perfectly situated to monitor how well the A-Brains are doing, and should be able to notice unproductive loops or other failure modes that the A-Brains may have without them realizing it. B-Brains are responsible for stimulating and suppressing A-Brain function to achieve maximum benefit for the organism.

• This idea fits wonderfully into the reactive model of robot control. The reactive *Groucho* include a simple B-Brain component that would notice if *Groucho* was stuck in a single behavior for too long.

• **Exercise:** Based on Minsky's concept, generalize this approach. For example, write a B-Brain function that notices if *Groucho* were to go back and forth between a pair of behaviors for several iterations. What other kinds of unproductive loops could be recognized? How would you do so?

Copyright Prentice Hall, 2001