

**The Programmable LEGO Brick:
Ubiquitous Computing for Kids**

by

James Randal Sargent

B.S. Computer Science and Engineering
Massachusetts Institute of Technology
Cambridge, MA
1989

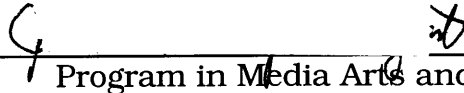
Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the Degree of

Master of Science in Media Arts and Sciences

at the
Massachusetts Institute of Technology
February 1995

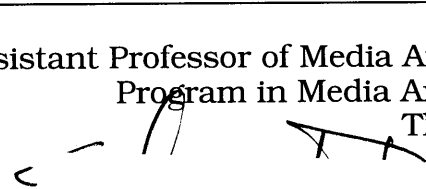
© 1995 Massachusetts Institute of Technology
All rights reserved

Signature of Author




Program in Media Arts and Sciences
January 20, 1995

Certified by



Mitchel Resnick
Assistant Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by



Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Rotch

MAR 28 1995

The Programmable LEGO Brick: Ubiquitous Computing for Kids

by

James Randal Sargent

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on January 20, 1995
in partial fulfillment of the requirements for the Degree of

Master of Science in Media Arts and Sciences

at the
Massachusetts Institute of Technology

Abstract

The Programmable LEGO Brick is a tiny, portable computer for kids which is capable of interfacing to the physical world in a large variety of ways. It is designed to support rich learning activities, motivated by the constructionist view of learning. The programmable brick breaks new ground for programming environments for kids: it connects programming to the "real world" in a much broader way than have previous systems, due to its portability and large number of input/output modalities. This thesis discusses goals for the programmable brick project, the design decisions made for the programmable brick, experiences with people using the brick, and the technical implementation of the brick.

Thesis Supervisor: Mitchel Resnick
Title: Assistant Professor of Media Arts and Sciences

This work was supported in part by the LEGO company and the National Science Foundation.

The Programmable LEGO Brick: Ubiquitous Computing for Kids

by
James Randal Sargent

The following people served as readers for this thesis:

// .

Reader _____
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering
MIT Department of Electrical Engineering and Computer Science

Reader _____
Brian Silverman
Director of Research
Logo Computer Systems, Incorporated

Table of Contents

| | |
|---|----|
| Acknowledgments | 9 |
| 1. Introduction | 11 |
| 2. Multiple | 17 |
| 3. Background | 29 |
| 4. Hardware Design | 33 |
| 5. Software Design | 45 |
| 6. Experiences | 63 |
| 7. More Activities | 79 |
| 8. Conclusions and Future Directions | 85 |
| Bibliography | 87 |
| Appendix A. 20 Things To Do with a Programmable Brick | 89 |

Acknowledgments

First and foremost, I wish to acknowledge and thank all the people who worked hard to make the programmable brick a reality. MIT undergraduates Fei Hai Chua and Elaine Yang helped with sensor design and manufacture. Owen Johnson, Chris Gatta, and Fei Hai Chua spent grueling sessions with me manually soldering the surface-mount parts for the first few programmable brick prototypes. MIT undergraduate Victor Lim worked on library routines to support the brick's infrared and sound capabilities. Then high school student Andrew Blumberg (now Harvard undergraduate) worked long hours writing the compiler for the brick Logo programming language. All of these people worked extremely hard to make the brick, and without them, the brick would not exist today.

Many people helped with ideas for the design of the programmable brick. Brian Silverman, Anne Wright, Fred Martin, Seymour Papert, and particularly Mitchel Resnick gave lots of ideas and feedback during various stages of the design.

Much of the technology behind the programmable brick comes from previous hardware and software for the MIT LEGO Robot Design Contest, created by myself and Fred Martin.

The LEGO company provided much support for this project, not only financially and through donations of materials, but also through their generous offer to mold the plastic case for the brick to specification.

In addition to his design ideas, Mitchel Resnick helped keep me on track to produce the brick and this thesis. Without such a gentle and supportive advisor I think I would have been tempted to give up on this project a few times during its development. He also spent lots of times reading and re-reading revisions of this document, and I believe his feedback helped me tremendously.

Lastly, I want to acknowledge the people who gave me emotional support during the course of this project. I have always had the unfailing encouragement and support of my parents, Jim and Betty Sargent. They always seem to know the right things to say, and right questions to ask. They also somehow always seemed to know when I should have been working on my thesis but wasn't. Anne Wright, who I hope to marry as soon as she's finished with her own thesis, has likewise given me much support. She provided lots of help with the initial designs of the brick, and she also gave lots of helpful feedback with the various revisions of this document. I am also grateful to her for her solution of removing all the games from my Macintosh, taking it off the network, and tying me to a chair in front of it in order to encourage me to get started writing this document.

Chapter 1: Introduction

The Programmable LEGO Brick is a tiny, portable computer, designed to be used primarily by kids. In contrast to most other portable computers, the programmable brick is capable of interfacing to the physical world in a large variety of ways, using various sensors and actuators. The brick is designed to support rich learning activities, motivated by the constructionist view of learning.

The programmable brick breaks new ground for programming environments for kids: it connects programming to the "real world" in a much broader way than have previous systems, due to its portability and large number of input/output modalities. One goal of the programmable brick project is to make available to kids new constructionist activities, which may let kids explore new ideas and new ways of thinking.

This thesis discusses goals for the programmable brick project, the design decisions made for the programmable brick, and experiences with people using the brick.

1.1 What is a Programmable Brick?

As seen in the figure 1, the Programmable LEGO Brick is a computer small enough to fit into a pocket, and yet it has more I/O connections than a Macintosh A/V computer. I/O ports can be found on five of its six sides (the sixth side, bottom, is used to snap to LEGO).

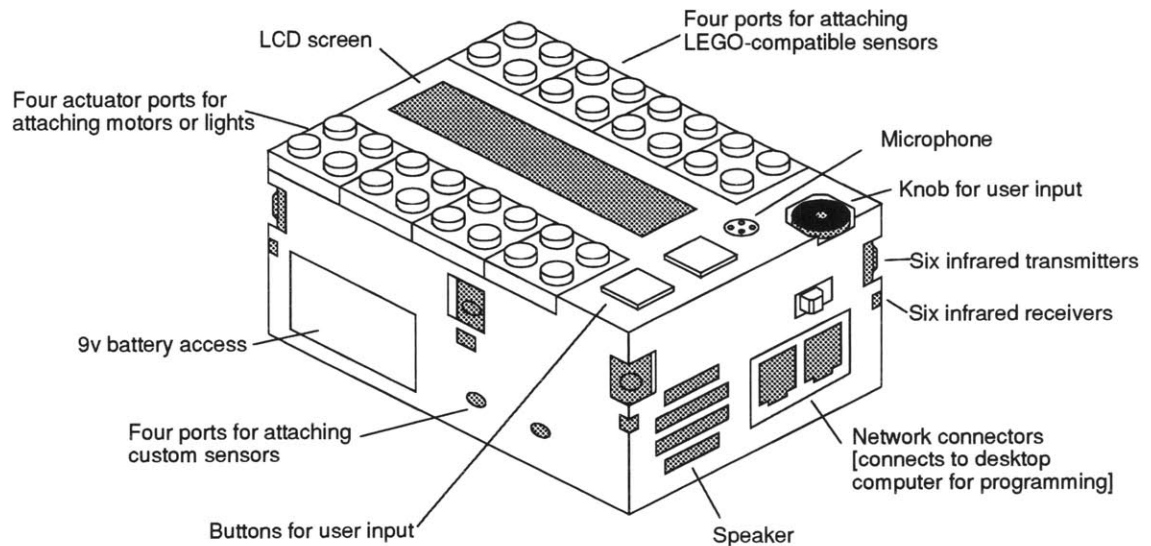


Figure 1.1: The programmable LEGO brick

The programmable brick has a multitude of I/O modalities: built-in speaker, microphone, infrared communications, a small LCD screen, buttons and a knob, and twelve ports for add-on I/O devices such as motors, and a wide variety of sensors. Users can download programs to the brick from a desktop computer, and the programs then run autonomously on the brick.

1.2 Why a Programmable Brick?

1.2.1 LEGO/Logo: The Next Generation

The programmable brick can be seen as a logical next step from the successful LEGO/Logo system developed together by the LEGO company and the MIT Media Laboratory. But a goal of the project is that this step in features would bring a leap in the range of computer-control activities and, more importantly, change the way children think about computation in general.

Before LEGO/Logo, computer programming for kids was generally limited to affecting things displayed by a computer screen. Computation could be applied only to things that could be represented *inside* the computer (and isolated from things *outside* the computer).

LEGO/Logo made an important step toward connecting programming for kids to the real world [Resnick, Ocko, Papert 1988]. Using LEGO/Logo, kids could connect a desktop computer to a motorized LEGO device, such as a conveyor belt or merry-go-round, and then write a program to control it. No longer was computation's effect limited to the computer screen.

But LEGO/Logo connects to the real world in a somewhat limited way. One problem is that the LEGO creations being controlled must be tied by wires to a desktop computer. Most of the real world simply doesn't sit on the desktop next to the desktop computer. Another problem is that LEGO/Logo has limited I/O modalities (can sense only pressure and light, and actuate with movement or light, and no modalities designed for communication). Lots of what goes on in the real world simply can't be connected to LEGO/Logo because the I/O modalities are not broad enough. Although LEGO/Logo makes the first step toward connecting computation to things that are real, it still suffers from a similar limitation to the pre-LEGO/Logo system: computation can still only be applied to a limited set of things.

The programmable LEGO brick is specifically designed to address the limitations of LEGO/Logo. The brick, through its portability, can be brought to, and even be left in, the real world. And through its larger variety of I/O modalities, it can be interfaced to a relatively large percentage of the things it can be brought to. Kids can program a brick to adjust a thermostat based on time of day and amount of activity in a room, or put a brick in a forest to try and attract animals by making animal noises and then record or even react to noises heard in return.

The programmable brick not only breaks us free of the desktop computer, it breaks us free of the desktop computer mentality: that computation created by kids be isolated to interaction with things inside the desktop computer, or things outside the computer but on the desktop itself. The programmable brick brings to kids the revolution of ubiquitous computing: computation connected to and spread throughout the environment. But it brings the revolution with a twist: kids are the *designers*, not just users, in this new realm.

1.2.2 Why design and build?

The major underlying motivation for the programmable brick is to get kids to design and build, a goal of the educational approach known as constructionism [Papert 80].

Constructionism starts with Piaget's theory of constructivism, which states, simply, that learners actively construct their own knowledge. A learner isn't just a tabula rasa, or blank slate, upon which knowledge can be written. Learners actively analyze what they see, then either assimilate their observations into their earlier mental models, or are sometimes forced to change their mental models to accommodate new observations which were inconsistent with their earlier ideas.

Papert takes constructivism a step further, in an approach he calls constructionism. Papert states that a particularly good way for a learner to construct knowledge is to construct things. Designing and building things is a very rich way to interact with materials and ideas. The

learner must think up ideas about how to build something. And it's usually the case that the learner finds his ideas, when implemented, don't work exactly as he had planned. This gives the learner an opportunity to add to or change his model of what's going on to account for the discrepancy, and try to think up new ideas about how to make it work. This back-and-forth between the learner and the artifacts he is trying to create keeps the learner engaged in the cycle of observation, then assimilation or accommodation, described by Piaget.

But even without this abstract, theoretical model of what's going on inside the learner's mind, it's easy to see reasons that designing and building can be good activities for learning. This list of reasons is taken from "Towards a Practice of 'Constructional Design'", by Mitchel Resnick [Resnick, in press]:

- Design activities engage students as *active participants*, giving them a greater sense of control (and responsibility) over the learning process, in contrast to traditional school activities in which teachers aim to “transmit” new information to the students.
- Design activities encourage *reflection and discussion*, since the artifacts that students design can serve as “props” for students to reflect on and talk about.
- Design activities encourage a *pluralistic epistemology*, avoiding the right/wrong dichotomy prevalent in most school math and science activities, suggesting instead that multiple strategies and solutions are possible (Turkle & Papert, 1990).
- Design activities are often *interdisciplinary*, breaking down the barriers that typically separate subject domains in school.
- Design activities provide a sense of *authenticity*, suggesting stronger connections to “real-world” activities (since most real-world activities involve design-oriented strategies, not the rule-driven, logic-oriented analyses that underlie many school activities).
- Design activities can facilitate *personal connections*, since students often develop a special sense of ownership (and caring) for the artifacts that they design.
- Design activities promote a *sense of audience*, encouraging students to consider how other people will use and react to the artifacts they create.

1.3 The rest of this document

The rest of this thesis goes into detail about the ideas behind the brick, the design of the brick, and experiences with the brick.

Chapter 2 is the "Multiple" chapter: it explores a common thread important to many of the different motivations behind the design of the brick: the desire for multiple sensor modalities on a brick, multiple computational processes running on a brick, multiple bricks interacting, and multiple activities for bricks.

Chapter 3 explores connections between this project and previous constructionist work, as well as connections between this project and the emerging area of ubiquitous computing.

Chapters 4 and 5 discuss the software and hardware design of the brick, exploring design decisions and tradeoffs, as well as describing in detail the product of the decisions: the brick itself.

Chapter 6 discusses experiences with learners using bricks in various situations and environments.

Chapter 7 explores some untried learning-rich activities designed around the brick, suggesting some possible future directions for explorations with the programmable brick.

Chapter 8 concludes this thesis, recapping the successes and failures, and pointing to future directions suggested by this work.

Chapter 2: "Multiple"

For lack of a better name, this is the "Multiple" chapter: it explores a common thread important to many of the different motivations behind the design of the brick: the desire for multiple input/output modalities on a brick, multiple computational processes running on a brick, multiple bricks interacting, and multiple activities for bricks.

2.1 The Need for Multiple Activities for Bricks

Having the brick support many varied activities is important for many reasons. One important reason is that it allows a larger fraction of people to become interested in using the programmable brick. If some connection can be made between the programmable brick and a person's interests, it will make that person more likely to try the brick out.

One example of this was an experience at The Computer Museum in Boston, where this author was attempting to interest junior high students to try out using the brick to make programmable musical instruments. But the kids lost interest quickly, and were drawn to a LEGO train set at the other end of the room. The programmable brick could be interfaced to the train as well, and once it was, the kids became much more interested in programming.

But not only can the brick be used for a variety of different activities, these different activities can often be merged, or connected, through use of the brick. People who start to use the brick for familiar and comfortable activities may get drawn into activities that are new. A kid comfortable with building a LEGO car but unfamiliar with computers may get drawn into programming to control the car. The same kid might program the car to make a simple tone when it hits a wall, and then get drawn into how to design increasingly complex sound effects.

Another advantage is that different people, with different backgrounds, can link up and combine their talents to make new projects. For example, a person with musical experience might work with person with sensor expertise to make a musical instrument with a nice user-interface for playing.

2.2 The Need for Multiple Input/Output Modalities

The programmable brick attempts to connect computing to the "real world" in as broad and deep a way as possible. But in order to connect with the real world, the brick must be able to act on it and react to it. The more ways the brick can sense or act on the world, the more broadly it connects to the real world, and the wider the variety of activities that can be done with it.

In fact, the variety of activities that can be done with a programmable brick does not increase simply linearly with the number of things the brick can connect to. Many of the simple and interesting applications of the brick can be thought of as living at the intersections between things the brick can sense and things the brick can actuate. Of course, more complex applications would use more than just one input and output. Figure 2.1 shows a grid of sample activities brainstormed for the brick, organized by sensor and actuator modalities. If this simple representation were the correct model for defining activities, then the number of activities might grow with the square of the number of sensor and actuator modalities. In fact, since more complex applications will use several modalities, and each modality can in fact interface to many different things in the real world, the number of possible activities grows at a much larger rate.

| Output Modality: | Speaker | Motor |
|---------------------------------|---|--|
| Input Modality | | |
| Microphone | Tape recorder Fake watch dog <i>bark when a noise is heard</i> | Artificial LEGO creature scared of loud sounds |
| Tone detection | Birds in a jungle <i>multiple bricks hear and respond to each other.</i> | Mechanical musical accompaniment <i>brick listens for notes and accompanies different ones by hitting a glass bottle or xylophone, etc.</i> |
| Touch sensor | Programmable saxophone | |
| Remote control | Ventriloquist <i>signal brick to play back sampled speech at a distance</i> | Remote controlled car |
| Knob & button | Sound Effect player <i>different sounds depending on different settings of knob</i> | Adjustable motorized fan |
| Light sensor | Greet people coming in room <i>detect people walking through a doorway and react to them.</i> | Artificial roach <i>creature that is afraid of light.</i> |
| Infrared person detector | Alarm system Haunted house <i>try to scare people with spooky noises as they walk by in a dark room</i> | Artificial creature that follows people |
| Angle/rotation sensor | Programmable LEGO trombone <i>angle sensor can measure in and out movement of the slide of a home-brew LEGO trombone.</i> | |
| Temperature sensor | Temperature alarm <i>brick can act like a teakettle, or warn of freezing temperatures at pipes.</i> | Adjust a radiator knob to regulate temperature |
| Distance sensor | Computer Museum "height" exhibit <i>Boston Computer Museum has an exhibit that measures a person's height and speaks the measurement to the person.</i> | |
| Time | Cuckoo clock | Open blinds at 8 am to wake up room occupants |

Figure 2.1, part 1 of 3
Activities at intersections of input and output modalities

| | | |
|---------------------------------|---|---|
| Output Modality: | Light or light switch | Infrared remote control output |
| Input Modality | | |
| Microphone | The "clapper" <i>clap to turn lights on or off</i> | VCR: try to skip commercials on the basis of volume |
| Tone detection | Musical light show <i>turn lights on and off in response to music detected</i> | Scan radio stations searching for favorite songs |
| Touch sensor | Turn on light when door is opened | Mute television when phone is off hook |
| Remote control | Turn lights off from bed | Remote control repeater <i>replay remote control signals sensed. several repeaters can allow controlling a stereo from several rooms away.</i> |
| Knob & button | | Programmable remote control |
| Light sensor | Turn on night light when it gets dark | Detect mouse crawling to cheese bait, and record with camcorder |
| Infrared person detector | Turn on light when person is detected | Record people who walk by with camcorder |
| Angle/rotation sensor | | Turn off TV if exercise bike isn't being ridden |
| Temperature sensor | Control a heat lamp to regulate temperature | |
| Distance sensor | | |
| Time | Deter burglars <i>write a program to turn lights on and off at somewhat random times</i> | Turn off stereo after an hour <i>play music until person falls asleep</i> |

Figure 2.1, part 2 of 3
Activities at intersections of input and output modalities

| | |
|---------------------------------|--|
| Output Modality: | Screen |
| Input Modality | |
| Microphone | Record how often your phone rings |
| Tone detection | Musical tuning aid |
| Light sensor | See if fridge light really goes off when you close the door. |
| Infrared person detector | Count people passing by |
| Angle/rotation sensor | Measure motor RPM |
| Temperature sensor | Thermometer |
| Distance sensor | Electronic tape measure |
| Networking | Send secret messages to someone else with a brick |
| Time | Digital clock |

Figure 2.1, part 3 of 3
 Activities at intersections of input and output modalities

This grid of activities is not intended to be complete by any means. Rather, it is indented to show the breadth of applications enabled by a wide variety of input/output modalities.

2.3 The Need for Multiple Processes

Prior to the design of the programmable brick, experience with children using the LEGO/Logo system suggested that programs would often be short and simple, and when they became complex, it was not complexity in terms of data structures, but rather in terms of program control flow. In other words, programs didn't often record and store up data in complex ways. The simplest programs sequenced a set of actions to be taken by actuators: turning motors and lights on and off. Programs of

medium complexity would generally react to sensor inputs and modify the sequence of actions taken by actuators in simple ways.

2.3.1 Multiple Processes Simplifies Programs

The most complex programs written using LEGO/Logo would generally react to many sensors and modify actuation based on these. However, in many cases, these programs, which looked complex, were intended to perform multiple, simple functions. The complexity was caused by trying to do more than one thing at once, inside a programming model that allowed only one thing to happen at a time.

One such case encountered by this author was during a LEGO/Logo workshop of fifth graders in which the goal was to create a make-believe "candy factory." One workshop participant had created a set of connected conveyor belts to move candy, and a "masher" in the middle of the conveyor belt to squish down the candy when it came by.

The program started off simple, with a light sensor placed in such a way that the program would know when a piece of candy was coming by the masher, and pushed the masher down for a while, then back up, like so:

```
to factory
  listento 7
  waituntil [sensor?]      Waits until candy in front of light sensor (sensor 7)
  talkto "a
  onfor 5                   Turn masher motor on for half a second (forward)
  rd                       Reverse masher motor
  onfor 5                   Turn masher motor for half a second (backward)
  rd                       Reverse masher motor
  factory                  Start back over from beginning
end
```

The participant had many problems with the mechanical structure of his linked conveyor belts -- some belts were high, some were low, and all were hooked to a single motor through rubber bands on pulleys. After spending some time trying to reinforce the structure, the participant decided to turn the "bug" of the somewhat unreliable conveyor system into a sort of "feature" -- he wanted to add to his program a test for when

the belt stopped running, and set off an alarm. He asked how to do this at the same time as his first program, and needed help to create a contorted program similar to the following, which checked a rotation counter connected to an axle on the last conveyor belt to make sure it kept spinning:

```
to factory
  wait-for-candy
  talkto "a
  onfor 5
  check-for-breakdown
  rd
  onfor 5
  check-for-breakdown
  rd
  factory
end

to wait-for-candy
  check-for-breakdown
  listento 7
  if not sensor? [wait-for-candy]
end

to check-for-breakdown
  resetc
  wait 5
  if counter = 0 [tone 500 10]
end
```

*Reset the counter
Wait for half a second
If the counter hasn't
incremented,
sound the alarm*

The flow of control for this program has become much more complex. Breakdowns are checked throughout the code. Even so, the code doesn't do exactly what the participant wanted: occasionally a piece of candy would pass by the masher's sensor during the execution of **check-for-breakdown**, since the breakdown check took half a second to run.

With a programming environment that allowed multiple threads of execution to be run at once, the participant could have simply added the following to his first program:

```
to check-for-breakdown
  resetc
  wait 5
  if counter = 0 [tone 500 10]
  check-for-breakdown
end
```

`check-for-breakdown` and `factory` could then be run at the same time, either by starting them manually, or with a procedure like this:

```
to start
  launch [factory]
  launch [check-for-breakdown]
end
```

By running `check-for-breakdown` and `factory` at the same time, the correct effect is achieved, with shorter and much simpler code, and without the bug of sometimes missing the candy.

2.3.2 Having Multiple Processes Allows for New Complexity

In addition to making some previously complex programs simpler, having multiple processes allows for new types of complexity. New language features allowing interaction between processes enables new models of computation to be explored.

One form of interaction between processes is the behavior model used by Brooks for programming robots [Brooks 89]. A primary feature of this model is that processes can effectively enable and inhibit each other.

Consider the following case, based on one workshop participant's programming of an "artificial creature" using the programmable brick. The participant had developed two separate processes, or behaviors, like so:

to follow-light

Follow a bright light. sensor A is a light sensor mounted on the left side of the robot, and sensor B is a light sensor mounted on the right

If light is brighter on left, steer to the left.

if sensora < sensorb [motora, on motorb, off]

If light is brighter on right, steer to the right.

if sensora > sensorb [motora, off motorb, on]

end

to avoid-obstacles

Go forward until the front bumper is hit (touch C). When bumper is hit, take evasive measures

if touchc [motora, rd on

motorb, rd on *Back up*

wait 20 *Wait 2 seconds*

motora, rd *Spin*

wait 5 *Wait half a second*

motorb, rd] *Go forward again*

end

The process mechanism given by the programmable brick made an implicit "loop" for each process: a procedure would be repeated while the its process was turned on. At first the user turned on and off the processes manually, only using one at a time. After some time, the participant decided he wanted the robot to do both. But running both processes at the same time led to bad behavior: when **avoid-obstacles** saw that touch sensor C was pressed, it first tried to run both motors full reverse, and wait for two seconds. Unfortunately, during the two seconds, **follow-light** had usurped control of the motors, turning one or the other off based on the light readings. What was needed was a way to have one of the processes inhibit the other.

With some help, the participant fixed his problem with a change to `avoid-obstacles` similar to the following:

```
to avoid-obstacles
  if touchc [  disable-follow-light1           Temporarily disable the
                                                    follow light process.
              motora, rd on
              motorb, rd on                     Back up
              wait 20                           Wait
              motora, rd                         Spin
              wait 5                             Wait
              motorb, rd                         Forwards again
              enable-follow-light2           Re-enable the follow
                                                    light process.
end
```

Now, the avoid obstacles behavior could turn off the follow-light behavior when it wanted to force the robot to back up and spin in order to get around an obstacle, thus keeping the follow-light process from interfering with motor control.

Although this example was quite simple, many processes can be intricately connected in this manner -- simple behaviors can be combined in different ways to produce more complex actions.

2.4 The Need for Multiple Bricks Interacting

One motivation behind the programmable brick project is the ability for multiple bricks to interact with one another, either through various signaling technologies (such as infrared or sound), or through modifying and then sensing the state of the environment. One motivation for doing this comes from previous research with StarLogo, designed by Mitchel Resnick [Resnick 91]. StarLogo is a software environment for experimenting with virtual worlds that have many interacting entities.

¹The actual primitive used was `stop-menu-1`. In this version of the brick software environment, each process had a menu item on the brick's control panel, and the user could manually turn these processes on and off by selecting them with the knob and clicking the button to start or stop. Thus, it was a natural to add primitives to allow the user to start and stop menu items from program control.

² `start-menu-1` was used here.

People using StarLogo can explore how many simple things can interact to produce complex behavior. This complex behavior can appear organized, even though there is no organizing entity: the organization emerged from the interacting parts. For example, StarLogo has been used to explore how ants, even with extremely simple rules governing their behavior, can exhibit complex and seemingly organized behavior in a colony. People are often surprised by the idea that organization can arise without an explicit plan or mechanism for control.

The programmable brick, through its ability to interact with other bricks, can take these sort of experiments out of the computer and place them in the real world.

Chapter 3: Background

3.1 Ubiquitous Computing

The idea of "ubiquitous computing," of computing capability spread throughout the environment, has received great interest in recent years [Weiser 1991] [Wellner, Mackay, Gold 1993]. As the size of the electronics required for computers gets ever smaller, it becomes easy to include computing in electronic items that previously had none, or even in devices that weren't previously electronic at all. Microwave ovens of a few years ago could only be set to cook at a certain power for a certain time; today, microwaves commonly do things like automatically defrost meat based on its weight by calculating an optimal sequence of different cooking powers over time. Until recently, cars used mechanical systems to adjust the intake of air to match fuel; today, many cars use computers to precisely adjust the fuel/air mixture to reduce exhaust pollutants based on dynamic feedback from sensors in the emissions system.

But sometimes by embedding computing into an everyday artifact, we see more than just a simple optimization of its existing functionality. Ordinary white boards, with computing and sensing built in, might be able to record and remember things drawn on it, and a smart telephone system, connected to person-tracking sensors, can direct incoming calls to the phone closest to where a person happens to be at the time.

The programmable brick is in many ways inspired by the ubiquitous computing movement. One way in which the programmable brick extends the body of work in ubiquitous computing is that it focuses on how kids would use ubiquitous computing. But perhaps more importantly, the programmable brick project differs in that it focuses on

kids *designing and creating*, rather than just using, ubiquitous computing artifacts.

Most of the thinking about ubiquitous computing has focused on people using ubiquitous computing as consumers, rather than as builders. This is natural since probably the closest relative to ubiquitous computing today is personal computers, and most people who use desktop computers today are consumers, rather than programmers or computer hardware designers. But there are a few reasons it might be important to consider people creating and programming their own ubiquitous computing artifacts.

Consider the history of personal computing: in its infancy, a very large fraction of its users were programmers. Personal computers came with programming languages built in. The wide variety of ideas that came out of those days helped make personal computers the widespread and important tools they are today.

But another point to consider is this: ubiquitous computing may be fundamentally different from desktop personal computing. Today, there are only a few "killer" applications for personal computers -- applications which people on the whole spend most of their time using. Today, these are word processors, spreadsheets, graphics and presentation software, and telecommunications software.

Perhaps the uses of ubiquitous computing, though, can't be boiled down easily into a few killer applications. If ubiquitous computing turns out to be characterized by a large variety of often transient uses, as possibly suggested by the previous chapter, the user may wish to have a general-purpose tool like the programmable brick, which lets the user make a custom solution to each custom problem.

And of course the most important reason to focus on people creating, rather than just using, ubiquitous computing artifacts has been explained before -- the primary goal of this project was to create a constructionist learning environment. So, at its core, this project focuses

on people creating ubiquitous computing artifacts because it focuses on people learning through creating.

3.2 Related projects

3.2.1 Logo Brick

The Logo Brick was a first attempt at making portable computation for kids that interfaced to sensors and motors [Martin 88] [Bourgoin 90]. However, it had a lot of failings: it was not robust (requiring MIT people around to fix it when it broke), was relatively large, and had limited I/O modalities (touch and digital light sensing were its only inputs). The brick did not allow for easy user interaction when disconnected; it had no screen, and had only a single input: the "run" button. The programmable brick addresses the shortcomings of this project, in addition to bringing new approaches to software environments, and new perspectives on the use of portable computation for kids.

3.2.2 MIT LEGO Robot Design Contest

The programmable brick shares some of its underlying hardware design with the portable robot-controller hardware developed by this author and Fred Martin for the MIT LEGO Robot Design Contest [Martin, Sargent 92]. The LEGO Robot Design contest is an annual event in which MIT undergraduates design and build LEGO robots to compete against each other. Robots typically do things like collect ping pong balls from a small playing field, or to locate and stack blocks. The hardware from the robot contest controller board could not be used directly, though: the system was developed for use by university students comfortable with circuit building and debugging. It is large -- about twice the size of the programmable brick (mostly to make it easy to solder by the students). It has no case, so it has lots of easily breakable wires going everywhere, which is OK for the university students, who can fix them. The controller board is also much simpler than the brick, missing much of the brick's functionality. Finally, the software for the MIT LEGO Robot system (a

variant of the C programming language) seemed to be inappropriate for younger kids.

3.2.3 Braitenberg Brick System

The "Braitenberg Brick" system, designed by Fred Martin, allows children to build robots to explore the concepts in Valentino Braitenberg's "Vehicles." Children build circuits with sensors and logic bricks (such as and's, or's, and one-shots) that endow their robots with various behaviors [Hogg, Martin, Resnick 91]. This system shares with the programmable brick the idea of portable computation that can be hooked up to the environment. The Braitenberg brick system, however, does not implement a general-purpose computation model: while it is easy to make simple "reflexes" that couple sensors to actuators, computation of the complexity typically seen on computers is at best difficult, and usually impossible, to implement using the system. The programmable brick project further differs from the Braitenberg brick system in its larger variety of I/O modalities and the focus on embedding computation in the real world.

3.2.4 Multiprocessing Logo

The programmable brick is programmed by kids in a version of the Logo programming language. The Logo environment for the brick allows multiple threads of execution to take place simultaneously, following the ideas of the research environments MultiLogo [Resnick 88] and Video Game Logo (by this author), and the recent commercial product Microworlds Logo. The user-interface for the programmable brick software environment borrows many ideas from Microworlds Logo.

Chapter 4: Hardware Design

The hardware design of the programmable brick attempts to achieve several sometimes conflicting goals:

- Brick should be portable:
 - Able to fit in pocket
 - Low power so can be run from batteries
- Should be able to be left in environment
- Should support large variety of I/O modalities
- Should support desired computation environment
- The brick should be practical to implement

The juggling of these constraints in the design and implementation of the brick hardware was time-consuming and difficult.

4.1 Processor Selection

One important aspect of the programmable brick hardware design was the choice of microprocessor. The choice of microprocessor affected ease of software development, features available to the user, power management, and size considerations.

After consideration of many processors, two main families emerged. One was small embedded controllers with good I/O but minimal processing power. The other was larger processors with higher processing power, but more limited I/O. A representative from each of these groups was chosen on the basis of familiarity and ease of software development.

The representative selected from the first group was the Motorola 6811. This processor had been used in previous hardware designs by this author and Fred Martin for the MIT LEGO Robot Design Contest. This

processor was small, had good I/O, and low power consumption, but had a relatively old instruction set, requiring programming of the core software in assembly language. However, software developed for the MIT Lego Robot Design Contest could be adapted for use in a programmable brick based on this processor.

The representative selected from the second group was the Motorola 68332. Although the 68332 used more power than the 6811, it incorporated advanced power management techniques that allowed reduced power consumption during idle times or times with low processing load. A design with the 68332 would be much larger than a design with the 6811 due to the increased memory requirements, as well as the need to add I/O devices.

From a software standpoint, the 68332 used the same instruction set as the Motorola 68000, so it would likely be able to run versions of Logo originally written for 68000 machines such as the Macintosh. Using the 68332 would make possible certain processor-intensive primitives such as sound processing and filtering that would be impossible with a slower processor such as the 6811. It was felt that since the 68000 could be programmed in higher-level languages than the 6811, development of new programming environments for it would be easier, and those programming environments could have more functionality as a result of the increased processing power.

The first design for the programmable brick hardware incorporated the 68332, primarily due to the perceived software benefits later on. In the course of actually trying to lay out the design on circuit boards, it was found that the 68332 caused a larger size increase over the 6811 than was originally realized, and entailed a fair bit more complexity as well. Also, the leading implementation candidate for the brick programming language at the time had been a modified version of the then-being-developed Microworlds Logo for the Macintosh, but it was found during the implementation of Microworlds that it would be too large to fit on the programmable brick. For these reasons, the 68332 design was abandoned in favor of a simpler design with the 6811 processor.

4.2 User Interaction

While the user can interact with the programmable brick through the host computer when the host is connected, the user is forced to use other means when the brick is away from the host. However, the brick clearly did not have room for the standard computer monitor and screen: even in today's smallest portable computers, a keyboard and monitor place serious restrictions on their minimum size.

We decided to outfit the brick with a small LCD screen capable of displaying two short lines of text (and limited graphics), two pushbuttons, and an easy-to-turn knob. This minimal interface would allow the user to select items from a menu displayed on the LCD screen by twisting the knob to the appropriate item, and selecting it by pressing one of the buttons.

Note that in many applications, the brick is embedded in a project that is moving or otherwise difficult to access. In this case, the screen, buttons, and knob are of limited use, although the speaker and infrared remote control can be used to some extent instead (see section 4.5).

4.3 Integrated vs. Separate Batteries

A big decision in the design of the programmable brick hardware was related to batteries and motors. While processors could be found that used very little power, running mechanical LEGO creations with motors simply required lots of power. The power required to run motors for a reasonable length of time resulted in batteries larger and heavier than the electronics circuitry for the brick itself. The power required for processing, using sensors, using the brick's display, and emitting sound were relatively quite low, requiring only a small battery.

One of the most obvious uses for the programmable brick is to make robots and artificial creatures. But, from the beginning of the programmable brick project, a conscious effort was made to explore uses

beyond just making mobile, motorized machines. One such application driving the design of the brick was the idea of taking the brick out into the world to collect sensor data, such as the pH levels of local streams. For this application, a small brick that allowed its user to put the brick in a pocket was very appealing. During the brainstorming for possible applications for the brick, many other applications were found that either didn't require motors, or required use of motors very intermittently (such as mechanically switching on and off a light switch when someone was sensed to pass through a doorway).

One way of possibly allowing the brick to be small when motors weren't being used heavily was to incorporate a small alkaline battery into the brick itself, and have an optional large rechargeable battery pack that could be attached to the brick only when necessary. This approach had several advantages and disadvantages.

Advantages of separate processor and battery brick:

- Processor brick would be much smaller when the battery brick wasn't needed.
- Battery bricks could be recharged while disconnected from processor brick. Since battery bricks would be cheaper than processor bricks, more could be made, and allow swapping batteries when one set ran out, allowing 100% usage of the processor brick. (A single brick would have to be taken out of service and recharged when its batteries ran out).

Advantages of a single brick with both the processor and a large set of batteries:

- More reliable: fewer connectors means fewer problems.
- The single brick would be somewhat smaller in the case when the large batteries were necessary.
- The double brick solution requires two sets of batteries; the single only requires one.

The size of the brick seemed to be the overwhelming consideration, so the separate brick solution was chosen. During use of the programmable brick, very efficient, low power motors were found that allowed many of the traditional, motor-intensive projects to get away with using the processor brick alone.

4.4 Power Management

Although the electronics of the programmable brick did not use much power, it was important to consider ways to further reduce the power used by them. The programmable brick was envisioned to be used in ways requiring it to be on for large periods of time -- perhaps taking weather samples over a few days. For this reason, it was important to be able to slow down or shut off certain functions when not in use to reduce power consumption.

Many of the sensor systems (including 5 of the 6 infrared inputs) were placed on a secondary, switchable power supply within the brick. This allowed the brick's power consumption to be reduced by about 30% during times the sensors were not required. Even if sensors were being used as often as once a second, this subsystem could be turned off in between samples to save power.

The second aspect of power management related specifically to LEGO type active sensors. LEGO type active sensors were designed to be given power with a high duty-cycle, with small pauses in the power to take a reading from the sensor. Through experimentation, it was found that the sensors could alternatively be powered with a large burst immediately before a reading, requiring power only when the sensor was actually being read. This resulted in around an overall 30% power savings for a brick system that used two active LEGO sensors, such as the light reflection or angle sensors.

Another way to save power in a processor system is to reduce the processor clock rate, or stop the processor altogether. The programmable brick did not incorporate such a feature, although its

circuit board had a connector that could be used for possible future experiments to add a small circuit to halt the processor for periods of time.

4.5 I/O Modalities

As stressed in chapter 2, a central feature to the programmable brick is its large variety of I/O modalities.

In addition to built-in I/O modalities, the brick is designed to attach to external sensors and actuators, allowing the user to configure the I/O modalities as desired, and allowing the creation of new types of sensors and actuators that can be attached to the brick.

One important part of the design for the modalities was deciding which would be physically built-in to the brick, and which would be separate and attached to the brick only when needed.

Reasons to physically include specific sensors/actuators in the brick were:

- Smaller combined size than separate brick and external device
- Increased convenience of using feature
- Increased reliability (more connectors -> less reliability)
- Some devices are difficult to interface through standard sensor/actuator ports

Reasons to use separate devices for sensors/actuators:

- Device's position needs to be flexible. Motors and many sensors, such as touch sensors and directional light sensors fall into this category.
- Device is not used very often. Since space in the brick is at a premium, unimportant modalities should not be built-in.
- Device is large and would add too much space to the brick.

A discussion of the various modalities included physically in the programmable brick follows.

4.5.1 Sound output

There are two important uses for sound output on the programmable brick. One use is for providing debugging feedback to the user from within programs. The other is to treat sound output as an end in itself, whether for music, communication, playing back sampled sound, etc. Since the brick would often be embedded in a moving or otherwise difficult-to-access project, the brick's screen would not always be visible. Therefore the speaker's ability to give debugging feedback was important enough to warrant the speaker's placement inside the brick.

4.5.2 Sound input

Through brainstorming uses for the brick, sound input was considered to be important, although not as important as sound output. Sound input requires a large amount of support circuitry, although the microphone element itself can be quite small. It would be annoying to have to make separate circuit boards as part of an external sound sensor. This factor led to the decision to place the sound sensor inside the brick.

4.5.3 Infrared input

The two primary uses envisioned for the infrared input were allowing the user to interact with the brick using standard infrared remote controllers, and to allow wireless communication between bricks. Although brick-to-brick communication would probably not be used very often, the remote control could serve as a keypad for the brick when it was hard to access. Therefore, as with sound output, infrared input was considered to be important enough to warrant its inclusion in the brick.

4.5.4 Infrared output

Infrared output is important for wireless communication between bricks, as well as allowing the brick to control a variety of standard electronic devices such as TVs and camcorders. Although these features might not

be used in the majority of brick applications, infrared output requires special modulation circuitry that would be extremely difficult to interface with through standard actuator ports. This factor led to the decision to place the infrared emitters inside the brick.

4.6 Design of ports for external sensors and actuators

The design of the ports to interface with external sensors and actuators was both made simpler and more complex by the existence of a previous standard. LEGO had designed a standard two-wire port connector for motors and sensors. LEGO motors are easy to interface to, but the two-wire design of the LEGO sensor port led to much complexity both in the brick and in the sensors themselves.

Figure 4.1 shows the standard LEGO 2-wire connector. The apparent simplicity of snapping the two connectors together like standard LEGO blocks is misleading. The shape of the metal contacts is actually quite clever, allowing the connectors to be put together in any of four relative orientations without shorting the two signals together. Two of the orientations connect with one polarity, while the other two orientations connect with the other polarity.

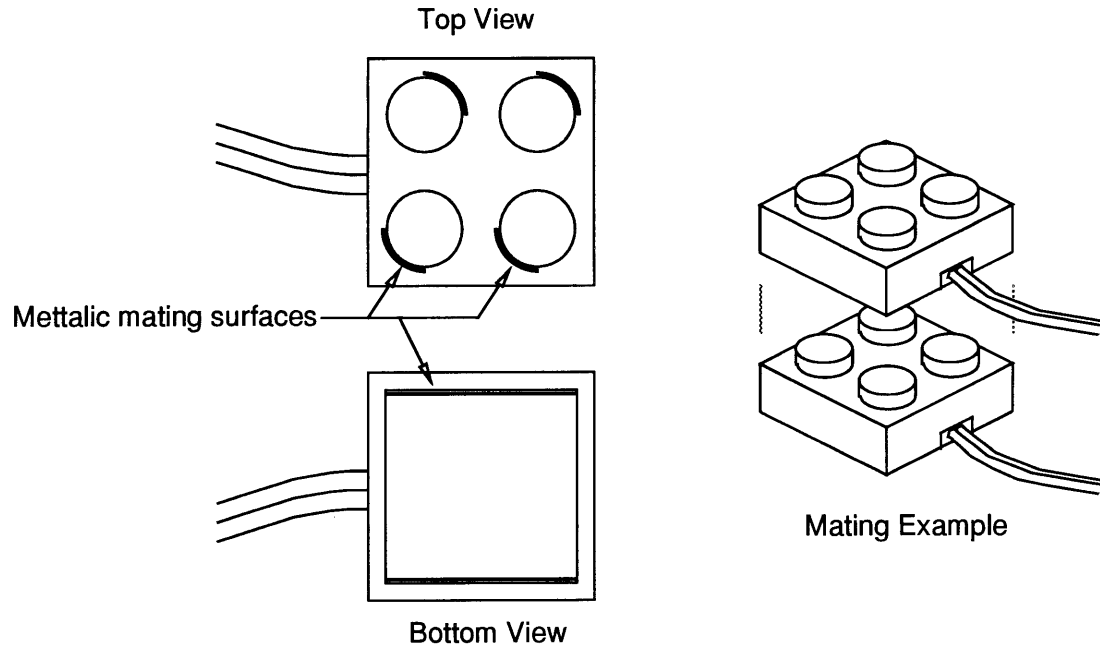


Figure 4.1
Standard LEGO 2-wire Connector

This connector is ideal for motors. Motors simply need two wires, and being able to reverse the polarity of the connection is a big plus, since the motor's direction depends on the polarity. (Swapping the direction of the motor doesn't require the user to change his software -- simply rotate the motor connection).

On the other hand, a two-wire port without guaranteed polarity is difficult to use for sensors. Some simple, passive sensors can give feedback through a two-wire system by simply changing their effective resistance. Some passive sensors, however, such as phototransistors, require that the direction of current flow be known. In the case of sensors which require power also, such as a reflected light sensor (which must shine a light), or the standard LEGO angle sensor (which has a quadrature break-beam rotation sensor), power of appropriate polarity must somehow be sent to the sensor in addition to the sensor sending back its measured value.

The LEGO company solves the problem of powering active sensors by sending power to the sensor over the two wires, and then periodically shutting power off and listening for a signal back. This requires some circuitry on the sensor to store up the power and sense when the power is off in order to send the signal back. Power and signal are multiplexed on the same lines. LEGO's scheme for dealing with the unknown polarity of the sensor connection is to pack the sensor with yet more circuitry to rectify the incoming current and send back the signal correctly regardless of the polarity of the connection.

A simpler connector standard, from an electrical standpoint, would connect three wires of guaranteed polarity between the brick and the sensor. Two wires would supply the power, and one would be the return signal from the sensor. This would require no special circuitry at the sensor to rectify or store up the power, and no circuitry to switch on and off the transmission of the sensor's measured signal.

A big advantage to adhering to LEGO's sensor standard is that, despite their internal complexity, LEGO's sensors are very high quality, and it would be unfortunate (and probably unsuccessful) to have to attempt to duplicate them ourselves if we chose our own 3-wire standard. However, it was important to the brick project to be able to design and build our own sensors, and using the LEGO standard would make this prohibitive in many cases. We decided each factor was important enough to warrant the support of two different sensor standards: in the programmable brick, there are four standard 2-wire LEGO sensor ports, and four 3-wire sensor ports for our own, home-brew sensors.

4.7 Functional Description of Programmable Brick Hardware

This section gives a functional description of result of the previous design decisions, and their implementation.

The programmable brick is a small (about 2"x3"x1.5"), battery-powered computer with a wide variety of I/O features. It uses the Motorola 6811

microprocessor and contains 128K of RAM which is saved on power down.

I/O features:

- 4 ports for connection to actuators compatible with the 9 volt LEGO system (motors and lights)
- 4 ports for connection to sensors compatible with the 9 volt LEGO system (light, touch, rotation, temperature)
- 4 ports for connection to sensors to be made at MIT
- Built-in microphone and speaker for 8-bit sampled audio input and output. This port will be able to do things like record and play back sound and detect noise levels. Depending on software, it may also be able to do rudimentary audio processing such as pitch or spectrum detection, and various processing effects on the audio output.
- 6 built-in IR transmitters and receivers for communication between bricks and to consumer electronics devices that use handheld remote controls, such as TVs, VCRs, and camcorders.
- Wired network connection for hooking up to host computer, other bricks, and sources of power (including the battery brick).
- Small, character-based LCD display (16 characters by 2 lines), for user interaction.
- Two pushbuttons and one knob, for user interaction.

Chapter 5: Software Design

In many programming environments, the programming language comes first, and the user interface for the programming environment is a secondary consideration. But for the programmable brick, the user interface and language were equally important. One reason is that a well-thought-out and easy-to-use interface is particularly important for computer novices. Another is that, as it turns out, the user interface design touched on some complex issues, such as the relationship between the programmable brick and a host desktop computer when the brick is connected to one.

After the overall software design, two software environments were created. Although both grew out of the software design which follows, each has a different underlying implementation, and made slightly different decisions which affect the user. The first environment was implemented by this author, the MIT undergraduate Victor Lim, and the high school student Andrew Blumberg. The second environment was implemented by Brian Silverman of Logo Computer Systems Incorporated, who has long worked with our group at the Media Lab. Both will be discussed in this chapter.

5.1 Programming Language Selection

Several programming environments were considered for use with the programmable brick. Environments considered included traditional, text-based programming languages, as well as graphical programming languages. The Logo programming language was chosen as the first to implement, primarily because of its design for use by computer novices, and partly out of familiarity by the implementors.

Many other approaches were considered, and on the whole, not many approaches were ruled out. Other interfaces are currently being tried -- there are now several efforts to make visual language environments for the brick that let the user create programs in new ways, as well as see and interact with programs as they are running.

Section 5.4 will present the ways in which the traditional Logo language was modified for use with the programmable brick.

5.2 Programming Environment User Interface

Because the programmable brick has a small screen and no keyboard, it was decided early on that it would be important to hook the brick to a host computer. The host computer would be used as a "dumb terminal": it would lend its screen and keyboard to the brick. This would allow programs to be typed in, and give the user fuller interaction for viewing program state and debugging than the brick's small screen, buttons, and knob would allow for. Later on though, the idea of making the host a "dumb terminal" to the brick was called into question; this will be discussed in the next section.

5.2.1 Host Computer Interface

The starting point for the design of the host user interface started with a then prototype of the Microworlds Logo programming environment from Logo Computer Systems Incorporated. The Microworlds user interface consists of several windows, as shown in Figure 5.1.

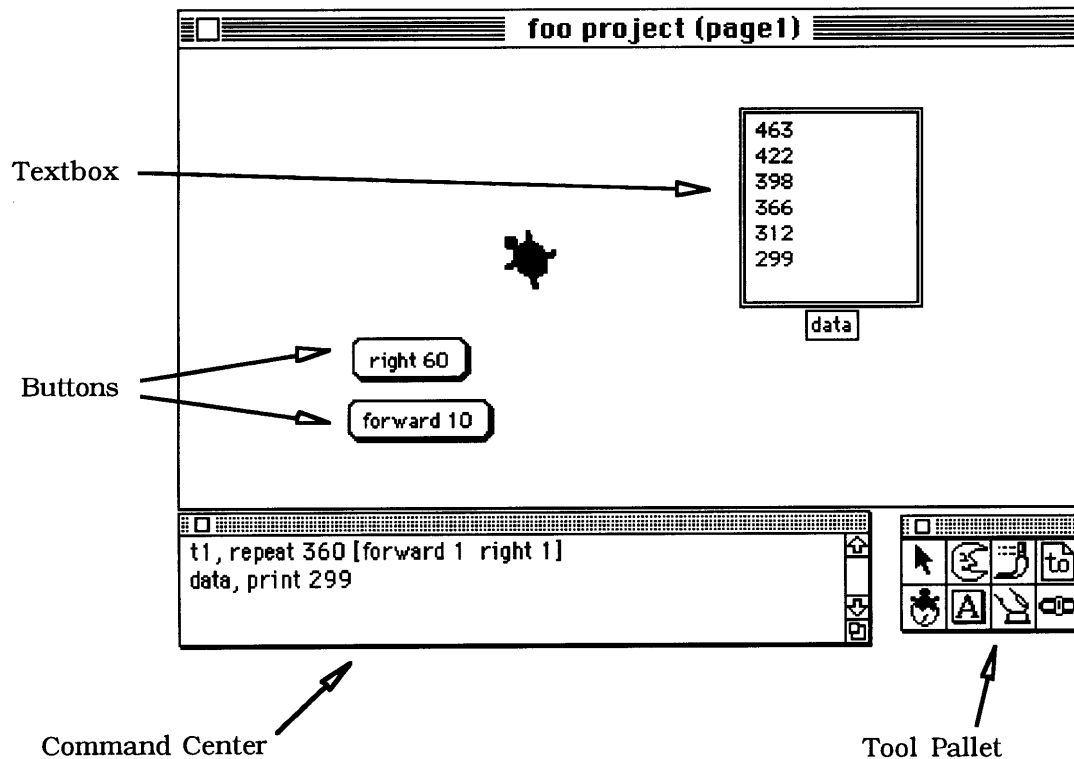


Figure 5.1
Microworlds Logo User Interface

The largest window is a place where the user can create and place various objects, such as buttons, sliders, text boxes, and graphical turtles. Logo expressions are embedded in buttons to be run when clicked, and the text boxes are good for things like showing various types of state, or recording state over time.

The command center is a small window in which Logo commands may be typed in and tried out interactively. Programs larger than a few statements are usually written as procedures in Logo; the procedures page (not shown), is a simple text editor in which the user can enter these.

5.2.2 Interaction between brick and host

In the early design phases, however, it became clear that there were certain tradeoffs to the model that the brick simply used the host

computer as a "dumb terminal" -- perhaps the model of the host and brick as two communicating computers, with each being programmable, would lead to more flexibility. One reason the user might want to be able to program the host in addition to program the brick is that the host is a more capable computer -- some desired features might simply not be available on the brick (for an example, see the discussion of turtle graphics, later in this document). Another reason is that the user might want to do things like upload sensor data taken from the programmable brick and analyze it while the brick is off gathering more data.

One concern with the "smart host" approach was that a novice user might be overwhelmed with the prospect of programming two separate computers. Whether the two programming environments are in different applications, in the same application but with disjoint sets of windows, or sharing the same windows, there would be added complexity along with the added flexibility. Since the target audience for the brick is computer novices, raising the threshold for first using the brick seemed like a bad idea. A secondary concern with the "smart host" approach was that it would probably take more effort to implement than would a simpler model.

It seemed like an ideal solution would neither raise the threshold of using the brick for the majority of the uses which wouldn't need a host Logo, nor penalize people who wanted the added flexibility of programming the host also. Here are the different approaches that were considered, and a short analysis of each.

1. Have the host side not be programmable. Things like off-line data analysis could be done on the host in an application separate from the programmable brick interface, such as a spreadsheet, or programming environment unrelated to the brick. This separate application wouldn't be able to show data as interactively, though, so viewing a graph on the fly would be difficult.
2. At the other extreme, have there be no clear distinction between the host and the brick computers. Programs would by default run on

the brick, but if the brick were disconnected, the user could keep interacting and computation started in the command center would run on the host computer instead. An error would simply be given if the user tried to execute a brick primitive when the brick wasn't hooked up. This approach was given up as unworkable -- it would lead to all sorts of confusing behavior and inconsistencies between brick and host state.

3. The third approach was to explicitly have a model of two computers, independently programmable, that communicate with each other. A given computation explicitly lives in one place or the other. Some buttons and textboxes would belong to the brick, and some would belong to the host computer. Perhaps there would be two separate command centers as well. This approach seemed to violate the constraint of keeping the environment easy to use for novices.

4. Anticipate what the user might want to do with the host apart from the brick, and implement it. If off-line data analysis is expected to be important, simply pre-package it into the interface. Unanticipated uses could fall back on approach 1: simply exporting the data to another environment. This was the least complex option to the user, but had the least flexibility as well.

5. Let the user program both the host and the brick, but have the host programming language be invisible to the user most of the time. Any pre-packaged graphing and data analysis tools (as in 4) would be written by the implementors in host Logo, but the casual user of these tools would never see host Logo code, nor a host Logo command center. Only the power-users who wanted to write their own Logo code to make a new data viewing or analysis tool would ever see the host Logo.

The first programmable brick environment stuck with the simplest of these implementations, option 1. The second programmable brick

environment was more like option 5, allowing the power user to use Logo to allow computation on the host computer.

5.2.3 Host and Brick state

The "smart host" approach complicated another aspect of the brick environment's user interface. With part of the user's project living in the host, and part of his project living in the brick, how would the user "save" his project to re-use at a later date? For example, should it be possible to store the combined host and brick state in the brick? Should it be possible to store the combined host and brick state in the host?

Some concrete examples of situations which would take advantage of these different capabilities follow:

- Brick can store host state:

Sue makes a data-collection program for her programmable brick, and takes it out into the field for a few days. She now wants to view the data and change the program using a computer at a different site. If the brick is capable of reconstructing the state of the host, all Sue has to do is hook up the brick. Otherwise, she would need to also bring along a floppy disk on which she had previously saved the host state.

- Host can store brick state:

Bob samples some musical instrument sounds into his programmable xylophone, and in fact has programmed a feature where he can record and play back sequences he pounds out on the xylophone keys. If the host were capable of storing the entire state of the programmable brick, it would be able to save these sampled sounds and the recorded songs to a floppy for Bob to get back to some time in the future.

At first glance, it seemed nice to be able to save the system state in either location. On the other hand, having each side able to reconstruct the other leads to some tricky problems. Consider what happens when the brick has uploaded its environment to more than one host, or when the host has downloaded its environment to more than one brick.

First, look at an example where the brick environment gets uploaded to more than one host:

Bob and Joe are working together at the host while it's hooked up to their programmable brontosaurus. Bob takes the brontosaurus away from the computer for a test-drive through the Cretaceous forest that's been set up on the floor. Joe starts adding some new procedures at the host (this assumes the host allows editing of programs while the brick is disconnected). After knocking over half the forest, Bob decides that he needs to slow down the brontosaurus by adjusting one of the variables. Seeing Joe in the middle of hacking on the procedure page on the host, Bob decides to hook the brick to a different host. The environment is now loaded into this second host and Bob turns down the speed. Playing with the dinosaur some more, Bob now adds a button to make the dinosaur turn around -- again using this second host. When it comes time to hook the brick back to the host where Joe has now finished his new procedures, there is a problem -- whose state should the brick now use?

It would be nice if the system would at least detect that this has happened, and ask the user before arbitrarily copying the host state to the brick (or vice versa). One possible further solution would be to try to automatically merge the changes Bob and Joe have made, which would probably require storing some previous version history from which a common ancestor could be taken. Automatic merging was rejected as too expensive to implement for the benefit it gave, and for the reason that it probably wouldn't do the right thing all the time.

The second case is where a single host downloads an environment into more than one brick:

Pat and Shawn are ready to clone their programmable ant a few times to see if any colony behaviors emerge. They construct new mechanical copies, each with its own programmable brick. They then hook each up in turn, one at a time, to the single host in order to download the environment into each ant. This doesn't seem to cause any special problems yet. But when Pat and Shawn give each ant some local state, perhaps different foraging parameters, or the ability for the ant to learn some behavior through interacting with the environment, the difficulty begins.

Pat and Shawn hook up ant A and change few lines of code. When they disconnect and hook up ant B to load the new code, ant B now receives the complete state of ant A -- it loses what it had learned before and now instead has all of ant A's learnings.

or

Bob's programmable xylophone gets copied, and each instrument had a different sampled sound it plays notes with. When Bob wants to add a volume control to his xylophone program, will he be able to make the change at the host once, yet have each of the xylophones keep their unique sounds?

A few different approaches to these situations were considered:

- Let the bricks keep their own individual "data" even when the "code" is changed from the host. "Data" probably means something assigned to a variable, while "code" probably means something defined by the user through the host interface -- such as procedures, or interface items like buttons. But what about the contents of a hypothetical text box originally created through the host but updated by the brick? Even if a division could be made that makes sense, one disadvantage to this approach is that the

user would have to understand this possibly tricky division. This approach of course also has the problem that a brick's state isn't completely stored in the host. Things like sampled sounds wouldn't be stored to disk.

- Make code uploads/downloads automatic (the host and brick are always in sync), but make data uploads and downloads be manual. This is a very flexible option, but might lead to more difficulty in using.
- Require that if different bricks won't be identical copies of each other -- if each has different local state -- that each has to have its own instance of the host application with which to talk. This would allow full saves of state on the host, and seemed to be the simplest model to the user.

As of this date, both brick environments use the simplest solution to the issue of host versus brick state: procedure definitions live on the host and are only downloaded to the brick, and data lives on the brick, and is not downloaded from the host. This means that the complete state of a brick project is spread between host and brick, neither side able to store the entire project. This has led to difficulties for users, but dodges many of the difficult issues brought up in this section.

5.3 Compiler Versus Interpreter for Brick Logo

Section 5.2.2 discusses the pro's and con's of how to divide what computation runs on the brick, versus what computation runs on the host, as the user views the computation. However, there are reasons we might want to make the internal implementation of the division of computation differ slightly from what is presented to the user.

Since the host computer is much faster and has much more memory than the brick, it would make sense to trade brick computation for host computation where it would make sense. It didn't seem to make sense to let the brick offload subcomputations to the host on the fly -- the

communications overhead would probably negate any gains attempted, and furthermore it would be of questionable use to have the brick be faster only when it was hooked up to a desktop computer if the main purpose of the brick is to run while disconnected.

But there was another way in which brick computation could be saved by extra host computation. By making the host precompile Logo code, the Logo code could run much more quickly on the brick. If done well, this would not cause a noticeable delay to the user when programs were downloaded from the host to the brick.

Here were the different options that were considered:

- Have the host translate Logo code directly into machine code. This would allow the brick to run Logo code very quickly. Unfortunately, native code compilers for small processors such as the 68HC11 used in the brick are complex to write and produce rather large code.
- Have the host translate Logo code into a byte code. The host translation program would be simpler than in the first case, and the code downloaded to the brick would use much less memory. It would require that a byte code interpreter be written for the board (but in fact one was already available). The code in this case would be a fair bit slower than native code, but it would still be quite respectable. Multitasking is easier to implement using byte code than with other approaches.
- Have a Logo interpreter on board. Writing a Logo interpreter in assembler for the 68HC11 would be somewhat time consuming, although source code for a single tasking Logo running on a similar processor was available. This approach would lead to a much slower execution speed than either of the compiled approaches.

The second option, compiling Logo into byte code, was chosen for both of the programmable brick environments implemented. One unexpected

disadvantage to this approach turned out to be that prototyping new and experimental host environments for the brick was hampered by the need to incorporate a compiler into these environments (a relatively complex piece of code).

Several experimental user interfaces for brick programming have stopped short of including a compiler and thus being able to download new programs. Instead, they have used the much less general approach of changing behavior by changing variables in a resident brick program created with use of one of the original environments. One example is a Braitenberg-like creature editor written by Rick Borovoy, using the Apple Newton. The user can change the behavior of a small LEGO creature equipped with two motors and two light sensors by drawing connections between graphical representations of the sensors and motors on the Newton screen. But the interface does not download new executable code to the brick. Instead, there is small Braitenberg circuit simulator on the brick (created with one of the Logo environments). When the user draws a new circuit, the Newton simply downloads different connections weights to the brick, changing the simulation. This approach unfortunately does not allow the user to change the basic topology of the Braitenberg logic elements.

5.4 Language Differences Between Brick Logo and Previous Logos

As discussed in chapter 2, experience with children using the LEGO/Logo system suggested that programs would often be short and simple, and when they became complex, it was not complexity in terms of data structures, but rather in terms of program control flow. The experience further showed that a program in LEGO/Logo with complex control flow often was only trying to do multiple simple things at one time.

This prior experience led to the emphasis on multiple processes and new control flow constructs, and to the de-emphasis of data structures in the design of the programming language that executed on the brick.

5.4.1 Multiple Processes

The need for multiple processes was accepted early in the design of the programmable brick software. What was left to decide was the set of primitives the user would be given to make use of and control the multiprocessing of the brick.

Traditionally, the design of multiprocessing languages has been rather complex, focusing on communication and synchronization between processes. But it's important in this case to understand how the programmable brick's reason for using multiple processes differs from that traditionally emphasized in computer science. Much of the traditional study of parallelism has been to take problems that can be easily solved sequentially, and develop algorithms for solving them in parallel for the purpose of speed. By splitting a problem among multiple processors, the group of processors may be able to find a solution faster than a single processor.

The use of multiple processes discussed here is for the purpose of providing a computational model more appropriate to the task at hand. By disentangling unrelated or only loosely related control flows, a program is simpler, and its structure more closely matches the problem at hand.

Because of the loose or nonexistent coupling of the multiple processes on the programmable brick, special communication and synchronization primitives were seen to be unimportant. When necessary, processes would communicate directly through the modification of global variables. More often the case was that processes interacted indirectly by modifying and then sensing physical state outside the computer.

5.4.2 Rules vs. Procedures

Brick programs written to use multitasking often exhibited certain patterns. It was very common for the user to write simple loops, which

checked for certain sensor conditions, and then took some action when the sensor conditions were met.

Here is an example, taken from a case where the programmable brick was mounted atop a LEGO train, controlling its movement. The train had two light sensors: one sensed when the train moved by a stationary light source mounted next to the track before a tight curve, and the other sensed a taillight mounted to another train occupying the same track.

The goal for the algorithm of the train was, at one point, to slow down for a short time when the curve marker was detected, and to back up for a short time (then go forward again) when the other train was detected in front.

Here is what the program would have looked like with standard procedures and multitasking:

```
to check-for-curve
    When the side-mounted sensor sees a bright enough light, reduce speed for two seconds.
    if [sensora < 20] [setpower 4 wait 20 setpower 8]
    check-for-curve
    Back to the beginning
end

to check-for-train
    When the front-mounted sensor sees a bright enough light, put train in reverse for 3 seconds, then back forward.
    if [sensorb < 20] [rd wait 30 rd]
    check-for-train
    Back to the beginning
end

to go
    launch [check-for-curve]
    start a separate process that runs check-for-curve
    launch [check-for-train]
    start a separate process that runs check-for-train
end
```

The actual program, in this case, used a new primitive called **when**, which simplified this common case of condition-action pairs:

```
to go
  when [sensora < 20] [setpower 4 wait 20 setpower 8]
  when [sensorb < 20] [rd wait 10 rd]
end
```

when is simply a syntactic shortcut for launching a loop with an **if**:

```
when condition action
is translated into
launch [forever [if condition action ]]
```

(where **forever** simply implements a never-ending loop.)

5.4.3 Edge vs. Level Sensitivity

Although condition-action rules seem fairly simple, certain types of bugs have appeared to creep up often in their use. In fact, one such bug occurred in the train project some time before the program shown in the previous example.

In this earlier program, the programmer wanted the train to reverse directions each time it went by the light beside the track. The intention was to have the train go around the track once forward, see the light, then go around the track in reverse, then repeat, going forward and reverse for consecutive trips around the track.

The first program tried looked like this:

```
to go
  when [sensora < 20] [rd] when side-mounted sensor sees
                                     light, reverse direction
end
```

Although the program seemed simple enough -- reverse whenever the sensor sees the light beside the track -- the behavior seemed unpredictable. Although the train would hesitate at the light, it sometimes chose to continue in the same direction rather than reverse direction (and sometimes would indeed reverse direction as the programmer intended).

What in fact is going on in this case is somewhat easier to see when we dissect how the computer executes the **when** statement. The computer checks repeatedly if the condition is satisfied, and as soon as it is satisfied, the action clause is executed. As soon as the action clause is finished, the computer goes back to checking the condition over and over. What happened in this case is that, although the computer *did* reverse the train's direction on contact with the light, the computer went right back to checking for the light. Since the computer is quite fast, the train was likely still in front of the light, resulting in another reverse direction. This cycle might go on for some time, until the train finally broke free of the light.

What the programmer *meant* **when** to do in this case was to execute the action only once for each time the light was seen. What **when** actually did was execute the action repeatedly for as long as the light was seen. With some help, the programmer fixed the problem as follows:

to go

when side-mounted sensor sees light, reverse direction, and then wait for 1 second

```
when [sensora < 20] [rd wait 10]
end
```

By waiting for one second after changing the train's direction, the train would have cleared the light by the next check of **sensora**.

But this solution is unsatisfying. If the train were moving at a different speed, perhaps the amount to wait would need to be changed. The delay of an arbitrary period of time seems to be unrelated to the fundamental problem at hand.

What is really needed is a way to make explicit the difference between doing something once for a period of the condition being met, and for doing something over and over during the period of the condition being met. The explanation of **when** (and, indeed, the word itself) is ambiguous enough to seem to fit either case. The programmer doesn't realize which he means when he uses **when**.

This idea is certainly not new; it comes up in all sorts of systems where conditions are linked on actions. In the world of digital electronics, the two ways of triggering the event are referred to as "edge-sensitive" and "level-sensitive" -- meaning, sensitive to a change in the condition, or sensitive to the condition's current level.

This "edge" versus "level" sensitive bug came up in standard LEGO/Logo, before the **when** statement. Typically, a buggy program like the following:

```
to check
  waituntil [sensora < 20]           wait until sensor sees light
  rd                                 reverse direction
  check
end
```

would be fixed by adding a line to look like this:

```
to check
  waituntil [sensora < 20]           wait until sensor sees light
  rd                                 reverse direction
  waituntil [sensora >= 20]         wait until sensor sees dark
  check
end
```

But now that we have the new syntax for **when**, it seems to be worthwhile to create two separate primitives with a similar syntax: one for level-sensitive reaction to the condition, and the other for edge-sensitive reaction. This would not only simplify the solutions in cases where edge-sensitivity was needed, it would also, by making explicit the difference between edge- and level-sensitivity, encourage the programmer to think about this issue, and thus hopefully avoid having the bug.

5.4.4 De-emphasis of Complex Data Structures

Although the programmable brick requires a richer set of control flow structures than Logo previously gave, it required less in the way of data structures and symbolic processing.

Experience showed that most LEGO/Logo programs reacted to the current state of the sensors. If an old sensor value was important, its value was usually encoded into the control flow. For example, to wait for a touch sensor to be pressed and then released, one might write:

```
to touch
  listento 7
  waituntil [sensor?]           Wait until button is pressed
  waituntil [not sensor?]      Wait until button is released
end
```

Different steps in the control flow imply different sensor histories. For example, during the second `waituntil`, we know that the sensor has previously been held down.

One normally wouldn't see a program that sampled the sensor for a few seconds, storing the data, and then analyzing the data to look for a press and a release.

Both Logo languages implemented for the brick did away with the traditional Logo data structures of sentences and words. This greatly simplified the language implementation, and improved its efficiency in terms of speed and memory usage as well.

On the other hand, direct experience with the brick showed that it was often desirable to store up simple data for later viewing and analysis at the host. One such project involved graphing the temperature inside a refrigerator over time. The brick passively recorded sensor values for later upload to the host. A simple set of primitives, much less general than the list processing primitives found in standard Logo, was implemented to record a simple list of data for later upload.

Chapter 6: Experiences

The programmable brick has been used in several different situations, by different sorts of people. Many of the experiences have been with kids aged 11-18, although some experiences have been with adults.

This chapter will describe several of the situations in which programmable bricks were used. It will discuss what people have tried to do (sometimes successfully, sometimes not) with the brick. It will also try to go a little into what people might have learned while using the brick. These initial experiences provide some feedback on the design of the programmable brick, and they suggest future changes to the brick and its software environment.

6.1 Door/Light Switch

One of the earliest projects with the programmable brick involved three kids aged 11-15, all of whom had used LEGO/Logo somewhat before: two had attended a LEGO/Logo workshop at the Boston Science Museum, and one of which had worked with LEGO/Logo at his school. We ran a one-day experimental workshop for them, with a rather early version of the programmable brick.

Two of the kids were intrigued with the idea of making an "active environment" -- making the environment "come alive" and react to people. After some consideration, they decided to make a device to flip on a room's light switch when people entered the room, and flip it off when people left.

At first, they studied the different possible sensors, trying to figure out which one could be connected to the door, and how. They decided to try a "bend" sensor to sense the opening of the door. (The bend sensor is a

several-inch-long plastic whisker that gives a measure of how much it is bent.) They first tried mounting it to the wall approximately where the doorstop was, but then decided it would require people to open the door very wide to be sensed. They then tried mounting the sensor at the door hinge in such a way that the sensor was bent in proportion to how widely the door was opened.

Before programming, the two kids wanted to be able to find the value of the bend sensor at different times to find out if they had mounted it well, and if the sensor would really give them the information they wanted. We hadn't yet introduced the programming primitives, so we wrote a program with them to simply show the value of connected sensors on the LCD. (Based on this, we later decided to add a sensor display mode to the brick itself, without any user programming necessary.)

The next task the two kids undertook was to attach a LEGO motor to the light switch in such a way that when the motor turned, the switch would flip. The kids appeared to be already quite comfortable with building LEGO mechanisms, and quickly built a device with a significant gear-down and a lever to connect it to the light switch. They designed their mechanism in such a way that spinning the motor one way would turn the light on, while the reverse direction would turn the light off.

But while these two participants found making the mechanism easy, they had a bit more trouble figuring out how to mount it to the wall next to the switch. They asked for masking tape to connect their creation to the wall next to the light switch. They ended up using quite a lot of tape, making it hard to take their project back down for later debugging. "We need a quick-release system here" one said as he was trying to get the project down. (Of course, LEGO is a sort of quick release system itself, but it seemed the problem was caused by the tape not being strong enough to hold against pulling the LEGO project from its attached base). Later, we offered double-sided sticky tape to attach the project more securely to the wall, which helped quite a bit, but this had the later disadvantage that it left a mark on the wall. In several later projects as well, attaching LEGO mechanisms to actuate things in the real world has

been a difficult task, so perhaps some future thought should go into ways of improving this. (Several people have suggested that the correct solution is that the world itself is at fault here: everything in the world should be built of LEGO, or at least have LEGO-compatible attachment surfaces. Admittedly, some may find this viewpoint extreme.)

As with the sensor, the two kids wanted to try out the motorized device without writing the full program to implement their desired algorithm of switching the light switch when the door was opened. We gave them a standard LEGO battery pack for this purpose, and they made modifications to their light switcher mechanism until it mostly worked.

At this point, the kids started focusing more on the algorithm for flipping the light switch when the door opened. They realized there was a problem: the door sensor only gave the information that the door was opened. It did not tell whether people were entering or exiting the room. The kids wanted some sort of sensor to tell whether someone was entering the room, in which case their machine should turn the light on (if it wasn't already). They decided if anyone left the room, the light should be turned off.

After a little thinking, the kids came up with a clever solution: they attached a LEGO bar to the inside of the door handle that was pulled to exit the room, and connected a LEGO touch sensor to this bar. In this way, the programmable brick could tell if people were leaving (in which case the door would be opened while the touch sensor in the handle was pressed), or if people were entering (door open but no signal from the touch switch).

Once the second sensor was in place and tested, they wanted to program. Their program turned out to be pretty simple (as was found with many later programmable brick programs), and worked without too many iterations. The finished program looked similar to the following:

```
to light
  if door open and handle sensor pressed, turn off light
  if (sensora < 105) and touchb [turn-off-light]
  if door open and handle sensor not pressed, turn on light
  if (sensora < 105) and not touchb [turn-on-light]
end

to turn-off-light
  motora, thisway run motor forwards for 3 seconds
  onfor 30
end

to turn-on-light
  motora, thatway run motor backwards for 3 seconds
  onfor 30
end
```

Once the kids got the project working , they ran in and out of the room repeatedly, breaking into big smiles each time the lights switched on and off.

In the end, the light switching project worked with about 50% reliability. When they decided they were finished with the project, the primary failure modes were mechanical failure of the LEGO mechanism, and mechanical failure of the attachment between the LEGO mechanism and the light switch. (Note that the kids program did not sense the "limits" to the switcher movement and stop the motor when the switcher was done. Instead, they relied on the mechanism jamming at the far ends of its travel, which is one reason for the mechanical failures.)

6.2 Light follower/helicopter

The third participant in our experimental workshop worked alone, and decided he wanted to build a helicopter with wheels that drove around, bumped into things, and spun its helicopter blade at various times.

We already had a motorized, wheeled LEGO base, which we offered for his use. He then put the brick on top and added the mechanism for the helicopter blade.

After finishing most of the structure, the participant looked at the various sensors available to him, and decided he wanted to make something that reacted to light instead of bumping into things. We suggested a few possibilities, and the participant decided he wanted to try to home in on a flashlight. He placed two light-measuring sensors on the front of his device, one to measure light coming from the left side, and the other to measure light coming from the right.

In this case, the program was written without first testing the sensors or the actuators (with the exception of the LEGO wheeled base, which was known to work). An early version of the program tested each light sensor to see if each saw brightness over a certain threshold. If neither sensor saw brightness over the threshold, the helicopter wouldn't move. If one saw brightness over the threshold, the helicopter would steer towards that light. If both saw enough brightness, the helicopter would move straight ahead, spin its propeller, and play tones on the speaker. The program looked similar to the following:

sensor a and sensor b are connected to light sensors on the left and right, respectively. Light sensors return lower values as they see brighter lights.

motor d is connected to the rotor.

```
to helicopter
  if it's brighter to the left, turn left
  if (sensora < 20) and (sensorb => 20)
    [turn-left motord, off]
  if it's brighter to the right, turn right
  if (sensora => 20) and (sensorb < 20)
    [turn-right motord, off]
  if both are bright, go straight, spin the rotor, and beep
  if (sensora < 20) and (sensorb < 20)
    [go-forward motord, on beep]
  if neither is bright, stop
  if (sensora => 20) and (sensorb => 20) [alloff]
end
```

motors a and b are connected to the left and right wheel sets. steering is accomplished as in a wheelchair or tank: if both motor a and b go forward, the helicopter goes forward. if motora and motorb go at different speeds, the helicopter will turn.

```
to go-forward
  motora, on
  motorb, on
end
```

```
to turn-right
  motora, on
  motorb, off
end
```

```
to turn-left
  motora, off
  motorb, on
end
```

At first, mechanical fixes were required to make the rotor work, but then the program debugging started. It turned out that the helicopter didn't work very well: it wouldn't come after a flashlight unless it was extremely close. After changing the brightness threshold in the program to be more sensitive, the program had a different bug: now, when the helicopter came within a few feet of the flashlight, it would stop correcting its course and just go straight.

The participant continued adjusting the threshold values, but didn't get performance that was much better. After a while, we suggested a possible fix: to compare the sensors to each other, rather than each to a threshold. The resulting program seeked the flashlight more reliably. The **helicopter** procedure now looked like the following:

```
to helicopter
  if (sensora < sensorb) if it's brighter to the left, turn left
    [turn-left motord, off]
  if (sensora > sensorb) if it's brighter to the right, turn right
    [turn-right motord, off]
  if (sensora = sensorb) if both are the same, go straight
    [go-forward]
  if (sensora < 20) and (sensorb < 20) if both are bright, turn on the rotor and beep
    [go-forward motord, on beep]
end
```

The participant was quite pleased with his creation. When asked how he would compare his experience with the programmable brick to his earlier experiences with LEGO/Logo, he replied:

It's nice when there are no wires and stuff... Otherwise I can't just move it [the helicopter] in here... It's hard to move with the wires on and it's hard to turn and everything.

6.3 Artificial Creatures Workshop

The second workshop for kids using the programmable brick took place over a 4-day period at the Boston Museum of Science. The five participants had three hours of workshop time per day, for a total of a 12 hours. Ages ranged from 12 to 16. Although students had varying amounts of previous experience with LEGO and programming (three hadn't programmed before at all), all were able to make a working programmable "creature" by the end of the class.

One focus of this workshop was the use of multiple processes for multiple behaviors. Different simple programs the participant wrote, such as "follow light" or "follow wall," could be run as different processes, and the user could turn these individual programs on and off using the brick's screen, knob, and buttons. Additionally, primitives were provided to allow students to turn on and off the different processes from program control. Thus, processes had the ability to start or stop other processes.

The three participants who had not programmed before made creatures that used relatively simple software. One made a creature that followed a line (a piece of tape laid down on the floor). Another made a creature that simply backed up when it hit an obstacle. The third made a creature that backed away from bright light (this participant had lots of fun playing with his creature using a flashlight).

The other two participants, both of whom had programmed before, managed to progress to the use of multiple, interacting behaviors. (The issue appeared to be a simple lack of time for the others, who had taken extra time to get up to speed learning to program. In evaluations written

by the students at the end of the class, they unanimously replied that the workshop had been too short.)

One of the projects with multiple behaviors, a combination light-follower and obstacle avoider, has been previously described in section 2.3.2. To recap, the participant first made the distinct light-following and obstacle-avoiding behaviors, and then wanted to make them both run at the same time. But when both behaviors operated, there were conflicts. The participant modified the obstacle-avoidance behavior to temporarily turn off the light-following behavior while the obstacle behavior was navigating around a detected obstacle.

The other project using multiple behaviors combined an attempt at navigating a path between several rooms at the museum using timing only (no sensor feedback), with a lower-level behavior to try to move around obstacles it inadvertently bumped into (such as chair and table legs).

The robot started off with a simple path it tried to follow: go forward for 20 seconds, turn left, go forward for 15 seconds, turn left, go forward for 20 seconds. This path was intended to make the robot leave the classroom, go down the hall, make a left into a different classroom, and turn left again to try to make it out the other classroom's back door. But this simple behavior didn't have much of a chance at working: the second classroom was full of tables and chairs, and the robot invariably hit one or two and got stuck. Sometimes, also, the timing of the path was a little off, or the robot drifted off its planned path, and ran into a wall unexpectedly. For this creature, running into a wall, chair, or table typically meant getting stuck and progressing no further.

To try to deal with this problem, the participant added a behavior: when the creature ran into an obstacle, it attempted to pilot around the obstacle and end up with roughly the same heading as it had in the first place. The behavior looked something like this:

```
to avoid-obstacles
  when the bump sensor is pressed, try to steer around the obstacle.
  if toucha [ spin-right
              wait 10
              go-forward
              wait 10
              spin-left
              wait 10
              go-forward]
end
```

The added behavior did not completely solve the navigation problem -- the robot did not navigate its course reliably. However, the robot typically got much further than without the behavior to get the robot "unstuck."

6.4 Robot Challenge Workshop

The "robot challenge" workshop also took place at the Boston Museum of Science³. It, like the Artificial Creatures workshop, took 12 hours over a 4 day period, and attracted ages ranging from around 12 to 16.

For this workshop, students created a mobile robot to compete in a contest on the last day. The contest goal was to navigate as fast as possible the playing field shown in Figure 6.1.

³ This workshop was run by this author and Fred Martin.

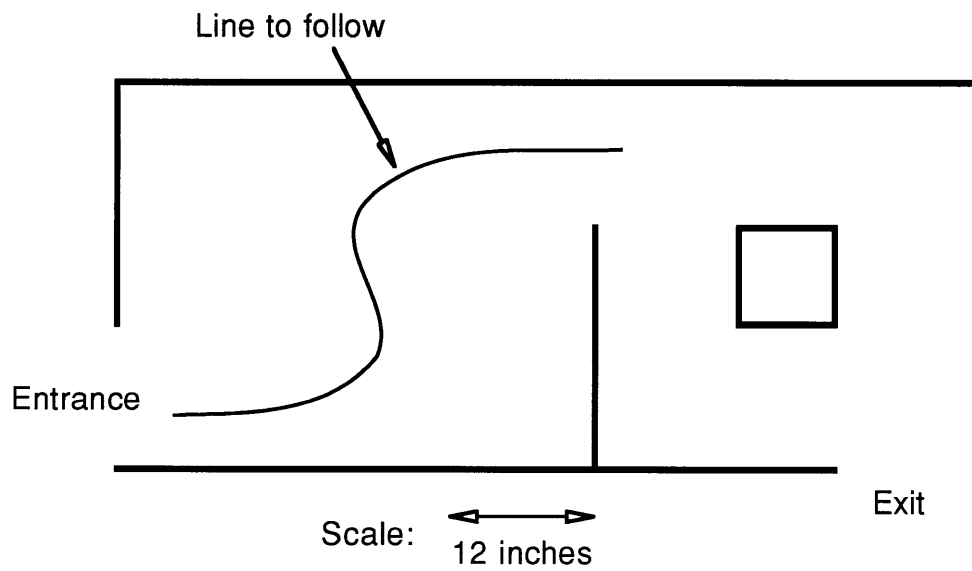


Figure 6.1
Robot Challenge Playing Field

This workshop had a much more structured goal than did the Artificial Creature workshop. It required a certain minimum level of performance of the robots in order to successfully complete the course. The high structure, high expectations, and limited time available unfortunately seemed to turn off some of the students. As a result, a second category was created for manually controlled entries for those who were unable to or did not wish to complete an automatic entry. Six of the twelve entrants created autonomous robots, while the other six controlled their entries manually.

A variety of different strategies were attempted by the autonomous entries. All but one used sensing to follow the line through the first half of the course, and to try to follow walls in the second half to find the exit. One entrant attempted to perform the course completely without sensing -- relying on timing only. While this entry was very fast, it was also very unreliable (although it came close to finishing sometimes, it never actually did).

In previous LEGO activities (either with LEGO/Logo, or the MIT LEGO Robot Competition), participants have often used similar "open loop"

strategies: trying to do precise things without feedback. These attempts are characterized by constantly tweaking parameters, trying to zero in on the precise amount of time a motor should be turned on, etc.

Unfortunately, on different runs, wheels may slip in different ways, or the batteries might be at different levels of charge, or any of a number of other things could be different as well. Failed or partially failed attempts at precision without feedback almost always prove to be instructive, both to the person who attempted it and also to the people watching and seeing the problems. People learn to appreciate the imprecision and uncertainty that often characterizes the physical world. And people learn that by sensing error and correcting, imprecise parts can be combined to make a precisely acting whole.

The hardest problem for people in this particular course -- the part that caused the most robots to fail -- was the transition from following the line to following walls. Many robots which sensed the line visually would try to steer to find the line if the robot strayed from the line. Unfortunately it is difficult to determine whether the robot has lost the line by veering off of it, or whether the end of the line has been reached. A common solution to this problem was to start following walls as soon as a bump sensor was triggered, but this did not always work. For example, some robots would commonly, in searching for the lost line, manage to make a u-turn and find the line again, but this time going backwards and find themselves back at the entrance.

The robot challenge course shared an aspect with the artificial creatures workshop: students were encouraged to use multiple behaviors. Most people who wrote programs to successfully navigate the challenge course used two behaviors: one to follow the line, and the other to try to get out of the maze at the end (possibly by following a wall). However, unlike the artificial creatures workshop, the structure of the robot challenge was such that having multiple behaviors running at the same time was not beneficial. Here, the main challenge wasn't in how behaviors interacted, but instead determining when to switch between two mutually exclusive behaviors.

6.5 LEGO Train

As mentioned in section 2.1, the programmable brick was brought to The Computer Museum in Boston. Specifically the brick was used in a special center for kids at the museum called the Computer Clubhouse.

As explained previously, this author had intended to work with kids at the clubhouse in making programmable musical instruments out of programmable bricks. But the kids weren't interested: instead, they wanted to play with the LEGO train on the other side of the room. The brick was of course flexible enough to be attached to the train, so the kids became interested in programming the train once it was hooked up to the brick.

Kids started off by writing simple programs for the train, such as switching direction every once in a while. Some wanted to be able to put a manually controlled train on the track as well, though. By a few simple modifications to the train on which the brick was mounted (to disconnect its electronics from the metal track), a manual train powered by the track could be run independently of the brick train, even on the same track.

Sections 5.4.2 and 5.4.3 discuss in some detail the program that one kid wrote for the train once the manual train had been put in place. This program made the train interact in various ways with light markers placed beside the track, as well as a light marker placed on the manually controlled train. Several kids enjoyed playing "tag" with the brick-controlled train by controlling the manual train to come close to the automatic train, causing the automatic train to stop or change direction.

As described in section 5.4.3, this project showed the desirability of distinguishing between looking for sensor events on the basis of "edge" versus "level" sensitivity.

6.6 Bike Trip

This section discusses a short project done with the brick by Brian Silverman, and his son, Eric. Brian Silverman was involved with the design of the brick, and was earlier mentioned as the author of one of the two programmable brick software environments.

Brian wished to graph the speed of his bicycle over time during his daily bike trips commuting to and from work. He decided to use a programmable brick, attached to the handlebar of the bike, to collect the data. After playing with various sensors to measure the rotation of the front wheel, Brian and Eric settled on a magnet mounted to the wheel, and a reed relay (mechanical magnetic sensor) to be mounted next to the wheel. Once per rotation of the front wheel, the sensor would see the magnet. The brick was then programmed to record, every two seconds, both the speed of the wheel, and the total distance traveled.

In order to graph the results, Brian and Eric wrote a special-purpose program in the host Logo. Although his recorded data measured speed versus time, they wanted to instead graph speed versus distance traveled. This is a case where having the host Logo really paid off -- if some standard graphing tools had been available in the brick environment, or if Brian and Eric had simply exported the data to a spreadsheet, these tools likely would have been able only to graph the speed versus time.

By graphing the bike's speed versus distance, they could superimpose trips taken on different days. Events at the same location on each trip (like traffic lights or stop signs) would be at the same place along the X-axis of the graph. The graph is quite striking: it shows things like train tracks, where Brian had to slow down every day, and traffic lights, where he came to a stop only some of the time. (See Figure 6.2 for the graph of a single trip to work and back).

By plotting his return trips in reverse, and superimposing them, many of the features (such as slowing down for the train tracks) were in common.

But consistent discrepancies (one stretch of the trip consistently being faster one direction than the other) indicated something was different between the trips to and from: consistent differences in speed indicated an uphill or downhill slope.

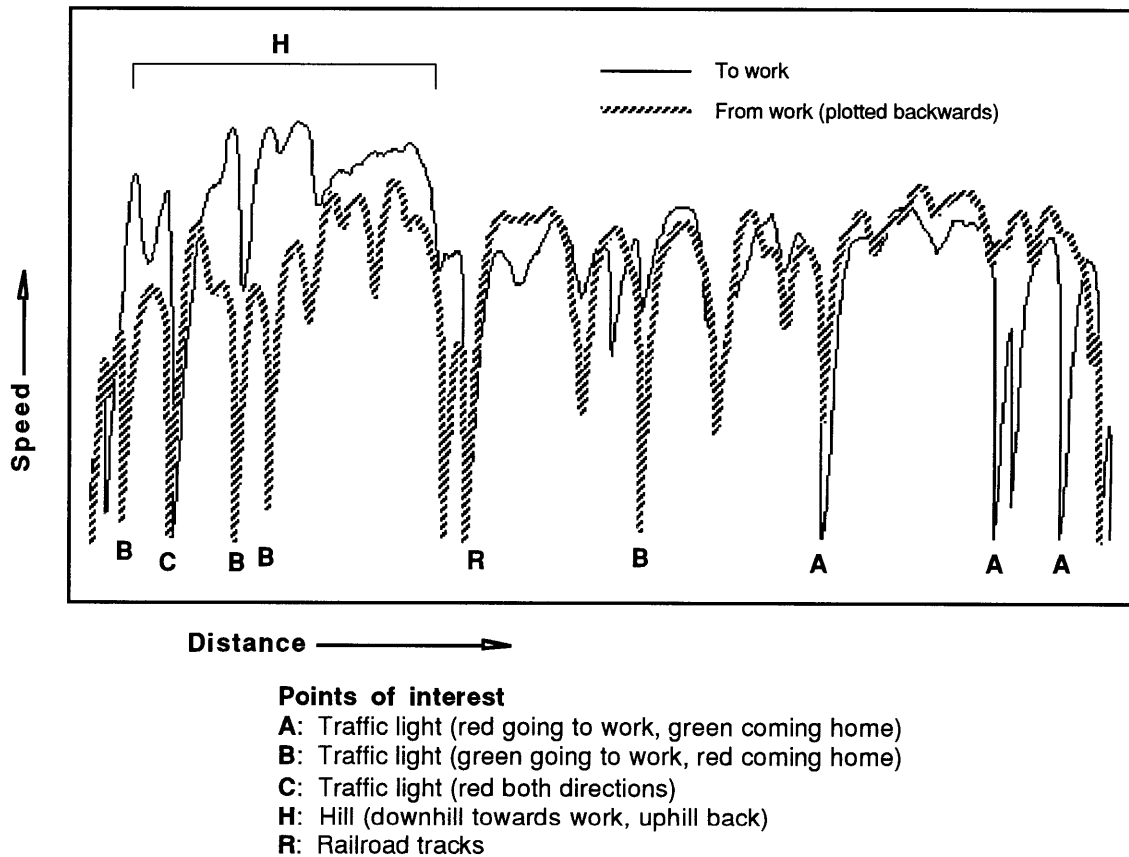


Figure 6.2
 Speed versus distance riding bike to and from work

6.7 Conclusions and the need for further study

One possible failing in the observations with people using the programmable brick is that they don't go into enough detail. There are no micro-analyses tracking how individuals engage new concepts. The observations so far do not provide a clear indicator of the difference between the learning going on using the programmable brick versus its predecessors, such as LEGO/Logo or even straight Logo programming.

On the other hand, there are some important conclusions to be drawn already. The types of projects described in this chapter are fairly different from those typically explored in LEGO/Logo (and in many cases are impossible to explore with LEGO/Logo). These observations indicate the brick project has met its goal of providing a wider range of learning opportunities than previous environments.

One big application of the programmable brick so far seems to be for portable, moving LEGO robots or creatures. This was one of the more obvious uses anticipated, as similar things had been attempted with LEGO/Logo but had always been thwarted by the tangling of the wires leading to the desktop computer.

One possibly new content area uncovered by the experiences so far is that the brick can support spontaneous scientific inquiry and experiments, such as graphing the speed of the bicycle. The brick's portability, and ability to measure and record many different sorts of sensor input, makes it easy to ask questions of the environment, and try to find answers.

More detailed observation of people using programmable bricks is needed. More exploration into different activities possible with the bricks is needed as well. I hope that work with the bricks continues to be active.

Chapter 7: More Activities

While children have used the brick in several different activities, some additional activities have been designed, and not yet tried. This chapter discusses two such activities, which hopefully might inspire activities to be tried with kids in the future.

7.1 Computer Clubhouse Wharf

The Computer Museum in Boston, one of the sites at which the programmable brick has been used, sits atop a wharf on the Boston Harbor. One activity, suggested by staff members of the computer museum and the children's museum next door, is to create activities around exploration of the wharf environment.

There are many things children can explore in the wharf environment. Hopefully, kids can think of their own questions to ask about the environment, which will lead them to explorations of their own design. They will learn things about ecology, and about computation, and the connections that can be made between them.

In order to explore different questions about the wharf environment, kids can make special-purpose measuring devices using the programmable brick. The brick can go places people can't go, and it can measure things that people can't easily quantify (such as light level or temperature). (Note that many explorations might involve using the programmable brick as a submersible probe, which would require some sort of waterproof case.)

Here is a list of some possible explorations kids might think of doing in the wharf environment:

- How cold is the water? Does the temperature change with depth? Does it change with time of day? Does it follow the temperature above water?
- Are there fish in the water? Use sonar to repeatedly try to sense nearby objects, and see if the nearby objects change over time. Are the fish attracted by certain noises? Repelled by others? Do they make noise? Are they more common at different times of day? What kinds of bait bring fish close by?
- Teleoperated videocamera. A somewhat sophisticated project might be to put some propellers and a videocamera on the programmable brick (with a cable leading back to the surface). Kids could drive around the water (raising and lowering with the cable, changing orientation with the propellers), exploring the bottom, and nooks and crannies around the wharf.
- How dirty is the water? How much sunlight can be sensed at different depths? How much do different depths block light? (measure the amount of light transmitted from a submersed light bulb to a sensor a few inches away) Are there pollutants in the water? (would require special sensors)
- How fast does the water flow? Is it different at different depths? Which direction does it go? Does it change at different times of day?
- How does the water environment change with the weather? Does a storm make the water murkier? In what ways do the seasons affect the water?
- Can you hear ships underwater?

- What kind of stuff is at the bottom? Using LEGO, kids can build a mechanical scoop that will grab a sample from the harbor floor. Using appropriate sensors, kids may be able to locate metal objects on the bottom for retrieval.

This list of activities is not meant to put forward a list of activities that would be universally enjoyed by kids. Rather, it is intended to indicate the breadth of activities possible, and some of the interesting things some of the activities might lead to. The average brick user might not be intrigued by the average item on the list, but hopefully, the flexibility of the brick will encourage a potential brick user to think of his own activity, one for which he will be involved and motivated.

7.2 Computerized Musical Instruments

Using the audio input and output features of the programmable brick, kids can make their own musical instruments. Not only can kids design what form of input an instrument might take, but they can also program in their own personal musical effects.

Through working with these activities, kids could learn about music theory, and about computation, and how the two can be hooked together.

Here are the sorts of things kids might try with their musical instruments:

- Make an instrument with 3 buttons, like a trumpet. How do the buttons affect which note is played in a real trumpet? What are alternate ways it could work? How is the note on a real trumpet affected by how you use your lips? What are other ways the instrument could get input for a similar purpose?

- Make an instrument with a sliding input, like a trombone. How does the position of the slide affect the note in a real trombone? What are alternate ways it could work? Can software be written to make a novice sound nicer?
- Make an instrument be able to record songs and play them back.
- Make an instrument be able to play a song in “rounds.” (This could be done with several programmable bricks communicating with one another).
- Make an instrument respond to the playing of a certain note combination. Perhaps it answers with more of the song, or perhaps could play a duet with the user.
- Make an instrument respond to the microphone input. A pre-recorded song could be played back at a tempo set by a drummer. One of ten pre-recorded songs could be played back by whistling the first few notes (the audio in can do very rudimentary pitch detection). How can note patterns be recognized even when they are in a different key? What changes about the frequencies?
- Make an instrument try to play chords (based on the single notes being input) that sound “nice.” What are some rules to follow? This could be a good way to start out into some music theory.
- Make an instrument that makes up its own song, or improvises based on some input from the user. What happens if you have two programmable bricks, each trying to improvise on what the other is playing?
- Make an instrument that makes sound based on things sensed in the environment. Make sound based on the reading of a light sensor and then put the sensor in front of a TV.

- Make an instrument with a completely non-standard means of input. A kid could put sensors all over his body and then dance to create music.

Again, as with the list of activities located at the Computer Museum Wharf, this list of musical activities is intended to show the breadth of possible applications. Hopefully each kid to use the brick would come up with his own special project.

Chapter 8: Conclusions and Future Directions

When I started the programmable brick project, I really had no idea how much work I was getting myself into. I was very surprised how much more work the brick hardware was than the LEGO Robot Design Contest hardware. In retrospect, I realize that trying to fit twice the functionality into less than half the size was perhaps a little overoptimistic.

In addition to the complex physical packaging and interconnections problems, I ran into difficult problems with procuring necessary parts. Unlike the LEGO Robot Design Contest hardware, the parts necessary to miniaturize the programmable brick are those typically found only in mass-produced consumer electronics. Many of these parts are difficult or virtually impossible to obtain in quantities less than a few thousand. A particular problem was the small LCD screen: I am indebted to Seiko Instruments Incorporated for allowing us to purchase, in small quantities, displays normally only manufactured for their hand-held language translators.

But I think that despite the difficulties with producing the programmable brick, the brick does appear to have achieved its design goals: it is a pocket sized computer, capable of running Logo and interfacing in a wide variety of ways to the real world. It provides new and engaging learning activities for kids. It seems that the brick makes good on its promise of a new breadth of activities.

Several of these new areas of application for the programmable brick were explored in Chapter 6. One big application is the use of the programmable brick for portable, moving LEGO creatures or robots. Another application explored was the use of the programmable brick to embed computing in the environment, to make the environment "come alive" (as with the example of the automatic light switch). The last

application area that seems to have come from use of the brick is "impromptu science experiments": the brick's ability to measure and record sensor input, along with its portability, makes it easy to ask questions of the environment, and try to find answers.

But the experience gained from using the brick is in many ways incomplete: big things such as inter-brick communication still haven't been tried. Much work still lies ahead in terms of trying different projects with the brick.

The experience gained from use of the programmable brick points to more than its breadth of use. The feedback points to some new activities to try with the brick, and some ways in which we should try to change the brick, such as the addition of "edge" vs. "level" sensitivity for sensors.

This thesis unfortunately does not show striking educational conclusions resulting from watching people using the brick. I hope that future work may delve more deeply into people using the brick in various new projects, and try to uncover more about what they are learning and how they are learning it. I believe the observation thus far does show that the brick provides a rich constructionist learning environment, so I hope the reader is still convinced of the bricks' educational importance.

Perhaps the most important success to me is that kids have fun using the brick. The kids' smiles as the light-switcher worked, a kid's excitement at having his robot successfully complete the Robot Challenge course: these made all the work worth it. I recognize in many of the kids I've worked with the same enjoyment and intensity of involvement I had when I was young and was first introduced to computers. I have achieved my personal goal: providing kids with a rich learning environment that they enjoy and voluntarily choose to become engaged with.

Bibliography

- Brooks R. (1989) "A Robot that Walks: Emergent Behavior from a Carefully Evolved Network", *Neural Computation* 1:2, 1989, pp. 253-262.
- Bourgoin, M. (1990) Children using LEGO Robots to Explore Dynamics. Master's Thesis, MIT Media Laboratory.
- Hogg, D., F. Martin, M. Resnick (1991). Braitenberg Creatures. E&L Memo No. 13. MIT Media Laboratory.
- Martin, F. (1988) Children, Cybernetics, and Programmable Turtles. Master's Thesis, MIT Department of Mechanical Engineering.
- Martin, F., and R. Sargent (1992) Learning Engineering Through Robotic Design. *Proceedings of the Ninth International Conference on Technology and Education*, edited by Nolan Estes and Michael Thomas. March 1992.
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books. New York.
- Resnick, M., and R. Sargent (1994). "Programmable Bricks: Ubiquitous Computing for Kids," in *Constructionism in Practice*, edited by Y. Kafai and M. Resnick. MIT Media Laboratory. Cambridge, MA.
- Resnick, M., S. Ocko, and S. Papert (1988). LEGO, Logo, and Design. *Children's Environments Quarterly*, vol. 5, no. 4.
- Resnick, M. (1993). "Behavior Construction Kits." *Communications of the ACM*, vol. 36, no. 7, pp. 64-71.

Resnick, M. (1992). "Beyond the Centralized Mindset: Explorations in Massively Parallel Microworlds." PhD dissertation. Massachusetts Institute of Technology.

Resnick, M. (in press). "Towards a Practice of 'Constructional Design'" in *The Contributions of Instructional Innovation to Understanding Learning*, edited by Bob Glaser and Leona Schauble. National Research Center on Student Learning. Pittsburgh.

Resnick, M., S. Ocko (1990). LEGO/Logo: Learning Through and About Design. E&L Memo No. 8. MIT Media Laboratory.

Wellner, P, W. Mackay, R. Gold, Editors (1993), Computer-Augmented Environments: Back to the Real World. *Communications of the ACM*, July, 1993, pp. 24-97.

Weiser, M. (1991) The computer for the twenty-first century. *Scientific American*. (Sept 1991), pp 94-104.

Appendix I: Twenty Things To Do with a Programmable Brick

The document "Twenty Things to do with a Programmable Brick," compiled by this author and Mitchel Resnick, is included here to present some of the breadth of possible applications conceived for the programmable brick.

1. Build it into a mobile robot, and pretend it's a new life form exploring its habitat.
2. Take it with you to gather sensor data. For example, measure the pH of the water at various places along a local stream, or measure the noise level at various places around a construction site.
3. Connect it to your body and gather data about your heartbeat, etc. as you run.
4. Connect it to your telephone to keep track of how often your telephone rings.
5. Attach it to your door frame (with a light sensor) to keep track of how many people walk through the door. Or program it to greet people as they walk through the door (with music or digitized speech).
6. Put it on the roof of the building to gather weather information.
7. Put a bunch of them in a room and program them to make sounds in response to sound that they "hear," so that they act (somewhat) like birds in the jungle.

8. Use it to program your VCR (using IR communication to the VCR much like a remote control would)
9. Use it like a standard LEGO/Logo interface box.
10. Control a videocamera.

As the brick can control devices controllable with infrared remote controllers, the brick can talk to many of the newer videocameras. The brick can tell the camera to record when certain events take place (like taking anyone's picture for a few seconds when he or she walks into a room). One can record things that would otherwise require much patience (make a setup to tape a picture of the mouse you suspect comes out at night). One can make crude time-lapse videos by recording a second every so often. The brick can point the camera -- one could program a sequence of camera movements for a certain shot. One could even rock or shake the camera for special effects.

11. Make it into a programmable musical instrument.

Build the instrument with LEGO (does it have buttons like a flute, or a sliding part like a trombone, or ... ?) Program the instrument to augment or improvise on your notes, or to simply play another part. Program the instrument to play a second copy of your notes with a certain delay to play in rounds.

12. Find out if the light really does go off when you shut the refrigerator door.
13. Send secret messages across a room to someone else with a brick.
14. Make games where the brick can play an active role.

The games could be "sit down" games (like board games or card games). The games could be "recess" games (like tag or hide-and-

seek), where each person could be running around with one of the bricks.

15. Use many bricks to haunt a haunted house.

Attach one to the door to make creaking noises when the door is moved. Drop spiders on visitors who trigger a light beam. If someone screams, repeatedly play back the scream in several voices to add to the panic. Point a spotlight at a brick animated toy monster to cast spooky shadows on the wall. Put a brick and wheels on the pumpkin to allow it to move, and put LEGO lights inside so the pumpkin can sneak up on people when it's "off", and make a noise and turn on the lights at the same time when it gets close to someone.

16. Control lights and appliances.

If one can create the proper LEGO mechanics, one can program the brick to turn on and off lights or appliances by physically moving the power switch. Turn on the lights in a room when someone opens a door. Turn up the heat

17. Water plants.

Make your programmable brick water your plants every few days.

18. Make sound effects.

Make your brick echo what it hears to make you sound like an announcer in a large stadium. Make it play songs with a sampled sound, much as a sampling audio keyboard can. Learn the tones needed to mimic touch-tone phones, and then dial numbers automatically by placing the brick next to a phone mouthpiece. Record your dog's bark into your LEGO remote-control car to play to curious cats in the neighborhood.

19. Put a brick on your pet's collar to record aspects of your pet's daily life.

What temperature does your pet like? Does your pet spend much time running around? Perhaps there are patterns to your pet's activity. With a "GPS" brick (global positioning satellite system, which can tell location on the earth), find out where your pet roams. Get into discussions about whether experimenting on your pet is ethical.

20. Come up with 20 more uses for a Programmable Brick. (The obligatory recursive call.)