MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo 1120 **July, 1989**

Squirt: The Prototypical Mobile Robot for Autonomous Graduate Students

Anita M. Flynn
Rodney A. Brooks
William M. Wells III
David S. Barrett

Abstract

This paper describes an exercise in building a complete robot aimed at being as small as possible but using off the shelf components exclusively. The result is an autonomous mobile robot slightly larger than one cubic inch which incorporates sensing, actuation, onboard computation and onboard power supplies. Nicknamed Squirt, this robot acts as a "bug", hiding in dark corners and venturing out in the direction of last heard noises, only moving after the noises are long gone.

The original point of this exercise was to show that even at such a small scale, the bulk of the mass of a mobile robot resides in the power supply and actuation systems, whereas the componentry in which research is often focused, the sensors and intelligence, is confined to a very small volume. By shrinking the brawn down to meet the brain, we can build very tiny intelligent robots. The result is that the end product becomes cheaper and therefore much more likely to make an impact.

Interestingly, this exercise of reducing a robot to its bare essentials has led directly to spinoffs (at the AI Lab) of making robots readily available to the entire graduate student population. While motors for elbow or shoulder joints of typical research manipulators often cost hundreds or even thousands of dollars, the computer and motor for Squirt were approximately $25 each. Squirt-style technology is powerful in that it delivers a package which consists of just enough hardware (sensing, actuation and programmability) to enable a student to get an initial hands-on feel for research issues related to developing intelligence, without undue cost, complexity or overhead. Within three months after starting the Squirt project, the AI Lab hosted a Robot Talent Show in which nearly 60 people participated to produce over 20 autonomous mobile robots. Such easy access to technology grants the opportunity to connect perception to action (we're shooting for a robot in every office), encourages experimentation, inspires creativity and leads the way to new ideas.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the research is provided in part by the University Research Initiative under Office of Naval Research contract N00014-86-K-0685 and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014-85-K-0124. The Robot Talent Show was funded by the Systems Development Foundation.

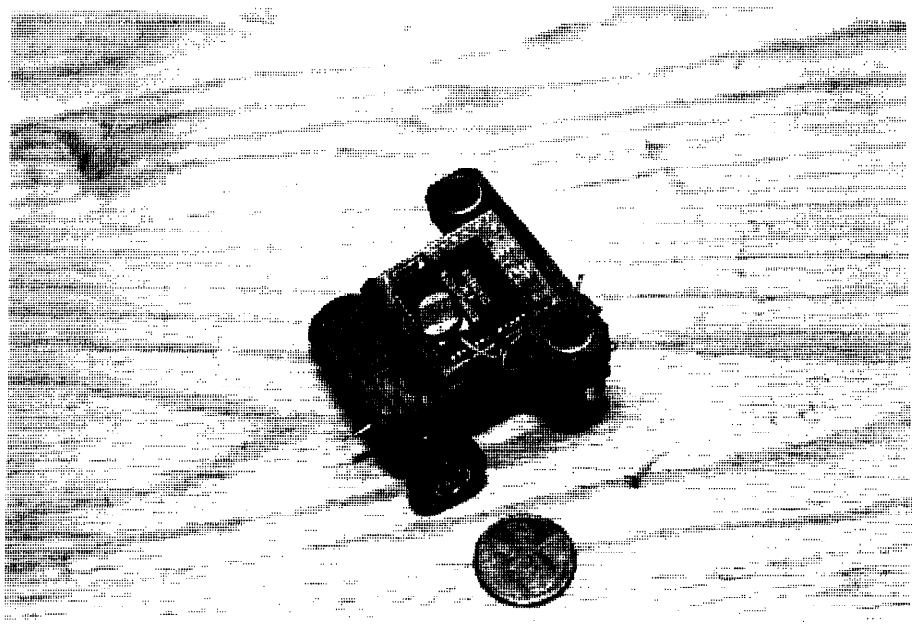

Figure 1. Squirt is a completely autonomous mobile robot in a volume slightly larger than one cubic inch. It carries onboard a microprocessor, three sensors, one motor and two lithium batteries.

Introduction

Artificial intelligence programs such as inference engines, reasoning programs or image understanding systems typically require large amounts of computational resources and memory, producing a trend towards supercomputers or massively parallel machines. In intelligent robot control systems, one common tack is to fuse data from various sensors into an internal world model of the robot's environment, using the model produced to plan intelligent action. This process of sensor fusion and map-making can be computationally intensive and hinder real-time response to emergencies.

An alternative approach, and the one used here, is to ignore sensor fusion and build intelligent robot behavior by layering many simple behaviors one on top of the other. This organization, termed the "subsumption architecture", directly connects perception to action without building internal world models. Sensor inputs, in a "sensor fission" approach, trigger distinct behaviors where some behaviors override other behaviors, depending on the layout of the control system. By using many simple sensors and actuators suitably combined, various types of intelligent robots have been designed, from walking machines to soda can retrievers. This approach to dealing with the computational aspects of intelligent control systems has led to the discovery that if the computations are well understood and the organization is well laid out, the actual amount of computational resources required becomes very small.

As such an intelligence system can be made to fit into a small amount of silicon, it seems feasible to imagine lowering costs for a given level of intelligence by scaling down an entire robotic system to the scale of the control system. As both electronics and sensors have become entrenched in the micro domain [Petersen (1985)] and with the onset of microactua-

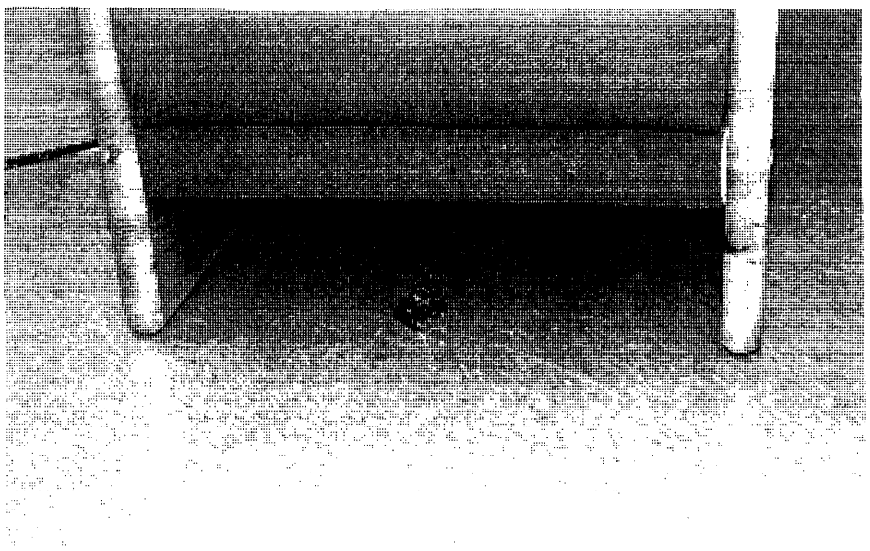

Figure 2. Squirt is shown here hiding under a chair. A photocell has triggered his behavior to move in a spiraling pattern if the light level is above a certain threshold. Once it falls below that threshold, Squirt stops. His spiraling behavior happened to have landed him under this chair where it is dark, so he has stopped.

tors [Muller (1987)], [Bart et al (1988)], [Fujita and Omodaka (1987)] and [Trimmer and Gabriel (1987)], it is possible that robotics as a whole should one day see the same drastic size and cost reductions. By integrating sensors, motors, intelligence and batteries onto a single piece of silicon, it should be possible to mass produce "gnat" robots in the same way integrated circuits are mass produced, accruing tremendous cost saving due to batch fabrication techniques [Flynn (1987)]. With lowered costs, a robotic system with a given level of intelligence becomes potentially more useful [Brooks (1987b)], [Flynn (1988)].

As microfabrication of actuators for propulsion systems remains a technological hurdle, we have undertaken the exercise of building the smallest robot physically possible with off the shelf technology, concentrating primarily on the organizational layout of the computations required to connect perception to action in a completely autonomous system.

Squirt; A One Cubic-Inch Robot

The resulting vehicle, which we've nicknamed Squirt, is shown in figure 1. Squirt's programmed behavior is to hide in the dark listening for sounds. It uses two microphones to localize the direction of noises. When all is quiet for a while it ventures out in the direction of the most recent noise, and after traveling for a specified time, it uses a spiral search to find a new hiding place. The end effect is that it gravitates towards the center of action. Figure 2 shows Squirt hiding under a chair, having found a dark corner after completing his travels.

The basic design strategy for maintaining small size was to keep all hardware extremely simple in order to arrive at the software stage as soon as possible. Thus the final design uses only one actuator to both drive and turn, with a one square inch printed circuit board to house a computer and all the analog circuitry for the auditory system, the motor drivers and a monocular light sensor. With an efficient compiler for downloading a subsumption architecture control system to the target computer, all the code fits in under 2K bytes of memory. As the target computer contains 2K EEPROM onboard, no external memory chips are required on the printed circuit board.

Mechanics

Squirt's overall mechanical design goal was to squeeze all of the components required for autonomous locomotion into a one cubic inch package, a requirement that produced very cramped quarters. The components needed were the one square inch 6811 circuit board for control, two lithium batteries for power, two microphones plus a photocell for sensing, a DC motor (Swiss-made Micro-Mo 1016N0066) for propulsion, a two-state controllable steering mechanism, wheels to roll on, and finally, a chassis to hold all the components together. Since the motor and battery alone comprised more than 70% of the available one cubic inch volume, fitting in the rest of the components was quite a challenge.

The biggest design dilemma this presented was how to package a steering mechanism in the remaining space. To achieve the design goal, Squirt only really needed two rudimentary steering states, either to go straight or to turn with a fixed radius. Adding a second motor for steering was out of the question. The motor used for propulsion was the smallest off the shelf motor commercially available (even so, it was much larger than needed, supplying 1/2 watt when only 1/20 watt would suffice). Microhydraulic cylinders and micropositioning solenoids have yet to be developed.

Passive steering devices like flip-flopping castors and tilting tricycles were prototyped without much success. The only practical option left was to derive steering from the main drive motor through some form of transmission. If both drive wheels could be synchronously powered as Squirt traveled forward, he would move in a straight line. If one wheel could be locked up while the second was powered as Squirt backed up, he would turn at a fixed radius.

Luckily, there is a mechanical device in the form of a unidirectional clutch that can be used to build just such a transmission, and which can be built very compactly to fit into tiny spaces. A unidirectional clutch has the fundamental property of transmitting torque to a shaft if rotated in one direction while slipping freely on the same shaft if rotated in the opposite direction.

This property (see figure 3) was exploited in Squirt's transmission in the following way. Squirt's drive motor runs through a Micro-Mo 16:1 planetary gearhead and then through a 22:1 worm gear to both slow the motor speed and raise the torque to usable ranges. This arrangement also facilitates rotation of the shaft's axis of revolution 90 degrees to allow both the motor and transmission to fit into Squirt's tight space constraints.

The drive wheel axle is broken in two at the one way clutch. The worm gear is fixed to the right hand shaft, as is the right hand drive wheel. This means that the right hand wheel is always powered.

The left hand shaft is attached to both the one-way spring clutch and a one-way locking brake, as well as the left hand drive wheel. This means that the left hand wheel either turns synchronously with the right hand wheel as Squirt travels forward or it declutches and locks to the body as Squirt backs up.

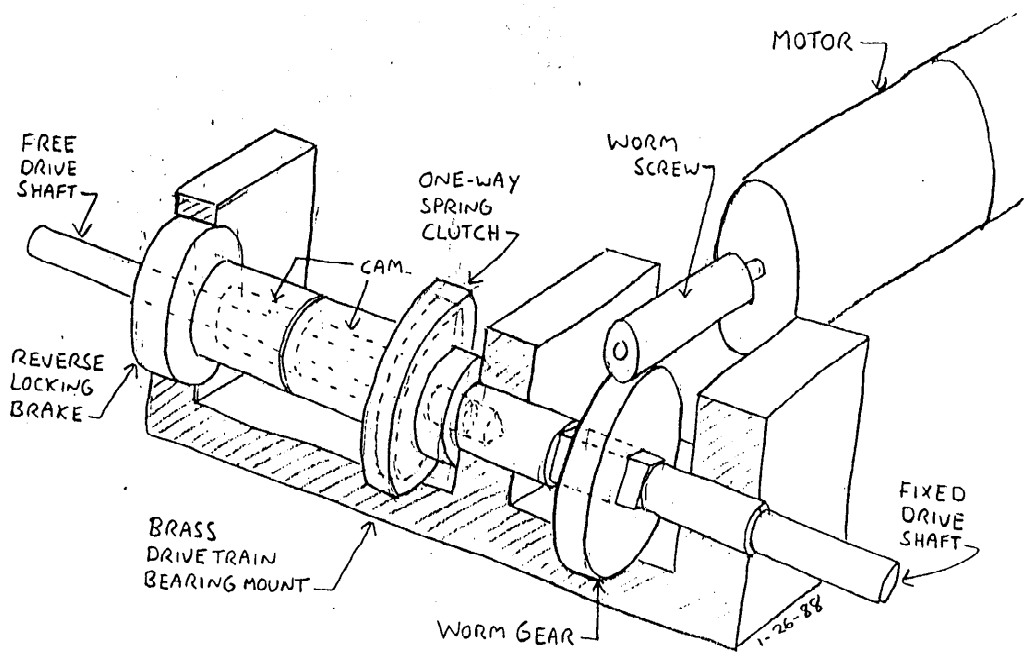

Figure 3. A clutch built into the rear axle allows two dimensional movement on the plane while utilizing only one motor, by causing the robot to either go forward or back and turn.

The combination of the two mechanisms allow Squirt to steer, simply by changing drive motor direction. Although not perfectly ideal for navigating, the main advantage of this transmission was that it could be packaged in a very small (1/4" square x 1" wide) volume with room to spare. In addition, after struggling with precision machining techniques to make the clutch ourselves, we found a very serviceable micro-clutch in a toy race car and with only slight modifications were able to incorporate it into the actual design.

Having dealt with steering, the rest of the design was fairly straightforward. Squirt's chassis was made from a single monolithic block of MDS nylon, with all the the component bays, mounts and wire ways essentially machined in place. This circumvented the problem of microfasteners and microbrackets and made for an elegantly simple body. MDS nylon was chosen for its lubrication, machinability and stealth properties.

Squirt's main drive wheels were taken from a toy four wheel drive truck (for traction) and his nose is supported by a simple castor. This arrangement allows the robot to go straight when moving forward and to pivot freely when backing up. When loaded with all components, Squirt fits neatly into about a 1 1/4 cubic inch package.

The largest mechanical problem run into in testing was the slipping of the drive wheels. On such a light vehicle, good traction was hard to come by and most experimental runs were troubled by rear wheel slippage when attempting to back and turn. Future designs might benefit from a more comprehensive traction analysis or from legged locomotion.

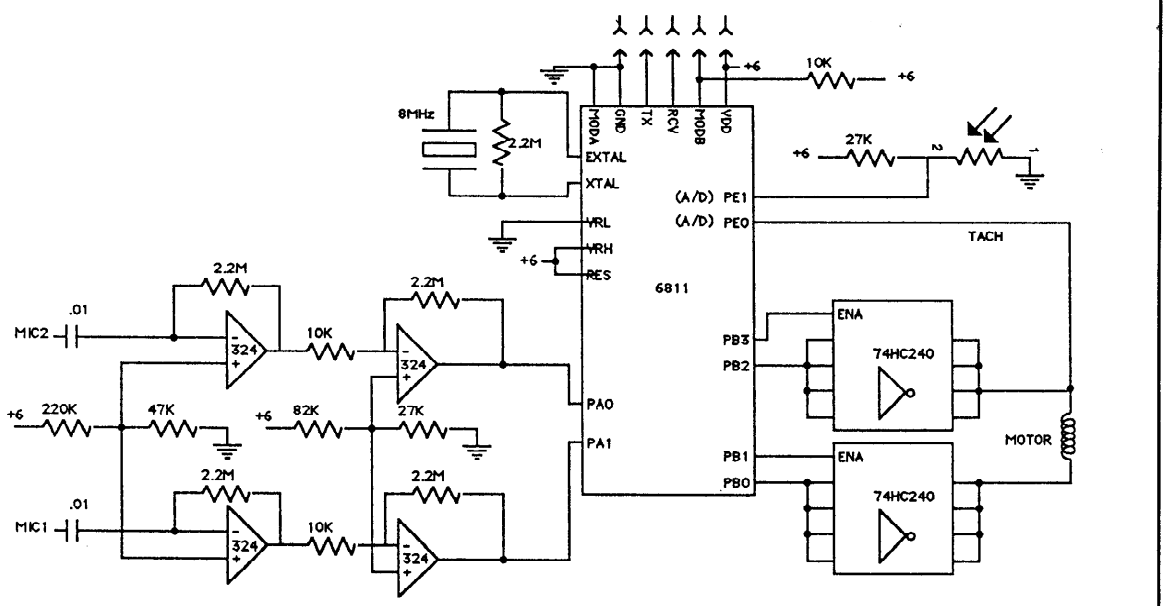

Figure 4. Squirt's control system is composed of a 6811 microprocessor with two CMOS bus driver chips piggybacked to form an H-bridge for driving the motor. One quad op-amp and a voltage divider interface the microphones and photocell.

Electronics

As with the mechanical system, the control electronics were designed to be as minimal as possible in order to conserve space. Again, the main strategy was to keep hardware lean and let software hoist most of the burden of the control system. The electrical hardware is a simple example of a complete control system with computer, sensors, and effector, yet is laid out on a one square inch printed circuit board. Figure 4 shows the circuit diagram making up Squirt's control system. Oddly enough, the small connector used to download code from the host Macintosh is the most expensive component on the robot.

In order to further save space on the board, the MAX233 level translator chip for serial port communications was taken off the board and actually built into the downloader cable. In addition, the downloader cable was designed such that plugging it in pulled the mode pin low, which set the microprocessor to downloading mode. Removing the cable set the 6811 to single-chip run mode. This saved space that would otherwise be required for a switch on the board.

The one square inch circuit board is a two layer board with surface mount components on both sides, shown in figure 5. There is a download and debugging connector on the bottom of the board for loading software developed on a host Macintosh, an 8 bit microprocessor, and three separate analog circuits for the two sensor systems and the motor. The CPU is a Motorola XC68HC811A2FN. This 6811 is a CMOS single chip microprocesssor that has 2K bytes of EEPROM, 256 bytes of RAM, 24 logic I/O lines, 8 analog-to-digital converter inputs, a serial port, and various counter/timer control lines. One advantage of using this chip is that connecting sensors directly to the A/D or timer inputs alleviates the need for

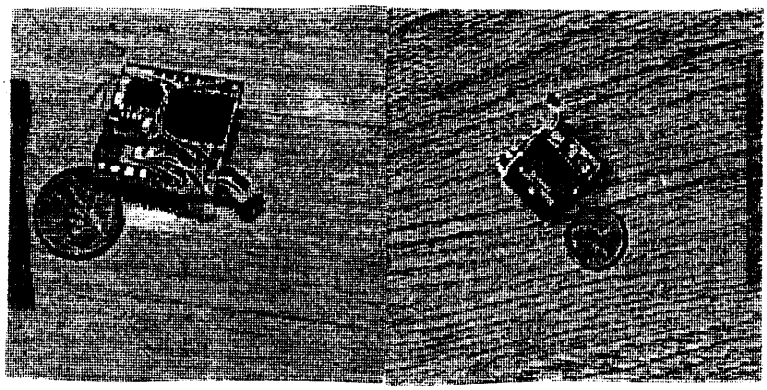

Figure 5. The one square inch printed circuit board holding a complete 8-bit microprocessor along with its interfacing circuitry to motors, sensors and host computers, is shown here, front and back.

external circuitry to do thresholding, etc. In addition, onboard EEPROM means no external equipment is required to burn EPROMs or download code to non-volatile RAMs.

The motor is driven with an H-bridge realized with CMOS bus driver chips mounted on the underside of the board. Because the motor is so small and draws such small levels of current (30mA) we can get away with using two paralleled 74HC240s in surface mount packages to realize this H-bridge. Although a surface mount package is fairly small, in order to save space, we sanded down the thickness of the chips and soldered all the legs together in a piggyback fashion. Through pulse code modulation velocity can be controlled by monitoring the back EMF of the motor between pulses and feeding that signal into an A/D input to close the loop.

Squirt uses two sensors to achieve his "bugging" behavior. The first is a photocell (a light dependent resistor) connected to one of the 6811's A/D inputs. It provides simple light level information to the lowest level of Squirt's control system which triggers his behavior to seek dark corners.

The second sensory capability is a crude directional auditory sense. Signals from two microphones are conditioned into logic signals using amplifiers and Schmitt triggers. The logic signals are connected to timer/counter control inputs on the 6811 and the time difference in their edges is measured in software, thus providing a rough measurement of orientation toward sources of loud noise. The analog circuitry is adjusted so that typical background noises such air conditioning, are ignored. The round microphones can be seen in figure 1 to the right and left of the circuit board. The photocell is resting on top of the microprocessor.

Batteries

For power supplies, the robot uses two 3V lithium batteries, DL1/3Ns from Duracell, which provide 160mAh capacity. By using a CMOS processor and 74HC series logic for motor drivers, it is possible to run all the electronics directly off the 6V provided by the batteries, without any need for voltage regulators or DC-DC converters. This both saves space and avoids any voltage drops and consequent waste of battery energy.

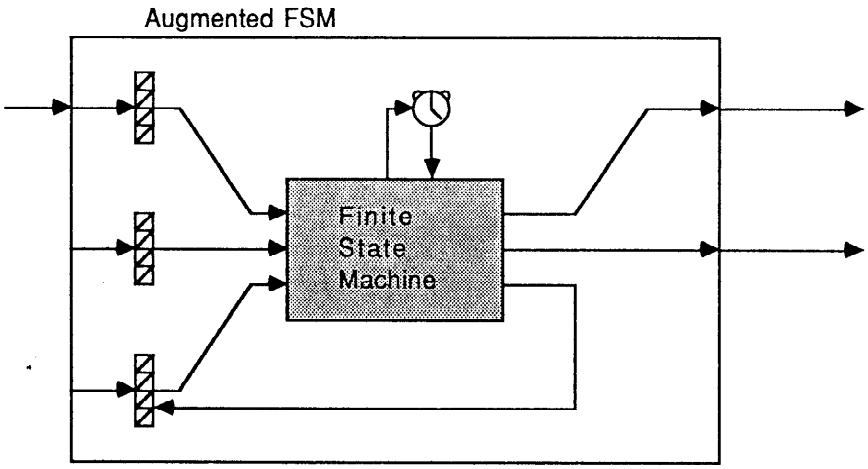

Figure 6. Augmented finite state machines are the essential building blocks of the subsumption architecture. Each AFSM is a finite state machine augmented with some timers and registers. These building blocks are wired up into networks using arbitration mechanisms called suppressor and inhibitor nodes. These are fixed topology mechanisms which allow higher levels in a carefully designed network to interject a change of control.

Each augmented finite state machine (AFSM), figure 6, has a set of registers and a set of timers, or alarm clocks, connected to a conventional finite state machine which can control a combinational network fed by the registers. Registers can be written by receiving messages form other augmented finite state machines over wires connecting the two. The messages get written into the registers and replace any existing contents. The arrival of a message, or the expiration of a timer, can trigger a change of state in the interior finite state machine. Finite state machine states can either wait on some event, conditionally dispatch to one of two other states based on some combinational predicate of the registers, or compute a combinational function of the registers directing the result either back to one of the registers or to an output of the augmented finite state machine. Some AFSMs connect directly to robot hardware. Sensors deposit their values to certain registers, and certain outputs direct commands to actuators.

A series of layers of such machines can be extended by adding new machines to the control system and connecting them into the existing network in the ways shown in figure 7. New inputs can be connected to existing registers, which might previously have contained a constant. Additionally, new machines can inhibit existing outputs or suppress existing inputs by being attached as side-taps to existing wires. When a message arrives on an inhibitory side-tap (figure 7, circled "i") no messages can travel along the existing wire for some short time period. To maintain inhibition there must be a continuous flow of messages along the new wire. (In earlier versions of the subsumption architecture [Brooks (1986)] explicit, long,

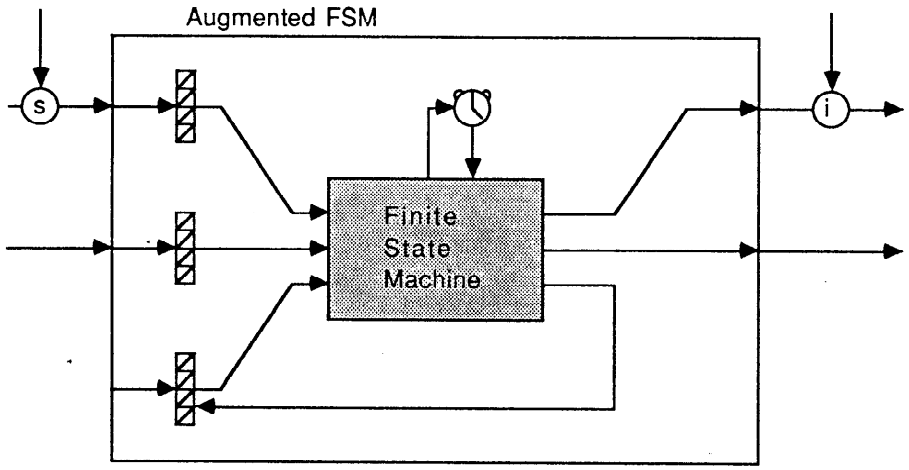

Figure 7. The subsumption architecture is a distributed network made up of augmented finite state machines, carefully designed and laid out. A fixed-priority arbitration mechanism is implemented through the use of suppressor, inhibit and default nodes which gate through different signals depending on the semantics of the node. These mechanisms allow higher level behaviors to guide lower level actions.

time periods had to be specified for inhibition or suppression with single shot messages. Recent work has suggested this better approach [Connell (1988)].) When a message arrives on a suppressing side-tap (figure 7, circled "s"), again no messages are allowed to flow from the original source for some small time period, but now the suppressing message is gated through and masquerades as having come from the original source. Again, a continuous supply of suppressing messages is required to maintain control of a side-tapped wire. One last mechanism for merging two wires is called defaulting (indicated in wiring diagrams by a circled "d"). This is similar to the suppression case, except that the original wire, rather than the new side-tapping wire, is able to wrest control of messages sent to the destination.

All clocks in a subsumption system have approximately the same tick period (0.04 seconds is typical among our robots), but neither they nor messages are synchronous. The fastest possible rate of sending messages along a wire is one per clock tick. The time periods used for both inhibition and suppression are two clock ticks. Thus, a side-tapping wire with messages being sent at the maximum rate can maintain control of its host wire.

Specifics of Squirt's Software

One of the key technical advances which has enabled us to build Squirt has been the development of new compilation techniques for the subsumption architecture, enabling it to be compiled down to very small microprocessors. Since subsumption programs consist of a

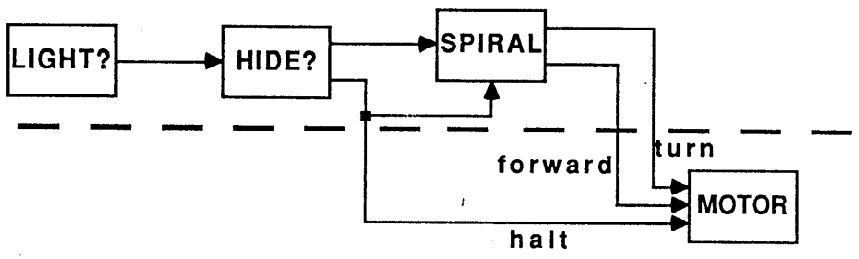

Figure 8. Squirt's lower two levels of behavior are pictorially represented by this subsumption diagram which depicts the layers of Motor Control and Flee-Hide.

set of asynchronous processes communicating over fixed topology wires, the challenge on a single processor is to adequately simulate these processes providing responsive service to high priority items while not starving out lower priorities. The solution has been to compile a special purpose scheduler for any particular subsumption program rather than use a general purpose scheduler and make runtime calls to it from compiled code.

The compiler we have produced is dynamically retargetable to many different processor architectures, while at the same time making use of both machine independent and machine specific optimizations. This is compiled into highly optimized native code with each output statement being open coded to deliver messages to target processes and setting appropriate message arrived flags.

The most primitive behaviors for controlling Squirt are shown in figure 8. At the lowest level, a single process (or augmented finite state machine) named MOTOR simply fields requests for the robot to go forward, to turn some angle in place, or to halt. It ignores requests that are inappropriate; e.g., if it is busy traveling forward it will ignore a request to turn in place. It is the responsibility of higher level layers to monitor the state of the robot and to only ask for sensible things.

The second behavior that is added is one to flee from light and hide in the dark. It consists of three processes which monitor the light level, cause the robot to stop when it is in the dark and to go in a spiral pattern whenever it is in the light. Effectively these three simple processes enable the robot to search for a dark hiding place.

The final layer, at the time of writing, is implemented as four processes, and is depicted in figure 9. The first, HEAR, monitors sounds and sends out messages indicating the direction from which they came. DRIVE records their direction and then if all has been quiet for some long time period it causes the robot to orient towards the last heard sound. GO then causes Squirt to drive in that direction suppressing the urge to hide for a little while. Once GO has completed its job the robot will most likely find itself out in the light and scurry for cover. IDLE? notices when the robot has been idle for a time and resets the listening behavior to watch out for new sounds to once again follow when the coast is clear.

At the time of writing, this is the complete operational program of Squirt, occupying slightly

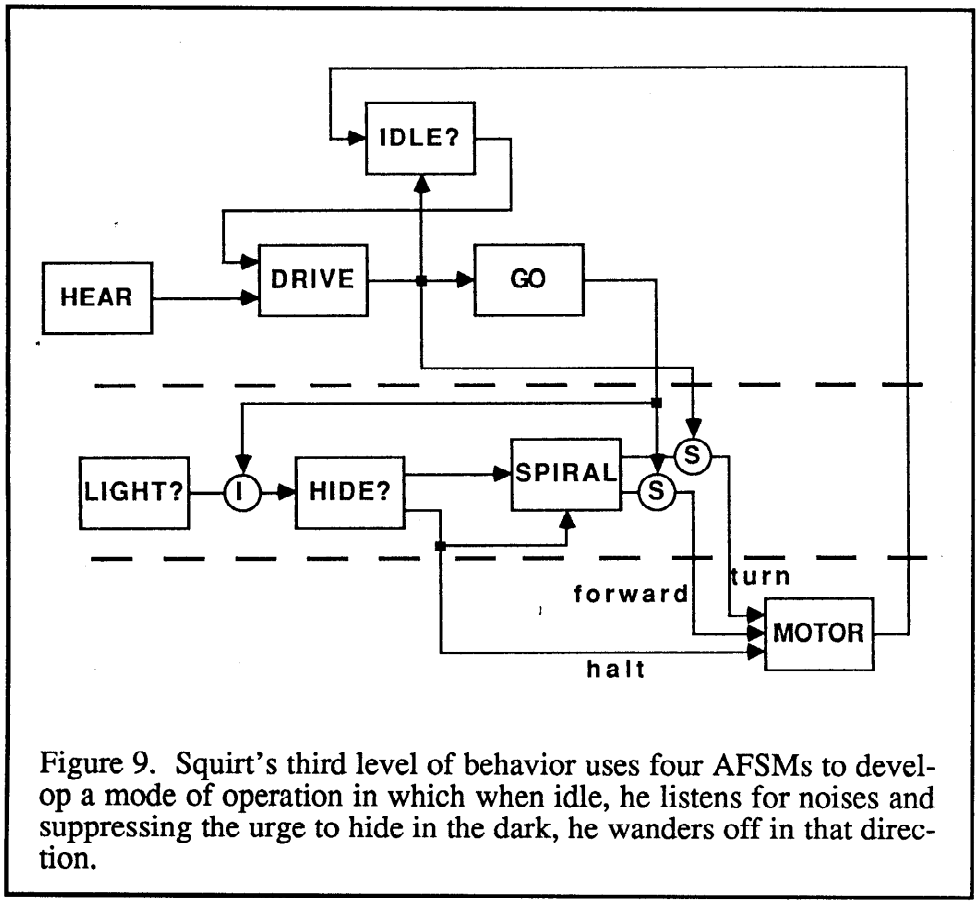

Figure 9. Squirt's third level of behavior uses four AFSMs to develop a mode of operation in which when idle, he listens for noises and suppressing the urge to hide in the dark, he wanders off in that direction.

less than 1300 bytes of EEPROM, but it is easy to see how to add more functionality within the subsumption architecture. In particular, there is a stall sensor built into the processor board and so an additional independent control layer could be added to monitor for this condition, and depending on whether the most recent action was to go forward or to spin backwards in place, a set of heuristic behaviors could be generated to unwedge the robot. At small scales such as this, obstacle avoidance capabilities are not needed, but ways of getting out of a jam are.

We've observed just such behavior in a few test runs. Every once in awhile, after a series of spirals and ballistic trajectories, Squirt runs head first into a chair leg. If he is in the middle of a ballistic trajectory, he continues ramming the obstacle but once that times out and the spiraling behavior kicks back in, he backs up and turns, eventually freeing himself.

In general, Squirt reliably wanders in a spiral, stopping where it's dark. He also reliably moves after hearing a pattern of sharp noise followed by a short period of silence. However, his orienting behavior and ballistic trajectory towards the source are often off, as the wheels routinely slip.

Future Implementations

The control system for the next generation tiny robot could be reduced in size substantially

Figure 10. The AI Lab turned out en masse to see what creativity had spawned at the Robot Talent Show. Here Lee Tavrow displays his inchworm, which sported exactly one bit of intelligence yet inched along without recourse to wheels.

participants had until the end of the traditional AI Lab Olympics (January 27th, 1989) to build a robot of their design. Everyone who built a robot then demonstrated it at the Robot Talent Show [Flynn et al (1988)], [Flynn (1989)]. Over 60 people got involved in building robots, including graduate students, undergraduates, faculty, staff, people from other labs around MIT, and even groups from other universities. Figures 10-12 give a snapshot of some of the creatures produced: a Lego inchworm, a sonar-laden blimp and a talking laser-tag machine.

There were several factors which made the Robot Talent Show a success; some were technical, others were organizational.

Technical Timing

Technically, much of the infrastructure developed in the process of designing Squirt turned out to be easily transferable from the Mobile Robot Lab to the AI Lab as a whole. Specifically, the choice of using the 6811 microprocessor on Squirt turned out to be the main enabling technology for the Robot Talent Show. For Squirt, it was chosen because it had onboard EEPROM which meant we could get by without any external memory chips and thus fit the entire brain into one square inch. For the Talent Show, it meant that no PROM programmers, downloader boxes or expensive emulator systems had to be provided to people in order for them to develop code. Since programs could be developed on a host computer and directly downloaded over a serial cable into the chip, everyone could effectively work in their offices on their workstations, programming their robots.

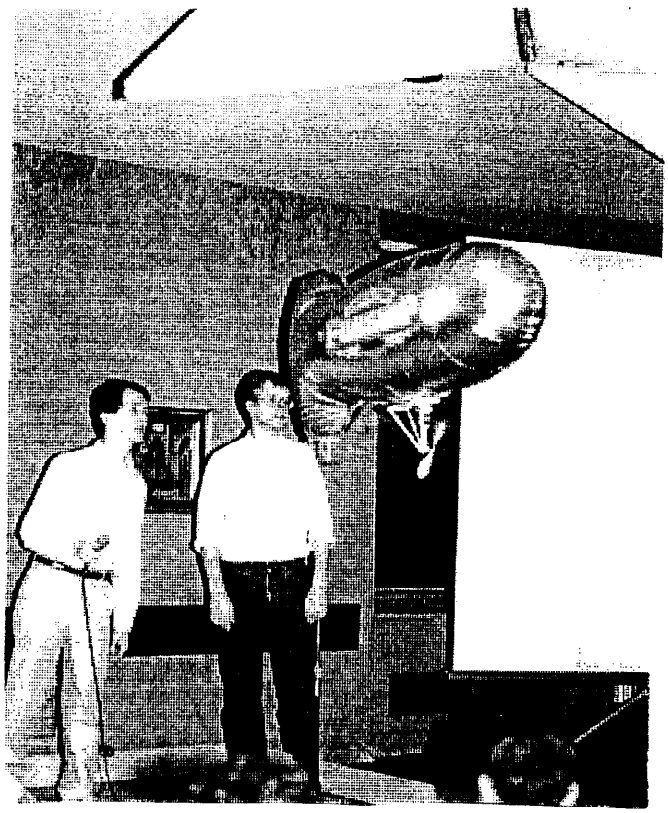

Figure 11. An autonomous blimp even made its way into the Talent Show. Mike Caine and Andy Christian describe how they fit a sensor, motor, computer and battery into a 100g payload.

The development cycle was further facilitated by fabricating printed circuit boards for the 6811 brain and porting the assemblers and compilers written for Squirt, originally on Macintoshes, to general lab workstations. As everything was written in Common Lisp, this was relatively straightforward.

Another technical choice for the Talent Show was to stick with simple sensors. Although AI has been devoted to vision for years, vision is a very complicated sense and the loop seldom gets closed. We got around the hard problem of visual sensing by handing out a wide variety of simple sensors. Kits contained everything from microswitches, microphones and photocells to infrared emitters and detectors, gyros and compasses. Participants could then pick and choose as they desired.

Finally, the strategy employed on Squirt, of keeping hardware simple and getting to the software stage as quickly as possible, was transferred to the Talent Show. To avoid the need for custom machining of mechanical components, we bought dozens of kits of mechanical building blocks; high-tech Legos that came with everything from rack and pinion steering and differential drives to motors and optical encoders. In addition, we encouraged participants to spend as much of their own money as they desired. Thus it was possible to start with a com-

Figure 12. Colin Angle describes his team's laser-tag robot, Clint, which announced "Dead Meat", whenever it found another robot in its sights. Clint then directed fire with an infrared emitter.

mercial chassis and add sensors and computation.

Technically, the timing was right for something as involved as a lab-wide robot talent show. VLSI technology had produced small single chip computers, Legos had come a long way since we were kids, subsumption research had produced lean control systems for mobile robots, and a large infrastructure in both software and hardware development was already in place. Nevertheless, technology could not have been the only reason for the excitement generated during that month of January.

Organizational Theories

What was it that drove people to put in hundreds of hours working on their robot during a month when theoretically, they could have been catching up on sleep?

Whatever it was, we'd like to grab it and keep it around.

Maybe it was simply the fact that we gave out free toys. A programmable gizmo with both sensors and actuators is in some sense the ultimate graduate student toy. The opportunity to get a hands-on feel for real world problems in building robots was possibly very appealing.

On the other hand, maybe it was the general environment of creativity that was spawned. The Talent Show was completely open ended. There wasn't any set feat which the robots had to accomplish. Rather, the theme was to pick your own problem and solve it, much in the spirit of graduate research versus undergraduate problem sets. The general thrust was to

try to build something clever which connected perception to action. Beyond that, it was up for grabs. The lone rule was that there would be "No Squashing", which meant no one was allowed to debunk anyone else's idea during the conception stage.

Again on the other hand, maybe the Talent Show provided the perfect escape to thesis procrastination (. . . a long and venerable tradition at the AI Lab. It's amazing how productive one can become when procrastinating on one's thesis). Diversion from the stated plan seems to be a cornerstone of winning organizations. As Ross Perot once described the manner by which his company stays on top, "We get up in the morning with a plan . . . then we weave, bob, duck, jump over fences . . ." [Silver (1989)]. To maintain a creative environment, the organization has to be reactive. Fortunately, the AI lab always seems to be flexible towards spontaneous ideas like staging this Robot Talent Show.

Conclusion

Squirt weighs in at less than 50 grams. It is a completely autonomous mobile robot with onboard power, actuation, sensing, and state-of-the-art artificial intelligence control program. With this robot we have achieved three orders of magnitude reduction in mass of such robots in the 1988 calender year. We would like to believe we have the in-house capabilities to continue a similar rate of progress in the areas of computation and sensing in 1989, but we need help from the micro-machining community to complement this work with similar gains in power and actuation technologies.

Although Squirt was a stepping-stone exercise towards gnat robots, the unexpected spinoffs have been rather pleasant. With the realization that cost savings accrue at the small, yet macroscopic dimensions of Squirt, we were able to multiply the number of robots that exist in the AI Lab and deliver valuable educational lessons by creating a tool which let the general lab audience experience the problems associated with real robots. Availability of that technology produced an environment that fostered experimentation and creativity, with lots of fun incorporated along the way.

References

[Ayache and Faugeras (1988)] "Maintaining Representations of the Environment of a Mobile Robot", Nicholas Ayache and Olivier Faugeras, *Proceedings of the Fourth International Symposium on Robotics Research*, Bolles and Roth eds., MIT Press, pages 337-350.

[Bart et al (1988)] "Design Considerations for Microfabricated Electric Actuators", S.F. Bart, T. A. Lober, R. T, Howe, J. H. Lang and M. F. Schlecht, Sensors and Actuators 14(3):269-292, July.

[Brooks (1986)] "A Robust Layered Control System for a Mobile Robot", Rodney A. Brooks, *IEEE Journal of Robotics and Automation*, RA-2, April, 14-23.

[Brooks (1987a)] "A Hardware Retargetable Distributed Layered Architecture for Mobile Robot Control", Rodney A. Brooks, *Proceedings IEEE Conference on Robotics and Automation*, Raleigh, NC, 106-110.

[Brooks (1987b)] "Micro-Brains for Micro-Brawn; Autonomous Microbots", Rodney A. Brooks, *IEEE Micro Robots and Teleoperators Workshop*, Hyannis, MA, November.

[Brooks (1989)] "A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network", Rodney A. Brooks. *Neural Computation* 1:2, MIT Press.

[Brooks and Connell (1986)] "Asynchronous Distributed Control System for a Mobile Robot", Rodney A. Brooks and Jonathan H. Connell, *SPIE Vol. 727 Mobile Robots*, 77-84.

[Chatila (1986)] "Mobile Robot Navigation: Space Modeling and Decisional Processes", Raja Chatila, *Proceedings of the Third International Symposium on Robotics Research*, Faugeras and Giralt eds., MIT Press, 373-378.

[Christian and Vaaler (1989)] Personal communication. Andrew Christian and Eric Vaaler. MIT AI Lab.

[Connell (1987)] "Creature Building with the Subsumption Architecture", Jonathan H. Connell, *IJCAI-87*, Milan, Italy, 1124-1126.

[Connell (1988)] "A Behavior-Based Arm Controller", Jonathan H. Connell, *SPIE Mobile Robot Conference*, Cambridge, MA, November.

[Flynn (1987)] "Gnat Robots (And How They Will Change Robotics)", Anita M. Flynn,, *IEEE Micro Robots and Teleoperators Workshop*, Hyannis, MA, November.

[Flynn (1988)] "Gnat Robots: A Low-Intelligence, Low-Cost Approach", Anita M. Flynn, *IEEE Solid-State Sensor and Actuator Workshop*, Hilton Head, SC, June.

[Flynn et al (1988)] "The Olympic Robot Building Manual", Edited by Anita Flynn, with contributions from Colin Angle, Rodney Brooks, Jon Connell, Anita Flynn, Ian Horswill, Maja Mataric, Henry Minsky, Peter Ning, Paul Viola and William Wells. *MIT AI Lab Manual*, December.

[Flynn (1989)] "The Official Photograph Album of the 1989 Robot Olympics", Edited by Anita Flynn. *MIT AI Lab Manual*, March.

[Fujita and Omodaka (1987)] "Electrostatic Actuators for Micromechatronics", Hiroyuki Fujita and Akito Omodaka, *IEEE Micro Robots and Teleoperators Workshop*, Hyannis, MA, November.

[Horswill and Brooks (1988)] "Situated Vision in a Dynamic World: Chasing Objects", Ian D. Horswill and Rodney A. Brooks, *AAAI-88*, St Paul, MN, August.

[Moravec (1984)] "Locomotion, Vision and Intelligence", Hans P. Moravec, *Proceedings of the First International Symposium on Robotics Research*, Brady and Paul eds, MIT Press, 215-224.

[Muller (1987)] "From ICs to Microstructures: Materials and Technologies", Richard S. Muller, *IEEE Micro Robots and Teleoperators Workshop*, Hyannis, MA, November.

[Petersen (1985)] "Silicon Sensor Technologies", Kurt E. Petersen. In *Technical Digest, IEEE International Electron Devices Meeting*, Washington, D.C., December, 2-7.

[Silver (1989)] "Ross Perot On Winning Organizations", Sheryl Silver. The Washington Post, May 7.

[Thorpe, Hebert, Kanade and Shafer (1988)] "Vision and Navigation for the Carnegie-

Mellon Navlab", C. Thorpe, M. H. Hebert, T. Kanade and S. A. Shafer, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 3, May.

[Trimmer and Gabriel (1987)] "Design Considerations for a Practical Electrostatic Micro Motor", W. S. N. Trimmer and K. J. Gabriel, *Sensors and Actuators*, 11(2):189-206.

[Turk, Morgenthaler, Gremban and Marra (1988)] "VITS - A Vision System for Autonomous Land Vehicle Navigation", M. A. Turk, D. G. Morgenthaler, K. D. Gremban and M. Marra, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 3, May.

Appendix

As Squirt is one of the simplest robots produced by the Mobile Robot Group, we include his subsumption code here for the curious. Attached is both the control program specified in terms of augmented finite state machines and also the operating system, written in assembly language, which the subsumption compiler references in order to implement the scheduler and facilitate calls to sensor and actuator hardware.

```
;;; make hide, spiral and drive-init work together right.

;;; squirt's brain program

;;; ***************************************************************************

;;; motor control
;;;
;;; first layer simply provides forward, turn, and halt capabilities
;;; second layer checks for motor stall and tries unwedging manouevers
;;;
;;; note that set-motor returns either:  -1 = backward, 0 = stopped, 1 = forward

(defafsm motor (control 1)
  :registers (halt go heading headc)
  :outputs (status)
  :states ((nil (event-dispatch heading turn
                                go forward))
           (forward (output status (set-motor ':forward)) waithalt)
           (waithalt (event-dispatch halt dohalt))
           (dohalt (output status (set-motor ':off)) nil)
           (turn (output status (set-motor ':backward)) initloop)
           (initloop (setf headc heading) loop)
           (wait (setf headc (- headc 1)) test)
           (test (cd (= 0 headc) dohalt loop))
           (loop (event-dispatch (delay 0.10) wait
                                 halt dohalt))))

#|
(defafsm stall (control 2)
  :registers (status)
  :outputs ()
  :states ((nil (event-dispatch (delay 0.25) checkstall))
           (checkstall (
|#



;;; ***************************************************************************

;;; spiralling and hiding behavior---triggered by being in light

;;; spiral relies on turns taking precedence over forward in the motor controller

(defafsm measure-light (light 1)
  :outputs (lightlevel)
  :states ((nil (event-dispatch (delay 0.1) test))
           (test (cd (> (read-photodiode) 6) dark light))
           (dark (output lightlevel 0) nil)
           (light (output lightlevel 1) nil)))

(defafsm hidep (light 1)
  :registers (llevel saved count)
  :outputs (dark light)
  :states ((nil (event-dispatch llevel save))
           (save (setf saved llevel) scount)
           (scount (setf count 3) samep)
           (samep (cd (= saved llevel) dec nil))
           (dec (setf count (- count 1)) donep)
           (donep (cd (= count 0) test ok))
           (ok (event-dispatch llevel samep))
           (test (cd (= 0 saved) drk lght))
           (drk (output dark 1) nil)
```

```
          (lght (output light 1) nil)))

(defafsm spiral (light 1)
  :registers (init pathlength pathc)
  :outputs (go turn halt)
  :states ((nil (event-dispatch init start))
           (start (setf pathlength 2) roll)                          ;****
           (roll (setf pathc pathlength) goroll)
           (goroll (output go 1) wait)
           (wait (event-dispatch (delay 1.0) count))                 ;****
           (count (setf pathc (- pathc 1)) test)
           (test (cd (= pathc 0) turnit goroll))
           (turnit (output halt 1) doturn)
           (doturn (ed (delay 0.05) rturn))
           (rturn (output turn 35) incsoon)
           (incsoon (ed (delay 4.0) inc))
           (inc (setf pathlength (+ pathlength 1)) roll)))

(defwire (light 1) (measure-light lightlevel) (hidep llevel))

(defwire (light 1) (hidep dark) (motor halt) ((reset spiral)))

(defwire (light 1) (hidep light) (spiral init))

(defwire (light 1) (spiral go) (motor go))

(defwire (light 1) (spiral turn) (motor heading))

(defwire (light 1) (spiral halt) (motor halt))

;;; ****************************************************************************

;;; drive forward behavior -- assumes the existence of (control 1) and (spiral 1).

;;; a heading is a number which is a measure of how much time to
;;; spin backwards
;;;
;;; drive-init waits until its received a heading (from sound localization)
;;; and waits an additional 60 seconds.  if all is quiet during that time
;;; it tells the motor controller to reorient the robot and tells goforward
;;; to go forward.  goforward repeatedly tells the motor controller to go
;;; forward for twenty seconds.  while the motor controller is handling the
;;; previous command to turn it ignores these messages.  when all this motion
;;; is started the checkidle machine is initiated and it waits for three
;;; seconds of total idleness of the robot before it lets drive-init start
;;; waiting for new sounds.

(defafsm hear (drive 1)
  :registers (temp)
  :outputs (sound)
  :states ((nil (event-dispatch (delay 0.5) check))
           (check (cd (= 0 (examine 'heardsound)) nil fire))
           (fire (setf temp (examine 'direction)) test1)
           (test1 (cd (> temp 25) goleft test2))
           (test2 (cd (< temp -25) goright goforward))
           (goleft (output sound 55) zeroflag)
           (goright (output sound 20) zeroflag)
           (goforward (output sound 0) zeroflag)
           (zeroflag (progn (deposit 'heardsound 0)
                            (deposit 'thresh 10)
                            (deposit 'allowedvar 64))
                     nil)))
```

```
(defafsm monitor (drive 2)
  :registers (dir)
  :states ((nil (event-dispatch dir pr0))
           (pr0 (progn (write-crlf)) pr1)
           (pr1 (ed (delay 0.05) pr2))
           (pr2 (progn (write-hex dir)) pr3)
           (pr3 (ed (delay 0.05) nil))))

(defafsm drive-init (drive 1)
  :registers (heading idle)
  :monostables ((seenhead 6.0))                                    ;****
  :outputs (start)
  :states ((nil (event-dispatch heading gothead
                                (and seenhead (delay 5.0)) turn))      ;****
           (gothead (trigger seenhead) nil)
           (turn (output start heading) wait)
           (wait (event-dispatch idle nil))))

(defafsm goforward (drive 1)
  :registers (init)
  :monostables ((innerdrive 15.0))                                 ;****
  :outputs (go)
  :states ((nil (event-dispatch init gostart))
           (gostart (trigger innerdrive) pulse)
           (pulse (event-dispatch (delay 0.033) out
                                  (not innerdrive) nil))
           (out (output go 1) pulse)))

(defafsm checkidle (drive 1)
  :registers  (mstatus count)
  :outputs (isidle)
  :states ((nil (event-dispatch (delay 0.1) init))
           (init (setf count 30) down)                             ;****
           (dec (setf count (- count 1)) test)
           (test (cd (= count 0) reset loop))
           (loop (event-dispatch (delay 0.1) down))
           (down (cd (= 0 mstatus) dec nil))
           (reset (output isidle 1) nil)))

(defwire (drive 1) (hear sound) (drive-init heading))

(defwire (drive 2) (hear sound) (monitor dir))

(defwire (drive 1) (drive-init start)
  (motor heading)
  (goforward init)
  ((reset checkidle)))

(defwire (drive 1) (checkidle isidle) (drive-init idle))

(defwire (drive 1) (motor status) (checkidle mstatus))

(defwire (drive 1) (goforward go)
  (motor go)
  ((inhibit (measure-light lightlevel) 0.066)))
```

```
;;; Operating system for the 1 cubic inch bug

(set-current-machine 'm68hc11)

;;; interfaces to the operating system

(defcode-subst opsys-decls ()
  (comment "register assignments -- index with #x1000")
  (=c porta #x00)
  (=c portb #x04)
  (=c porte #x0a)
  (=c tcnt #x0e)
  (=c mic2-capture #x12)
  (=c mic1-capture #x14)
  (=c tctl2 #x21)
  (=c tmsk1 #x22)
  (=c tflg1 #x23)
  (=c tmsk2 #x24)
  (=c tflg2 #x25)
  (=c pactl #x26)
  (=c spcr #x28)
  (=c baud #x2b)
  (=c sccr2 #x2d)
  (=c scsr #x2e)
  (=c scdat #x2f)
  (=c adctl #x30)
  (=c option #x39)
  (=c tachometer #x31 (comment "will get a/d of e0"))
  (=c photodiode #x32 (comment "will get a/d of e1"))
  (=c xbase #x1000 (comment "base of internal regs"))
  (comment "masks for registers")
  (=c rti-enable #x40 (comment "RTI enable bit for tmsk2"))
  (=c rti-rate #x03 (comment "32.77ms at 8MHz for pactl"))
  (=c portd-wired-or #x20)
  (=c 1200baud #x33)
  (=c 9600baud #x30)
  (=c trena #x0c)
  (=c tietrena #x8c)
  (=c rdrf #x20)
  (=c tdre #x80)
  (=c stop #x0000 (comment "stop the motors"))
  (=c forward #x0101 (comment "drive forward"))
  (=c backward #xff04 (comment "drive backward"))
  (=c a-to-d-scanner #x30 (comment "pick up e0 thru e1"))
  (=c adpu #x80 (comment "turn on a/d convertor"))
  (=c tctl2-mask #x05 (comment "rising edges on both mics"))
  (=c mic1-bit #x01 (comment "flag bit for mic 1"))
  (=c mic2-bit #x02 (comment "flag bit for mic 2"))
  (=c micboth-bits #x03 (comment "flag bits for both mics"))
  (comment)
  (comment "memory locations")
  (=v d0 #xc8 (comment "pseudo register 0"))
  (=v d1 #xc9 (comment "pseudo register 1"))
  (=v d2 #xca (comment "pseudo register 2"))
  (=v d3 #xcb (comment "pseudo register 3"))
  (=v2 clock #xcc (comment "two byte system clock"))
  (=v2 hclock #xce (comment "clock overflow protector"))
  (=v heardsound #xd0 (comment "flag that have heard a sound"))
  (=v outhex1 #xd1 (comment "first byte hex out"))
  (=v outhex2 #xd2 (comment "second byte hex out"))
  (=v outhexnull #xd3 (comment "always zero"))
  (=v2 outstring #xd4 (comment "out string pointer"))
  (=v2 ts1 #x50)
```

```
  (=v2 ts2 #x52)
  (=v2 ind #x54)
  (=v2 xn #x56)
  (=v2 mean #x58)
  (=v2 ss #x5a)
  (=v2 dtemp1 #x5c)
  (=v grpideal #x5e)
  (=v direction #x5f)
  (=v grp #x60)
  (=v badcount #x61)
  (=v thresh #x62)
  (=v allowedvar #x63)
  (=v2 xarr #x64)
  (=v2 sarr #x74)
  (comment)
  (comment)
  (comment "constants")
  (=c stack #xFF (comment "stack pointer"))
  (comment "assume an 8Mhz clock")
  (=c clock-reset 8192 (comment "value to reset clock to"))
  (=c clock-overflow 16384 (comment "number of ticks until clock `overflow'"))
 )



(defmacro always (form)
  form)
(defmacro whenio (form)
  `(if *include-io-code*
     ,form
     '()))
(defmacro whennotio (form)
  `(if *include-io-code*
     '()
     ,form))

(defcode-macro opsys-init ()
 (append
  (always '(
  (= #xf800)
  init-opsys
  (comment)
  (lds (! stack) (comment "initialize stack pointer"))
  (comment "first initialize serial port")
  (ldx (! xbase) (comment "leave indx here forever"))
  (bset (& spcr) portd-wired-or (comment "set up serial port bit"))
  (ldaa (! 9600baud) (comment "set serial port to 9600 baud"))
  (staa (& baud))
  (ldaa (! trena) (comment "enable transmit and receive"))
  (staa (& sccr2))
  (ldaa (! rti-rate) (comment "real time interrupt rate"))
  (staa (& pactl))
  (ldaa (! rti-enable) (comment "enable real time interrupt"))
  (staa (& tmsk2))
  (ldd (! clock-reset) (comment "initialize clock"))
  (std clock (comment "  -- illegal to be 0"))
  (ldd (! clock-overflow) (comment "initialize clock overflow"))
  (std hclock)
  (ldaa (! tctl2-mask) (comment "set mic edge conditions"))
  (staa (& tctl2) (comment "  in control register"))
  (ldaa (! micboth-bits) (comment "set interrupts to occur"))
  (staa (& tmskl) (comment "  on both mics"))
  (ldaa (! adpu) (comment "turn on the a/d"))
```

```
   (staa (& option) (comment "  system"))
   (ldaa (! a-to-d-scanner) (comment "switch on scanning"))
   (staa (& adctl) (comment "  of 4 a/d lines"))
   (comment "initiliaze microphone datastructures")
   (jsr flushstuff)
   (ldab (! 10))
   (stab thresh)
   (ldab (! 64))
   (stab allowedvar)
   (clr heardsound)
  ))

  (whenio '(
   (clr outhexnull)
   (clr outstring)
   (clr (computed (+ outstring 1)))))

  (always '(
   (cli (comment "enable interrupts"))))
   (whenio '(
   (clr outhex2 (comment "  and warm up the serial line"))
   (ldy (! outhex2))
   (jsr write-string)))))

(defcode-macro opsys-epilog (num-timestamps)
 (append
  (always `(
  (comment "initialize microphones")
flushstuff
  (clra)
  (clrb)
  (std mean)
  (std ss)
  (std xarr)
  (std sarr)
  (std (computed (+ xarr 2)))
  (std (computed (+ sarr 2)))
  (std (computed (+ xarr 4)))
  (std (computed (+ sarr 4)))
  (std (computed (+ xarr 6)))
  (std (computed (+ sarr 6)))
  (std (computed (+ xarr 8)))
  (std (computed (+ sarr 8)))
  (std (computed (+ xarr 10)))
  (std (computed (+ sarr 10)))
  (std (computed (+ xarr 12)))
  (std (computed (+ sarr 12)))
  (std (computed (+ xarr 14)))
  (std (computed (+ sarr 14)))
  (staa grp)
  (staa ind)
  (ldab (! 8))
  (stab badcount)
  (ldab (! 14))
  (stab (computed (+ ind 1)))
  (rts)
  (comment "interrupt handlers")
  (comment)
  (comment "heartbeat timer")
heartbeat
  (ldaa (! rti-enable) (comment "clear the interrupt"))
  (staa (& tflg2))
  (ldd clock (comment "time to beat the heart"))
```

```
   (addd (! 1))
   (std clock)
   (ldd hclock)
   (subd (! 1))
   (beq heart-overflow)
heart-continue
   (std hclock)
   (rti)
   (comment "handle clock overflow")
heart-overflow
   (ldd (! clock-reset) (comment "initialize clock"))
   (std clock)
   (ldx (! 0) (comment "point to first timestamp"))
   (ldd (! ,num-timestamps) (comment "initialize timestamp count"))
heart-loop
   (subd (! 1))
   (blt heart-done (comment "branch when all timestamps updated"))
   (std hclock)
   (ldd (& 0) (comment "pick up a time stamp"))
   (subd (! clock-overflow) (comment "  adjust it"))
   (bge heart-ok (comment "  when it is out of date already"))
   (clra (comment "    just zero it out"))
   (clrb)
heart-ok
   (std (& 0) (comment "  and store result"))
   (inx)
   (inx (comment "increment timestamp pointer"))
   (ldd hclock (comment "  and check for more to do"))
   (bra heart-loop)
heart-done
   (ldd (! clock-overflow) (comment "initialize clock overflow"))
   (bra heart-continue (comment "  and complete interrupt handling"))
   (comment)
   (comment "update means etc for mics")
losesound
   (jmp realend)
update
   (ldd ts1)
   (subd ts2)
   (std xn)
   (cpd const200)
   (bgt losesound)
   (cpd constm200)
   (blt losesound)
   (lsrd (comment "upper byte is 0 or -1"))
   (lsrd)
   (lsrd)
   (lsrd)
   (stab (computed (+ xn 1)))
   (ldx ind)
   (ldd mean (comment "pick up current mean"))
   (subd (& xarr) (comment "subtract oldest member from mean"))
   (addd xn (comment "  add newest member"))
   (std mean (comment " and save the mean"))
   (ldd xn (comment "now remember newest older member"))
   (std (& xarr) (comment "  for later"))
   (comment "now compute scaled sum of squares")
   (bge updalpos (comment "branch if xn already positive"))
   (negb (comment "negate lower 8 bits"))
updalpos
   (comment "now in the range 0-200 roughly")
   (tba (comment "copy low 8 bits into acc a"))
   (mul (comment " and square it"))
```

```
   (asld (comment "multiply by 8"))
   (asld)
   (asld)
   (std dtempl (comment "save it"))
   (ldd ss (comment "get sum of squares"))
   (subd (& sarr) (comment "subtract oldest"))
   (addd dtempl (comment " add newest"))
   (std ss (comment "  and save sum squares"))
   (ldd dtempl (comment "pick up 4*X**2"))
   (std (& sarr) (comment "  and save for later"))
   (ldx (! xbase) (comment "restore index register x"))
   (ldaa (computed (+ ind 1)) (comment "pick up index"))
   (suba (! 2) (comment " decrement"))
   (staa (computed (+ ind 1)) (comment " and store"))
   (bge updok (comment "  and skip if non negative"))
   (ldab (! 14) (comment " else restore it to 14"))
   (stab (computed (+ ind 1)))
   (ldd mean (comment "compute the variance for this group"))
   (tsta (comment "  first take absolute value of"))
   (bge okok (comment "  small number in double byte"))
   (negb)
okok
   (tba (comment "copy into both accs"))
   (mul (comment "  and square"))
   (subd ss (comment "substract sum squares"))
   (coma (comment "and negate"))
   (comb)
   (addd (! 1))
   (tsta (comment "now make sure this is small"))
   (bne updok (comment "if not then discard group"))
   (cmpb allowedvar (comment "compare to desired"))
   (bge updok (comment "  and quit if too big"))
   (ldaa grp (comment "pick up group index"))
   (bne grpnorm (comment "  and branch if non-zero"))
   (ldd mean (comment "save the mean"))
   (stab grpideal)
   (inc grp)
   ;(bra temphack)
   (bra updok)
grpnorm
   (ldd mean (comment "pick up the mean"))
   (subb grpideal (comment "  and subtract from ideal"))
   (bge grpok (comment "  and take abs value"))
   (negb)
grpok
   (cmpb thresh (comment "  now see if it's ok"))
   (bgt badthresh (comment "if not check time up"))
   (ldaa grp (comment "otherwise increment count"))
   (inca)
   (staa grp)
   (cmpa (! 4) (comment "see if done"))
   (bne updok)
temphack
   (ldaa (! 1))
   (staa heardsound (comment "flag sound arrival"))
   (ldaa grpideal (comment "  and save in safe place"))
   (staa direction)
   (bra flushall (comment "then flush everything"))
badthresh
   (dec badcount)
   (bne updok)
flushall
   (jsr flushstuff)
```

```
updok
realend
   (clra)
   (clrb)
   (rts (comment "and return"))
const200
   (!16 #.(round (* 1.2 2000000) 12000) (comment "time to travel 1.2 inches"))
constm200
   (!8 #xff)
   (!8 #.(- 256 (round (* 1.2 2000000) 12000)))

   (def-ass-subst mic-interrupt (startlabel mcts otherts miccapture endlabel mic-bit)
     startlabel
        (ldd & miccapture)
        (std mcts)
        (ldd otherts)
        (beq endlabel)
        (jsr update)
        (std otherts)
     endlabel
        (ldaa ! mic-bit)
        (staa & tflg1)
        (rti))
   (comment "microphone 1 interrupt")
   (mic-interrupt mic1-interrupt ts1 ts2 mic1-capture mic1end mic1-bit)
   (comment)
   (comment "microphone 2 interrupt")
   (mic-interrupt mic2-interrupt ts2 ts1 mic2-capture mic2end mic2-bit)
  ))

   (always '(
   (comment)
   (comment "motor controller")
%motor-off
   (ldd (! stop))
   (stab (& portb))
   (rts)
%motor-forward
   (ldd (! forward))
   (stab (& portb))
   (rts)
%motor-backward
   (ldd (! backward))
   (stab (& portb))
   (rts)))

   (whenio '(
   (comment "serial line reader")
   (comment "  returns 0 if no char avail")
   (comment "  otherwise the character")
inchar
   (clra)
   (brclr (& scsr) rdrf donechar)
   (ldaa (& scdat))
donechar
   (rts)

   (comment "serial line writer")
   (comment "  simply gives up if the line is busy")
outchar
   (ldy outstring)
   (bne write-string-done)
   (staa outhex2)
```

```
   (ldy (! outhex2))
   (bra write-string)

   (comment "hex writer -- uses write string")
outhex
   (tab)
   (iterate ((i 4)) (lsra))
   (adda (! #.(char-code #\0)))
   (cmpa (! #.(+ 9 (char-code #\0))))
   (ble write-hex-done1)
   (adda (! #.(- (char-code #\A) (1+ (char-code #\9)))))
write-hex-done1
   (staa outhex1)
   (andb (! #x0F))
   (addb (! #.(char-code #\0)))
   (cmpb (! #.(+ 9 (char-code #\0))))
   (ble write-hex-done2)
   (addb (! #.(- (char-code #\A) (1+ (char-code #\9)))))
write-hex-done2
   (stab outhex2)
   (ldy (! outhex1))
   (bra write-string)

outstring-entry
   (puly)
   (pshy)
   (iny)
   (iny)
write-string
   (ldd outstring (comment "if busy"))
   (bne write-string-done (comment "  then exit"))
   (ldaa (&y 0))
   (staa (& scdat))
   (iny (comment "increment string pointer"))
   (sty outstring)
   (ldaa (! tietrena) (comment "enable interrupts"))
   (staa (& sccr2))
   (cli)
write-string-done
   (rts)

   (comment "serial interrupt handler")
serial-interrupt
   (ldab (& scsr))
   (ldy outstring)
   (ldaa (&y 0) (comment "pick up next character"))
   (beq serial-done (comment "  done if null"))
   (staa (& scdat) (comment "else write the character"))
   (iny)
   (sty outstring (comment "increment pointer"))
   (rti (comment "  and dismiss"))
serial-done
   (ldaa (! trena) (comment "disenable interrupts"))
   (staa (& sccr2))
   (clr outstring)
   (clr (computed (+ outstring 1)))
   (rti)
  ))

  (always '(
  (comment "random traps")
  (comment)))
```

```
  (whennotio '(serial-interrupt outchar outhex outstring-entry))
  (always '(
bad-instruction
bad-interrupt
  (ldaa (! #b11111111))
  (staa (& portb))
  (jmp bad-instruction)

  (comment)
  (comment "hardware dispatch vectors")
  (comment)
  (= #xFFD6)
  (!16 serial-interrupt)
  (iterate ((i 9)) (!16 bad-interrupt))
  (!16 mic1-interrupt (comment "mic 1 detects sound"))
  (!16 mic2-interrupt (comment "mic 2 detects sound"))
  (!16 bad-interrupt)
  (!16 heartbeat (comment "real time interrupt"))
  (!16 bad-interrupt)
  (!16 bad-interrupt)
  (!16 bad-interrupt)
  (!16 bad-instruction)
  (!16 bad-interrupt)
  (!16 bad-interrupt)
  (comment)
  (= #xFFFE)
  (!16 init-opsys (comment "boot address"))))))

;;; primops

;;; these next two are obselete--used to debug the initial proto-proto-board

(defprim set-led
  :args ()
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code `((ldaa (! #b1111111))
          (staa (& portb))))

(defprim unset-led
  :args ()
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code `((clr (& portb))))

(defprim set-motor
  :args ((x literal))
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code (case x
          (:off `((jsr %motor-off)))
          (:forward `((jsr %motor-forward)))
          (:backward `((jsr %motor-backward)))
          (otherwise (error "Unknown set-motor command"))))

(defprim read-photodiode
  :args ()
  :extra ((val !accum-a))
  :result val
  :isrep value
```

```
  :code `((ldaa (&x photodiode))))

(defprim read-char
  :args ()
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code `((jsr inchar)))

(defprim write-char
  :args ((char !accum-a))
  :result char
  :isrep value
  :code `((jsr outchar)))

(defprim write-space
  :args ()
  :extra ((bl !accum-a))
  :result bl
  :isrep value
  :code `((ldaa (! #.(char-code #\space)))
          (jsr outchar)))

(defprim write-crlf
  :args ()
  :extra ((bl !accum-a))
  :result bl
  :isrep value
  :code `((clra)
          (jsr outchar)))

(defprim write-hex
  :args ((val !accum-a))
  :result val
  :isrep value
  :code `((jsr outhex)))

(defprim write-string
  :args ((str literal))
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code (let ((label (gentemp "BYPASS-STR")))
          `((jsr outstring-entry)
            (bra ,label)
            (!string ,str)
            (!8 0)
            ,label)))

(defprim compute-var
  :args ()
  :extra ((val !accum-a))
  :result val
  :isrep value
  :code `((jsr compvar)))
```