Tangible Programming Bricks:
An approach to making programming accessible to everyone

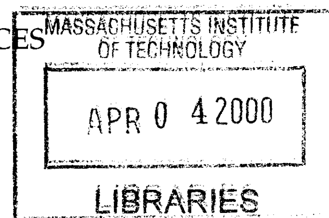
by

Timothy Scott McNerney

B.S. Music and Computer Science, June 1983
Union College, Schenectady, NY

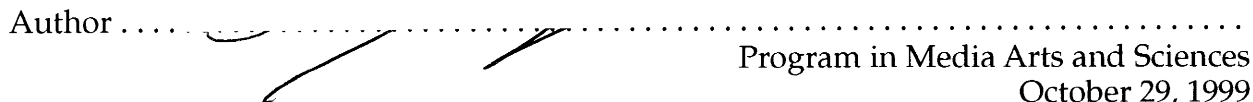
Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

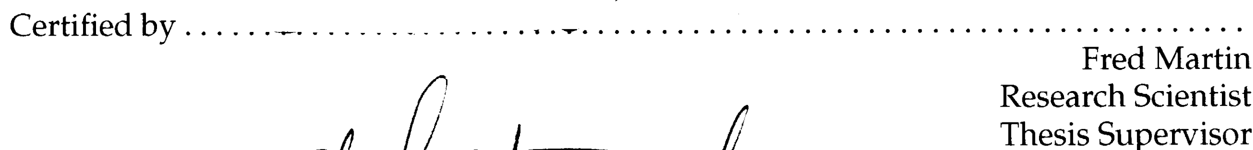
MASTER OF SCIENCE IN MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology

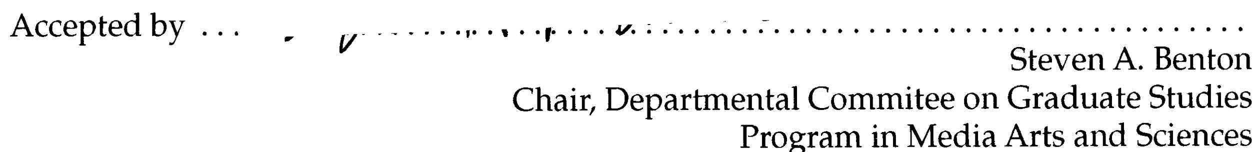

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
APR 0 4 2000
LIBRARIES

February, 2000

© Massachusetts Institute of Technology, 1999. All rights reserved.


Author ....
Program in Media Arts and Sciences
October 29, 1999


Certified by ....
Fred Martin
Research Scientist
Thesis Supervisor


Accepted by ....
Steven A. Benton
Chair, Departmental Commitee on Graduate Studies
Program in Media Arts and Sciences

Tangible Programming Bricks:
An approach to making programming accessible to everyone

by

Timothy Scott McNerney

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

Abstract

Thanks to inexpensive microprocessors, consumer electronics are getting more powerful. They offer us greater control over our environment, but in a sense they are getting too powerful for their own good. A programmable thermostat can make my home more comfortable *and* save energy, but only if I successfully program it to match my life-style. Graphical, direct manipulation user interfaces are step in the direction of making devices easier to program, but it is still easier to manipulate physical objects in the real world than it is to interact with virtual objects "inside" a computer display. Tangible, or graspable user interfaces help bridge the gap between the virtual world and the physical world by allowing us to manipulate digital information directly with our hands. Tangible Programming Bricks are physical building blocks for constructing simple programs. In this thesis I provide technical details of the Bricks themselves, demonstrate that they are useful for controlling a variety of digital "everyday objects," from toy cars to kitchen appliances, and set the stage for future research that will more rigorously support my hypothesis that tangible programming is easier to understand, remember, explain to others, and perform in social settings, when compared to traditional programming mechanisms.

Thesis Advisor: Fred Martin, Research Scientist

Tangible Programming Bricks:
An approach to making programming accessible to everyone

by

Timothy Scott McNerney

Thesis Committee

Thesis Advisor ....
Fred Martin
Research Scientist
MIT Media Laboratory

Thesis Reader ....
Hal Abelson
Class of 1922 Professor of Electrical Engineering and Computer Science
MIT Laboratory for Computer Science

Thesis Reader ....
Hiroshi Ishii
Associate Professor of Media Arts and Sciences
Fukutake Career Development Chair
MIT Media Laboratory

Thesis Reader ....
Mitchel Resnick
Lego Papert Associate Professor of Media Arts and Sciences
MIT Media Laboratory

Acknowledgements

It's hard to know where to begin and where to end when it comes to acknowledgements. This project couldn't have been done without the support of my friends and colleagues at the MIT Media Lab, the insightful suggestions from my advisor and readers, and the generosity of LEGO, who manufactured the indispensible plastic casings that made the Tangible Programming Brick possible. Last-but-not-least, I offer a special acknowledgment to my wife Eve, who gave birth to our beautiful twins, Miriam and Noah, right in the middle of my efforts to finish this thesis.

My advisor

Fred Martin

Thesis Readers

Hal Abelson, Mitch Resnick, and Hiroshi Ishii

MIT Media Lab

Rick Borovoy, Rich Fletcher, Neil Gershenfeld, Chris Hancock, Jofish Kaye, Bernard Krischer, Bakhtiar Mikhak, Maggie Orth, Rehmi Post, Matt Reynolds, Brian Silverman, Carolyn Stoeber, and Brygg Ullmer

LEGO

Erik Hansen, Camilla Holten, Torben Svingholm Jensen, and Gregg Landry

Dragon Systems

Jonathan Young

My Wife

Eve Leeman

Table of Contents

Chapter 1. Introduction 9

1.1 The promise of programming "for the rest of us" 9

1.1.1 Programming is not just for professionals 10

1.1.2 Kids as programmers 11

1.1.3 User interfaces of digital "everyday objects" 12

1.1.4 Information appliances 13

1.2 Overview 13

1.3 Organization of this document 15

Chapter 2. Motivation and Context 16

2.1 Scope: What is programming? 16

2.2 What is tangible programming? 17

2.3 Why tangible programming? 17

2.3.1 Collaborative Programming 18

2.3.2 Gender differences 19

2.3.3 Debugging 19

2.3.4 Limitations of "visual" programming languages 19

2.4 Related Work 20

2.4.1 Tangible User Interfaces 21

2.4.2 Direct Manipulation User Interfaces 21

2.4.3 Visual Programming Languages 22

2.4.4 Logo and its descendants 23

2.4.5 TORTIS button box and slot machine 23

2.4.6 Logo Blocks 24

2.4.7 LEGO Mindstorms 25

2.4.8 Other languages for children 26

2.4.9 Ubiquitous computing 27

2.5 Hardware History 27

2.5.1 Programmable Bricks 28

2.5.2 The Handy Board 28

2.5.3 Crickets and Thinking Tags 29

2.5.4 Tiles 30

2.6 Setting the stage 31

2.6.1 The tangible programming challenge 31

2.6.2 Magnetic Programming Kit and microTiles 31

2.6.3 Ladybugs 32

2.6.4 Blocks 33

2.6.5 Programmable toy trains 33

2.7 Tangible Syntax 34

2.8 Programming Styles 36

2.8.1 Imperative programming 36

2.8.2 Functional programming 36

2.8.3 Rule-based programming 37

2.8.4 Behavior mixing with priorities 38

2.8.5 Database queries 38

Chapter 3. The Implementation 40

3.1 The Tangible Programming Brick Hardware 40

3.1.1 The early 4x2 design 40

3.1.2 The 6x2 Brick design 42

3.1.3 The Cards 44

3.1.4 The Evolution of Intra-Stack Communication 46

3.2 The Tangible Programming Brick Software 47
3.2.1 Firmware 47
3.2.2 Application code 48
3.3 The Tangible Programming Brick Demonstrations 49
3.3.1 Dance Craze Buggies 49
3.3.2 Kitchen appliances 50
3.3.3 Toy trains (revisited) 50

Chapter 4. Discussion 52

4.1 Design Issues 52
4.1.1 Communication issues 52
4.1.2 Power 53
4.1.3 Connectors 53
4.1.4 Signaling and power: To multiplex or not? 54
4.1.5 Optical issues 54
4.2 A Design Critique 55
4.3 User testing 55

Chapter 5. Future Work & Conclusion 57

5.1 The near term 57
5.1.1 Formal user testing 57
5.1.2 Expanding beyond linear stacks 57
5.1.3 Issues of size and shape 57
5.2 Alternate implementation mechanisms 58
5.2.1 Bar Codes 58
5.2.2 RFID tags 59
5.2.3 Specialized work surfaces 59
5.2.4 Reusable/Re-printable blocks 60

5.3 Applications 61

5.3.1 Music synthesizers 61

5.3.2 Robotic vehicles and "spatial" programming 61

5.3.3 Tangible interfaces for the visually impaired 62

5.4 Conclusion 63

Bibliography 65

Appendix A. Schematics and Drawings 69

A.1 Schematics 69

A.1.1 6x2 Brick – Top board 69

A.1.2 6x2 Brick – Bottom Board 70

A.1.3 IR/Bus Card 70

A.1.4 EEPROM Card 71

A.1.5 Display Card 71

A.2 Engineering Drawings for LEGO Plastic Casing 71

A.3 Proof Drawings faxed back to me by LEGO 75

Appendix B. Software 76

B.1 Kitchen Brick (Logo) 76

B.2 Microwave oven controller (Logo) 77

B.3 Device Driver for Timekeeper bus device (Logo) 80

B.4 Cricket Logo interpreter/OS firmware (Assembly) 82

Chapter 1 — Introduction

> I began to see how children who had learned to program computers could use very concrete computer models to think about thinking and to learn about learning and in doing so, enhance their power as psychologists and as epistemologists. For example, many children are held back in their learning because they have a model of learning in which you have either "got it" or "got it wrong." But when you learn to program a computer you almost never get it right the first time. Learning to be a master programmer is learning to become highly skilled at isolating and correcting "bugs," the parts that keep the program from working. The question to ask about the program is not whether it is right or wrong, but if it is fixable. If this way of looking at intellectual products were generalized to how the larger culture thinks about knowledge acquisition, we all might be less intimidated by our fears of "being wrong." This potential influence of the computer on changing our notion of a black and white version of our successes and failures is an example of using the computer as an "object-to-think-with." [38, p.23]
>
> Seymour Papert, *Mindstorms*

1.1 The promise of programming "for the rest of us"

> If a machine is to serve one very specialized role, such as providing mechanical power, it ought to be hidden and of no concern. But if the purpose of the machine is flexibility and personal adaptibility, we had better figure out how to give users maximum control. Expressive power and nuance are incompatible with invisibility and inaccessibility. [9]

Graphical User Interfaces (GUIs) designed for programming are still in their infancy, so most professional programmers use text-based programming languages and editing tools for their work. For the novice programmer, text-based languages can be daunting, fortunately visual programming languages "for the rest of us" have enjoyed some success.[1] The premise of this thesis is that constructing and modifying programs using even the most modern GUIs is an unnecessary obstacle to programming. We can do better.

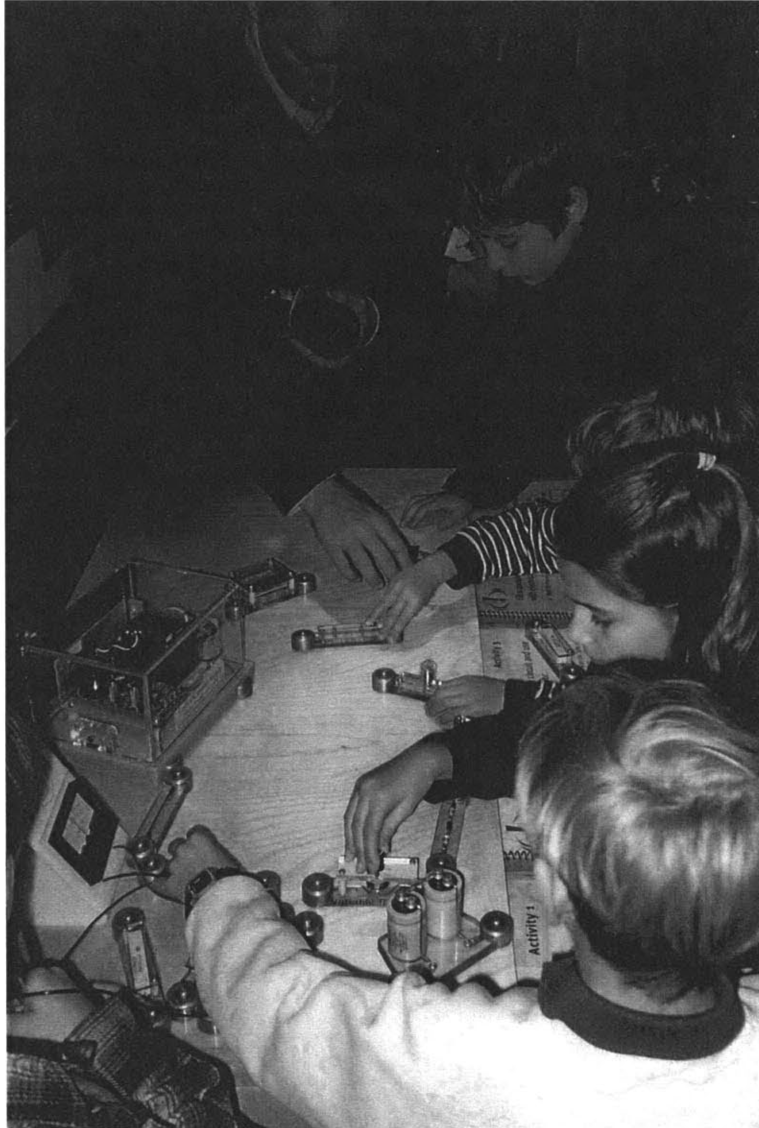
1. In some ways, "visual" is a misnomer, especially given that, when Microsoft says "visual" they really mean *textual* augmented with a direct manipulation GUI builder. Perhaps "graphical" is a better term.

When experts in human-computer interaction (HCI) say "direct manipulation" what they really mean is *indirect* manipulation using a mouse and keyboard to control "cursors" which act as our proxies. To the rescue comes the emerging field of *tangible user interfaces* where virtual objects inside computers are represented by physical "manipulatives" in the real world. To the lay person, the most elusive of virtual objects are programs themselves, the invisible building blocks that makes all computers function. It is my goal to de-mystify the art of programming by making programs as tangible as a stack of LEGO bricks.

1.1.1 Programming is not just for professionals

Programming in the broad sense is no longer a task relegated solely to professional programmers. Accountants use electronic spreadsheets to build business models. Musicians program their MIDI synthesizers to accompany them in performance. And a host of consumer electronics demand rudimentary programming skills from the general public: VCRs, microwave ovens, bread makers, thermostats, and cameras, just to name a few. It is in these consumer products that existing user interface technology shows its greatest weaknesses. VCRplus® was successfully introduced in the U.S.[1] after studies showed that even intelligent, well-educated people found it frustrating to program their VCR to record their favorite TV shows, preferring instead to type a single multi-digit number listed in their newspaper that encodes the program's time, duration, day of the week, and channel. But this approach gets the job done at the expense of taking creative control *away* from the consumer. "One touch cooking" might do a great job at baking a potato, but what about creating my own recipes?

1. G-code in Japan.



Inventors Workshop at San Jose Tech Museum

1.1.2 Kids as programmers

When computers were first introduced into schools, what we now call "computer literacy" often meant learning how to program "turtles" using the Logo programming language [38]. Nowadays, learning computer literacy is more likely to mean learning to use canned programs like word processors, paint programs, and educational computer games. Less and less computer science is taught to school children.[1] My goal is to reverse this trend by making programming a hands-on activity and by making it enjoyable to lit-

1. Though I am delighted to see that Brian Harvey's *Computer Science Logo Style* [19] is back in print.

erally "toy around with" fundamental concepts of computer science in a playful environment.

Fortunately, today we can still find children programming in Logo. In addition to instructing virtual turtles to draw geometric shapes on a computer screen, kids can program whole colonies of virtual termites [44], give behavior to toy robots, and build science experiments. These programs, written by children using, for example the LEGO Control Lab, do not take anywhere near full advantage of the potential of the computer, and, like the early days of Logo, these projects require supervision and guidance from teachers and mentors. Logo's syntax is partly to blame, but it is my claim that the main problem is that the mechanics of building, fixing, and downloading programs are necessarily complicated by even the best screen-based software construction tools. In this thesis I tackle this problem in particular, and hope to inspire researchers to go one step further by developing programmable toys which encourage children to modularize, abstract, and reuse the software components they build.

1.1.3 User interfaces of digital "everyday objects"

The comprehensibility of user interfaces in household appliances and handheld electronic devices took a dramatic nose dive with the introduction of inexpensive microcontrollers. Where once there was a natural, often a one-to-one mapping between the appliance's functions and the controls on the front panel, the embedded microprocessor allowed designers to decouple the functions from the controls, and even worse, allowed them to have fewer controls than functions. The epitome of this trend is the multi- function digital watch.[1] Although it might only have two or three push-buttons, it will boast a dual time-zone clock, calendar, timer, stop watch, alarm clock, and even an address book. Well, you might say, criticizing user interfaces with small displays and limited controls is like shooting fish in a barrel. At the other extreme is the home entertainment system remote control and restaurant cash registers where there are many functions and equally many buttons, yet the interface is still daunting. Even general-purpose computers with modern graphical user interfaces (GUIs) can be difficult to learn. Watching nov-

1. Arguably the first "wearable computer."

ice computer users learn to use even "user-friendly" software, one quickly notices that the mechanics of operating so-called "direct manipulation" user interfaces dominate the learning task. This is especially true for our parents' and grandparents' generation.

Even when the mechanics of GUIs are mastered, they frequently require so much attention from the user that they can't be used in social situations without the user appearing rude. This is particularly evident when manual dexterity is required for selecting and manipulating graphical objects, when (having nothing to do with GUIs) the interface is error prone, for example speech and handwriting recognition, and when the interface introduces long or unpredictable delays.

1.1.4 Information appliances

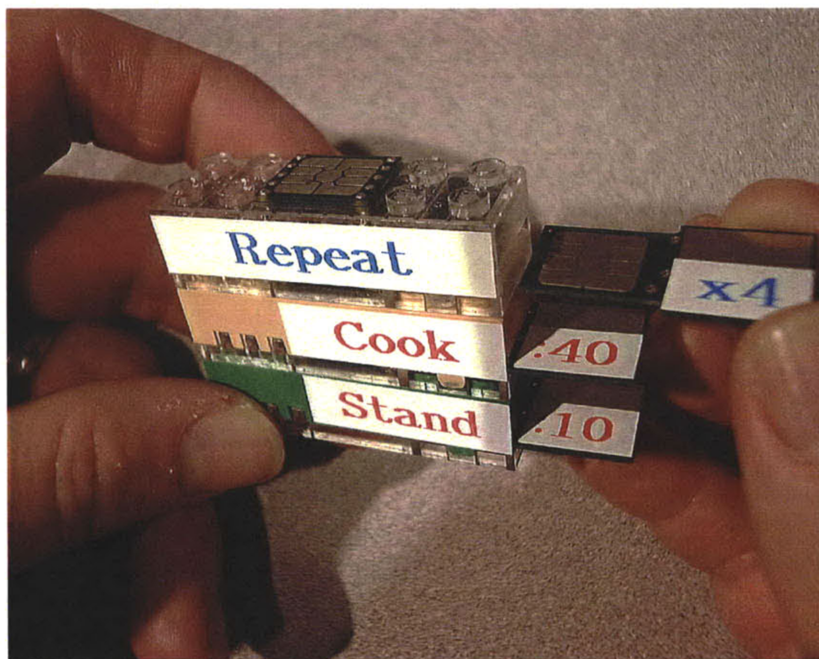
In his book *The Invisible Computer* Don Norman makes the case that general-purpose "personal computers" are becoming too complicated, and that special-purpose "information appliances" will in the end do a better job at certain tasks. But personal computers you have certain practical advantages, for example it is easier for several applications to share information when it is all on the same machine, and certain economic advantages, notably that, no matter how many software applications I buy for my laptop computer, I only have to buy the state-of-the-art color LCD display once. This is noticeable savings considering it is the single most expensive component in the device. The most successful information appliance to date is 3Com's Palm line of personal organizers. To keep costs down and to keep the unit compact, the display on a Palm Pilot is a small, low resolution, black and white display that can only display of few "objects" at a time. This limitation can be truly crippling when the user needs to manipulate even moderate volumes of information. My answer to this problem is to rely less on display technology by letting people see and manipulate information as physical objects.

1.2 Overview

The Tangible Programming Brick is a general-purpose research tool based on the Cricket architecture that can be repeatedly programmed using the Cricket Logo software development environment. It features an innovative connector system that is both electrically

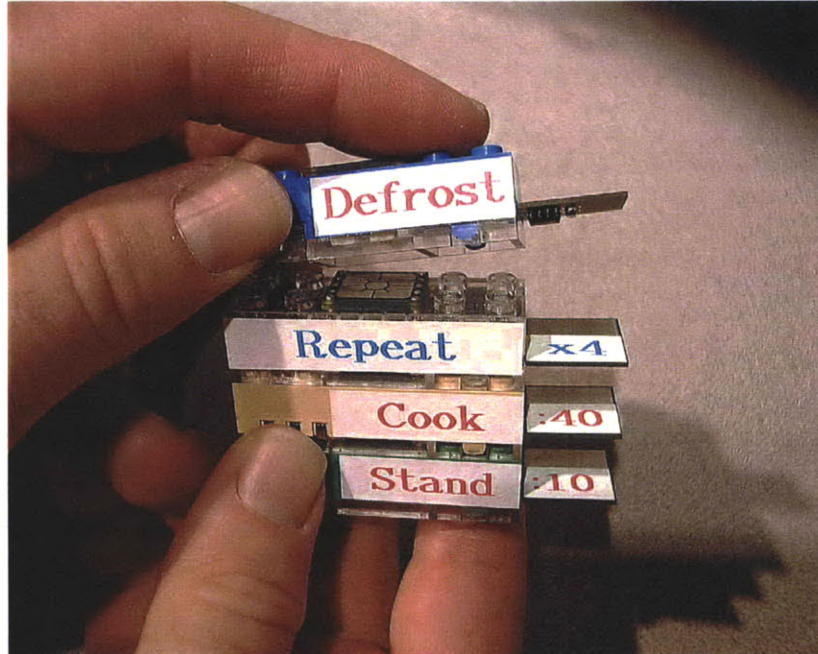
reliable and easy to assemble and disassemble by hand, three colored LEDs for user feedback, a capacitive touch sensor for user input, and a card slot which accepts a variety of peripheral devices including an I^2C EEPROM parameter card, an infrared communication card, an alphanumeric display, and a growing number of sensors and actuators designed for the Cricket bus system.

To the end user, the Tangible Programming Brick offers two "affordances" for constructing programs, (1) inserting parameter cards, and (2) stacking a Brick above or below other Bricks.



Inserting a parameter card

Based on these two basic interaction techniques, I implemented three simple programming languages using a set of Tangible Programming Bricks, one language to control toy cars, one language to control toy trains, and one language to control microwave ovens.



Stacking Bricks together

1.3 Organization of this document

This thesis is divided into five chapters. Chapter 1 has presented a set of problems that have motivated my research. Chapter 2 gives the context for my research: an introduction to tangible programming, previous work, including a comparison with other programming environments, and gives an overview of devices I have built that led up to development of the Tangible Programming Bricks. It discusses what kind of programs people can build with my system, and it discusses language issues and syntax. Chapter 3 provides details of the design of the blocks themselves, and concludes with a design critique. Chapter 4 offers a discussion of issue that are relevant to my research. Chapter 5 presents directions for future research and my conclusions. Appendix A contains schematics and technical drawings of the Tangible Programming Brick. Appendix B details the application software used in the microwave oven demonstration and my modifications to the Cricket Logo firmware.

Chapter 2 — Motivation and Context

> The first lesson that any technologist bringing computers into a classroom gets taught by the kids is that they don't want to sit still in front of a tube. They want to play, in groups and alone, wherever their fancy takes them. The computer has to tag along if it is to participate. This is why Mitch Resnick, who has carried on Seymour's tradition at the Media Lab, has worked so hard to squeeze a computer into a LEGO Brick. These bring the malleability of computing to the interactivity of a LEGO set. [14, p. 146]
>
> A child has a wealth of knowledge about how the world works that provides the common sense so noticeably absent in computers. Similarly, Seymour Papert feels that the use of computers for education has gotten stuck. We learn by manipulating, not observing. It's only when things around us can teach us, that learning can be woven into everyday experience. He's not looking to duplicate the mind of a good teacher; he just wants a tennis ball that knows how it has been hit so that it can give you feedback. [14, p. 201]
>
> Neil Gershenfeld, *When Things Start to Think*

2.1 Scope: What is programming?

For purposes of this thesis I define programming very broadly. It ranges from the complex to the simple and everyday. Computer programs written by software engineers can take years to write in span thousands of pages. Programs written by kids are typically less than a page, and even a few lines of code can produce very interesting behaviors. It is programs of this scale that this thesis focuses upon. I make no claims that my techniques are ready to compete with, for example, Metrowerks CodeWarrior, but I do not consider this a serious shortcoming, as even simple programs can be profoundly empowering to children and adults alike.

To consumers, the term "programming" often means something more basic like getting the VCR to record their favorite TV show on Thursday night. Sometimes programming really means configuring, tailoring, or personalizing. These activities often have little to do with time or sequential actions. For example, in most television broadcasts areas there are unused channels. Modern television sets allow the owner to identify these unused channels so that pressing the channel up button skips to the next active channel. This is an example of an activity which is only done once every few years, so it is typically per-

formed with manual in hand, if at all. Another such example is programming a modern thermostat to set a different temperature at night, during the day, and on weekends, according to the life-style of the inhabitants. Other kinds of consumer programming are technically feasible but rarely done. On some bread makers it is possible to set the start time and fine tune temperatures and times to suit a particular bread recipe, but most people, finding this too cumbersome, just stick to the basic settings. This is not surprising considering the number of VCRs that flash 12:00 day in and day out.

Even in the computational world there are "programs" which aren't quite programs. For example, accountants do not have to be programmers to build electronic spreadsheets. This is because a spreadsheet describes functional (i.e. "what is") relationships between cells in the familiar grid, leaving the procedural (i.e. "how to" compute) aspects to the computer. Similarly, database searches are expressed using a query language (e.g. SQL[1]) that allows users to specify well-defined operations and filtering on vast quantities of data without any mention of what order the operations are to be performed.

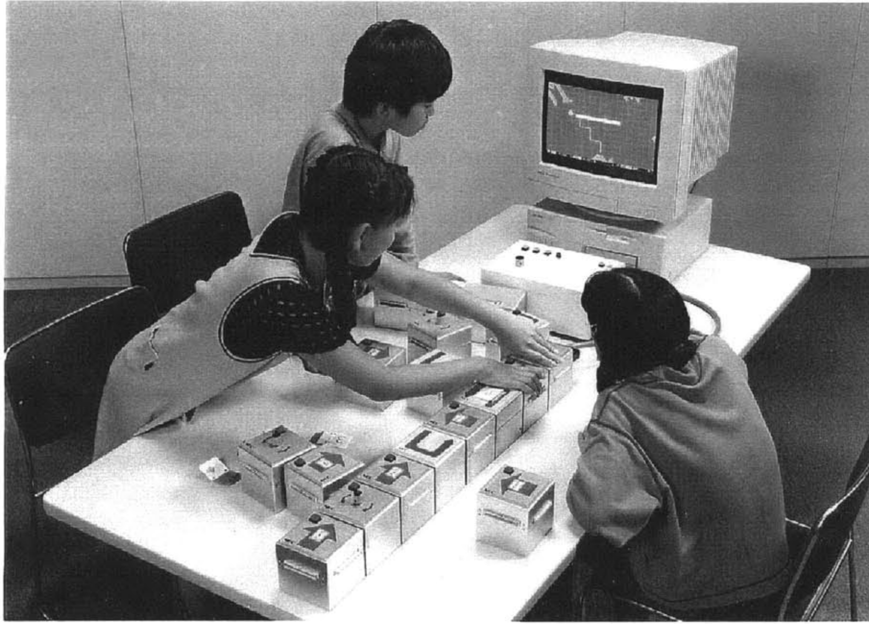
2.2 What is tangible programming?

The term "tangible programming language" was coined by Suzuki and Kato to describe their AlgoBlock collaborative programming environment for children. The unique feature that sets AlgoBlock apart from other programming environments is the graspable nature of its user interface. Instead of manipulating *virtual* objects displayed on a computer screen, users of AlgoBlock arranged *physical blocks* on a table to communicate to the computer. "Tangible programming" refers to the activity of arranging the blocks to build (as opposed to "write") computer programs.

2.3 Why tangible programming?

There are a number of reasons why one might want to program by manipulating physical objects. Some people, adults and children alike, learn more readily when their bodies are involved in the learning process. Some kinesthetic thinkers, as they are sometimes

1. SQL = Structured Query Language (pronounced "sequel")



AlgoBlock collaborative programming system

called, go on to become star athletes and virtuoso pianists, but are less successful at learning activities that aren't at all physical. My primary motivation is to make programming an activity that is accessible to the hands and minds of younger children by making it more direct and less abstract. Tangible programming may have an appeal even to experienced abstract thinkers. Graduate students have suggested that they long for the day when they can quickly prototype software as easily as they now "breadboard" electronic circuits.

2.3.1 Collaborative Programming

One of my original motivations for pursuing tangible programming is to design a programming environment where a small group students can build programs together. Like Suzuki and Kato [54, 55], I am interested in programming environments which encourage collaboration. This is difficult using traditional screen-based programming environments because only one user can type on the keyboard at a time. When a program is constructed out of physical objects, several people sitting around a table can work together to assemble or modify the program as a team or each person can build or modify their own methods independently of their teammates.

2.3.2 Gender differences

One of the things to which I attribute my success at learning about computers was that I was interested in them as an end in themselves and less as a tool. In the mid-'70s computers weren't used casually as tools, they were used as business equipment and scientific equipment. If you were not a scientist or a computer professional, you had very little opportunity to use a computer as a tool. Kids nowadays primarily use computers as tools and as games. They can use these interests and activities as stepping stones to an interests in programming. There are well-documented differences between the sexes in "kid culture" and "computer culture" [26]. This makes the transition for boys much easier than transition for girls to become programmers. We hope that, by introducing tangible programming as an activity performed away from a traditional computer (with keyboard, mouse, and screen), girls and boys will be equally engaged in this computational but not "computer" learning activity. There is growing anecdotal evidence that making programming accessible to younger and younger children, at a time when gender differences are less developed, has the effect of narrowing the "gender gap" in later grades when the more technical "crafts" traditionally become dominated by boys [35].

2.3.3 Debugging

The most difficult aspect of learning how to program is learning how to debug your programs. Debugging is a difficult activity to teach beginners, because the process of debugging is largely a process driven by knowing what to look for and having an extremely clear model of what is going on inside the computer. Since it is exactly this modeling of these invisible processes that one is trying to teach when teaching students how to program, learning how to program is hard. The threshold for becoming confident is high. One of my original goals is to make these invisible processes visible by making programs something that you can watch as it runs, thereby making debugging a skill with a shallower learning curve.

2.3.4 Limitations of "visual" programming languages

Screen-based graphical programming languages suffer from a number of limitations: Tools for manipulating textual programming languages are much more mature than

graphical programming tools. Textual programming languages make better use of screen real estate then graphical programming languages, which often include extra decorations around each functional block.[1] In general, the primitive state of metaphors and tools for a manipulating complexity in programs constructed using visual programming languages result in visual programming languages being difficult to express and manage complex software systems. In the domain of programming languages for children, this is less of an issue because novice programmers need to gain experience writing simple programs before they tackle larger, more complex programs.

A common approach for evaluating and comparing visual languages is to quantify how elegantly and concisely a particular algorithm can be implemented (e.g. generate sequence of prime numbers) [18]. I believe that this is too narrow a measure of programming languages, and one which is driven by the notion that programs are not part of the everyday world around us.

2.4 Related Work

My work sits squarely between visual programming languages [17, 40, 51], direct manipulation [50], tangible interfaces [16, 23, 54, 55, 58, 61], and "end-user" programming [52]. I focus my attention specifically on children as programmers. A number of programming systems have been designed for children: Logo [2, 38], ObjectLogo [10], the TORTIS Button Box and Slot Machine [40], ToonTalk [25], Cocoa (a.k.a. KidSim) [51], Agentsheets [15, 42], and AlgoBlock [54, 55], just to name a few. My work follows previous research done at the Epistemology and Learning group at the MIT Media Lab, where researchers have developed a number of computational toys [31, 45, 46] designed with education in mind. Professors Mitchel Resnick (MIT), Robbie Berg (Wellesley College), and Mike Eisenberg (University of Colorado), have developed a methodology and research agenda called "Beyond Black Boxes" designed to encourage kids to explore science by building their own scientific apparatus [47]. My research was influenced in intangible ways by this project.

1. This is closely related to a phenomenon that Tufte calls "chart junk" (a typical edition of USA Today contains a number of graphs with examples of "chart junk")

My work has been particularly inspired by fundamental principles of human-centered design espoused by Don Norman [36], who hopes that, one day, computers will be so embedded in everyday objects as to be rendered invisible [37], by graphic designer and visual thinker, Edward Tufte [56], and lastly, but not least, by the work of Guy Steele, Hal Abelson, and Gerry Sussman, who developed not only a powerful, expressive programming language, Scheme [53], but more importantly, a way of thinking about computation which allows programmers to capture how they *think* about a problem, not merely how to solve it [1].

2.4.1 Tangible User Interfaces

Ullmer and Ishii define tangible user interfaces as "user interfaces employing physical objects, surfaces, and spaces as representations and controls for computationally mediated associations." [59] They refer to these physical objects as "tangibles." My system, like many tangible user interfaces, is composed of a collection of tangibles. In the conceptual framework of Fitzmaurice, Buxton, and Ishii, my system is a "graspable user interface." [12]

Unlike systems where absolute position and orientation of objects in space control the interaction (e.g. Illuminating Light [61]) my system uses sequence and juxtaposition (e.g. mediaBlocks [58]), and constructive assembly to convey meaning, configuration, and topology (e.g. Geometry-defining processors [3], Triangles [16], and MERL Blocks [4]).

Most closely related to my research is a work by Suzuki and Kato. They implemented their AlgoBlock system to study collaborative learning of programming concepts among children, and in [54] they coined the term "tangible programming language." In their system, each hand-sized block is roughly equivalent to one Logo statement, for example "go forward," and "turn right." The tangible program assembled by the children directed an on-screen submarine around an obstacle course.

2.4.2 Direct Manipulation User Interfaces

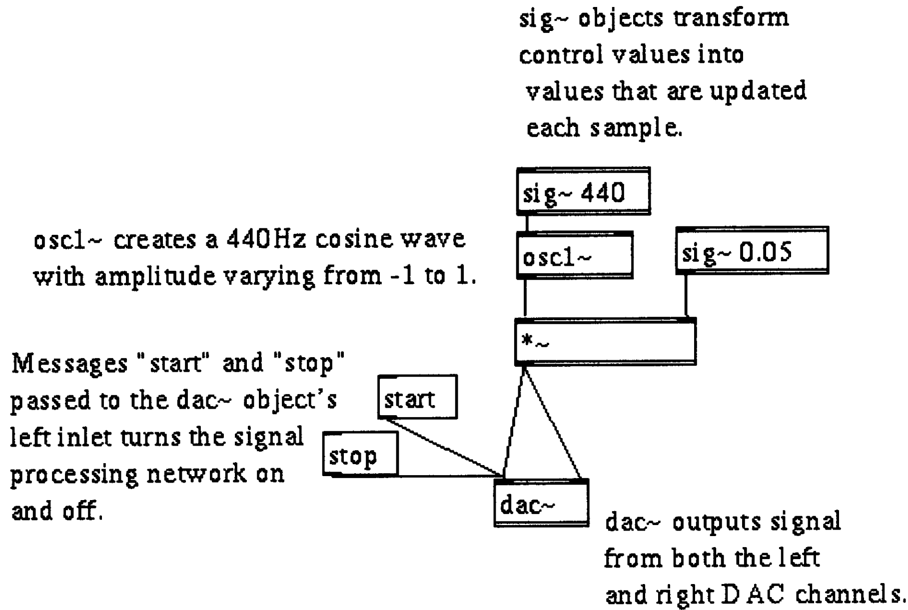
Schneiderman writes in [50] among others, about "direct manipulation user interfaces," and espouses their virtues over traditional command line interfaces, programming lan-

guages, and agents [29]. Direct manipulation user interfaces are almost singularly responsible for making computers usable by a wide range of users from trained professionals to small children, by endowing "objects" on the screen with a quality of manipulability that approaches real world objects. When screen-based objects (the "nouns") are distinct and well separated, and the number of actions that can be applied to them (the "verbs") is small, direct manipulation user interfaces work admirably well. But when screen based objects are small, numerous, or overlapping, the palette of actions is large, notions of "selection" and "operating mode" begin to cloud the picture, or there is little tolerance for error, for example in real-time music applications, direct manipulation user interfaces can be, at best cumbersome and frustrating. The main problem with direct manipulation user interfaces in the context of graphical interfaces is that they're not really direct. The "objects" are separated from the user by a pane of glass, and the only way to manipulate them is through a proxy (the "cursor") controlled via the mouse or other pointing device. The arrangement is reminiscent of teleoperated manipulators used to safely handle radioactive materials. Compared to ungloved hands, dexterity is reduced and, without extreme care, errors become more common. By contrast, tangible user interfaces can take advantage of the full dexterity of two hands unencumbered by go-betweens.

2.4.3 Visual Programming Languages

My work is related to the field of visual programming languages,[1] which strives to reduce the barrier to entry of programming for "end users" (i.e. not professional programmers) [52]. Most commercial visual programming languages (e.g. IRCAM/Opcode

1. Microsoft Visual Basic and Visual C++, although very popular, are not truly visual programming languages. They are software development environments that combine traditional textual programming languages with a direct manipulation, graphical, user interface builder.



IRCAM Max DSP example: A sine wave generator

MAX, National Instruments LabVIEW [24], and Cassidy and Greene Spreadsheet 2000) to use a "box and wire" paradigm, reminiscent of flowcharts, to indicate function and data flow. I deliberately chose not to use physical wires to avoid physical tangles.

2.4.4 Logo and its descendants

Ever since and Papert's seminal work on the Logo programming language [38], much work has been done on programming environments for children. A number of descendants of Logo exists including StarLogo [44], which introduces parallelism as a way of allowing children to explore aggregate and emergent behavior by programming "flocks" of "turtles" with simple rules, and ObjectLogo [10], which introduces object-oriented programming without the usual complexity of instances and classes.

2.4.5 TORTIS button box and slot machine

In 1976, Radia Perlman and Danny Hillis [40] built the TORTIS button box and slot machine to explore how non-traditional interface techniques might be used to teach pre-school children about programming by avoiding the cumbersome mechanics of typing in Logo programs. The button box presented to the user an array of large buttons that

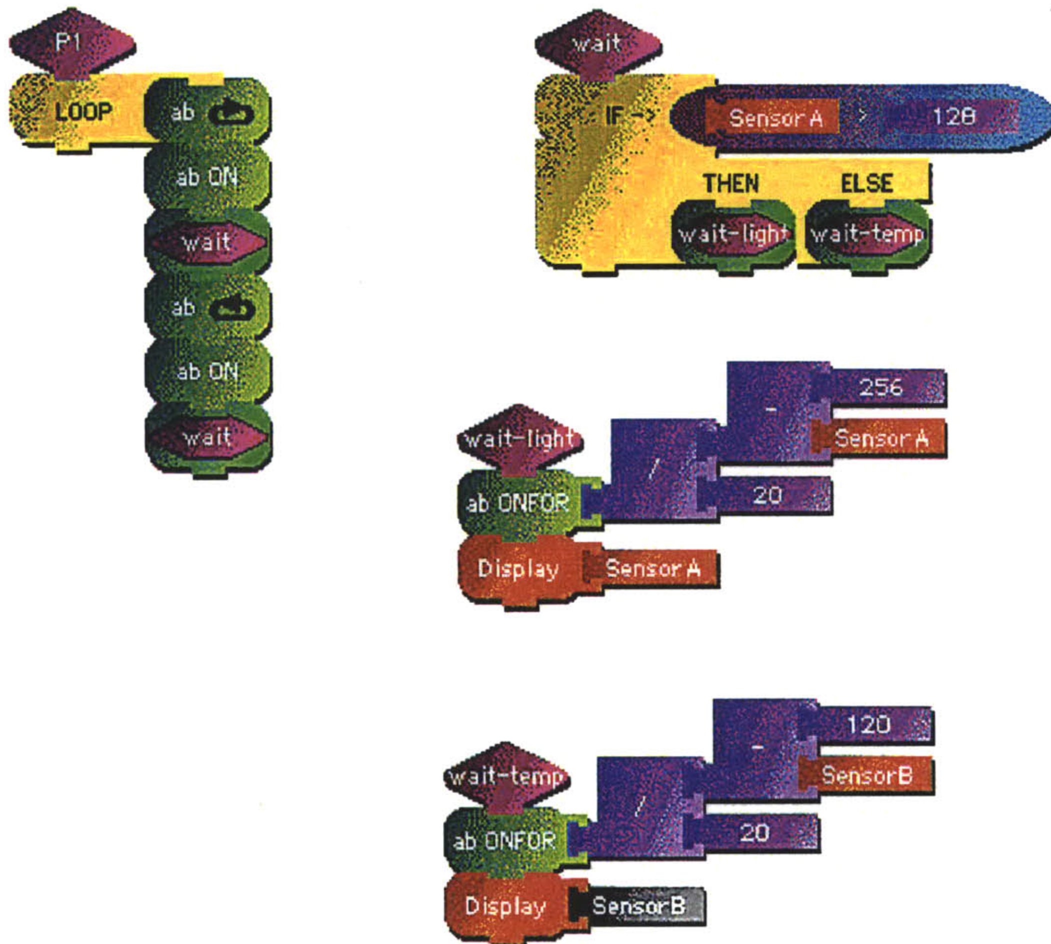
could be used to directly control the actions of a "turtle" (a small, bubble shaped, robotic vehicle tethered to the computer). The the basic button box provided only simple commands like forward, back, right, left, and beep, which could be repeated several times for greater effect (e.g. turn further to the right). Experiments were conducted with an auxiliary button box which provided numeric parameter buttons 1 through 10. In addition to controlling the turtle's actions, a transcript was displayed on the screen and made available for further manipulation.

The slot machine was possibly the first tangible programming language. It consisted of four color-coded racks of slots in which cards could be inserted, side-by-side. Each card represented a command with a fixed parameter (e.g. forward 10). Other racks (a.k.a. procedures) could be invoked ("called") with a colored card. As the program was executed in sequence, lights under each slot would indicate which card was being performed.

2.4.6 Logo Blocks

Logo Blocks [5, 49] is a visual programming language directly based on Logo. Its structure is much more two-dimensional in nature, and also provides procedures and procedure calls. The visual aesthetic of Logo Blocks is reminiscent of a colorful jigsaw puzzle. One limitation of Logo Blocks that is also an issue for tangible programming is the fixed size of the blocks, which sometimes makes it cumbersome to assemble certain legitimate programs without introducing "padding" blocks. In Logo Blocks this could be solved by

making certain blocks stretchable. In the tangible world this tension between physical structure and semantic structure is an ongoing challenge.

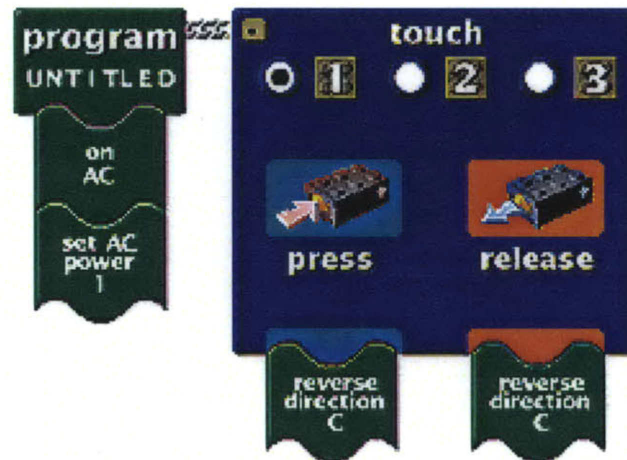


Logo Blocks example: Motor "wiggles" faster when sensor A gets hotter

2.4.7 LEGO Mindstorms

LEGO's "RCX Code" visual programming language was introduced with the Mindstorms product debut in 1998. It was inspired, in part, by an early prototype of Logo Blocks [34]. In the LEGO tradition, an RCX Code program consists of a collection of one-dimensional instruction "stacks" made up of brick-like icons. Each "brick" is a complete program statement, for example

set AC power 1



An RCX Code Sensor Watcher

The parameters to this statement (AC and 1) are filled in by clicking on the brick. This expands the brick (like a familiar "dialog box") revealing the appropriate menus, buttons, and text fields. It is worth noting that in the tangible world there are no dialog boxes, so a statement like this needs to be assembled out of individual parameter blocks.

RCX Code is suitable only for writing very simple programs. This is partly due to limited screen real estate, and limited language features. For instance, though there are procedures and procedure calls, there is no procedure parameter passing,[1] and there is only a single variable/counter.

2.4.8 Other languages for children

Of course, not all languages for children derive from Logo. AgentSheets and Visual AgenTalk [42, 43] provides a more powerful visual programming language for designing and sharing screen based simulations in the spirit of SimCity®. Programs in Visual AgenTalk are a collection of rules with patterns and consequent actions expressed partially as text and partially as icons. I believe this particular programming style is particularly well-suited for implementation as a tangible programming language. Cocoa (a.k.a. KidSim) [51] is a visual language inspired by AgentSheets that allows kids build animated simulations by designing sets of simple, graphical "rewrite rules" to control

1. So some people would say these are really "macros."

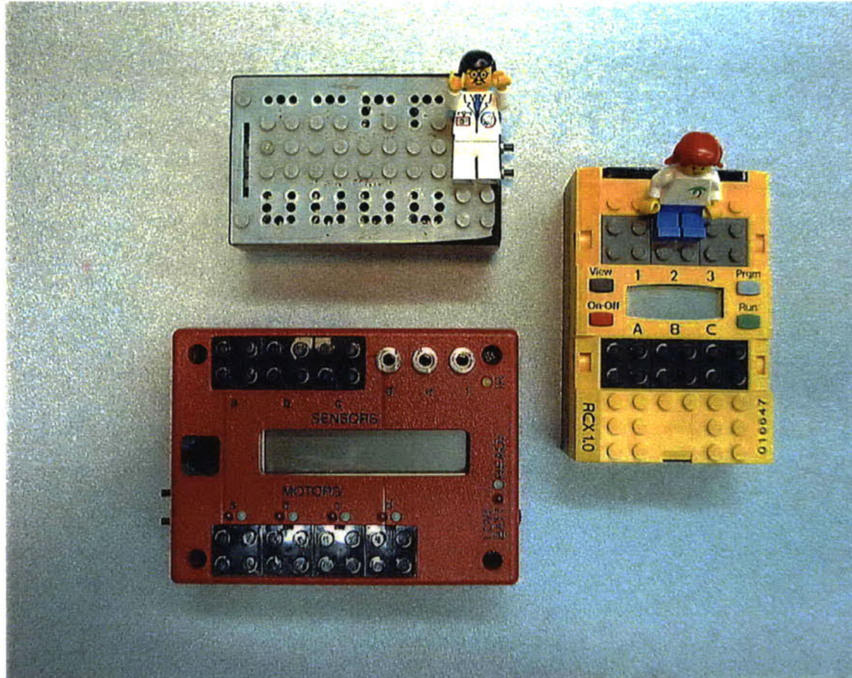
behavior of animals on the screen. Here a rewrite rule is composed of a "before" picture showing a situation (a pattern to match) and an "after" picture (the rewrite). Smith's canonical example shows a child faced with the task of teaching a gorilla to move forward and jump over rocks. The completed program includes patterns and rewrites for each phase of the jump, in addition to a simple rule for moving forward. Ken Kahn's ToonTalk [25] is a very powerful object-oriented, programming-by-example, video-game-like environment for kids. Kahn has succeeded through carefully chosen metaphors and fun animated characters and data structures to make accessible to children a programming language rich enough for computer scientists.

2.4.9 Ubiquitous computing

Ubiquitous computing [62] and information appliances [37] promise a world where general-purpose computers take a back seat to small devices dedicated to a single task. Instead of personal computers which critics argue do to many things not very well, and which are confusing for "non-computer literate" people, advances in this direction would result in a plethora of everyday objects with computers inside but virtual extinction of the term "the computer." Tangible programming fits neatly into this model by providing convenient and intuitive interfaces to digital appliances without keyboards or displays.

2.5 Hardware History

I didn't develop the Tangible Programming Brick in a vacuum. It descended from a long line of small, programmable computers developed at MIT by Fred Martin and others.



The Gray Brick, the P-Brick and the LEGO RCX

2.5.1 Programmable Bricks

The MIT Programmable Brick (a.k.a. P-Brick) [32] is a small computer programmable in Logo. It was designed so children could incorporate cybernetic behavior into their LEGO construction projects [30]. It has an internal battery, sensor inputs, motor control ports, and a two line LCD display. The P-brick went through a number of design iterations, and was eventually commercialized by LEGO as the RCX Brick, which has a simplified programming model.

2.5.2 The Handy Board

In 1989 Fred Martin, Randy Sargent, and P. K. Oberoi created the MIT 6.270 LEGO robot design competition and developed the hardware and software for students to design and program their own robots. The Motorola 6811-based microcontroller used in the course, after going through a number of design iterations, including the "Mini Board," eventually grew to become the popular "Handy Board." One of the keys to the Handy Board's success was the "Interactive C" programming environment which allowed stu-

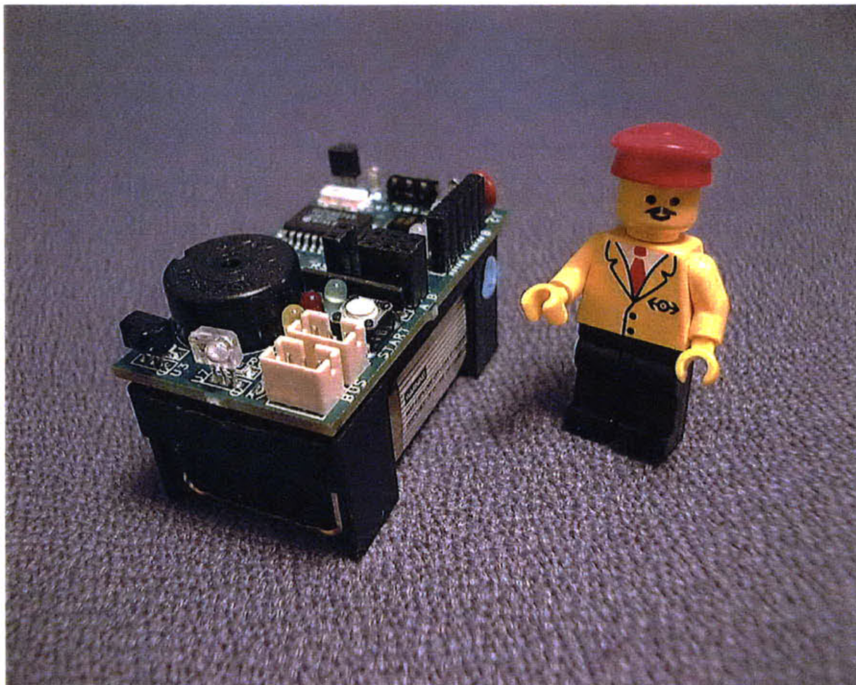
dents to quickly develop and debug their robot software. The hardware is very similar to the Programmable Brick.

2.5.3 Crickets and Thinking Tags

The Cricket was developed by Fred Martin, Brian Silverman, and Robbie Berg as an outgrowth of the Thinking Tags project, originally developed for the 10th anniversary of the Media Lab. The Tags, or "affinity badges" are worn like a name tag. They communicate with other badges via infrared light, much like a TV remote control, and have green and red LEDs to indicate how much two people have in common (e.g. interests, opinions, etc.).

Crickets are programmed in a subset of Logo. Logo programs are developed on a personal computer and downloaded via an infrared link to the Cricket.

The Cricket and the Cricket Logo programming environment provided considerable leverage as I pursued my research in tangible programming. Cricket users benefit from being able to program in a high-level language on a desktop computer, downloading their programs to one or more Crickets, and then interactively debugging the resulting



The "Blue Dot" Cricket

system by remote control from the desktop computer. I modified the Cricket operating system and extended the Cricket Logo interpreter to accommodate the design of the electronics in the Tangible Programming Brick. This allowed programming efforts to progress very quickly.

The design and troubleshooting of the electronics in the Tangible Programming Brick also went very smoothly because my design was based on the proven technology of the Cricket. In hybrid software-hardware designs, there is frequently a bottleneck in the testing process because the software cannot be finished before the electronics are prototyped, and the electronics cannot be tested before working software is available.[1] I addressed this problem by augmenting the electronics of several Crickets to reflect the modified electronics of the Bricks, and testing software changes on this platform before the assembled electronics of the Bricks were available. I originally prototyped a capacitive touch sensor based on a design by Rehmi Post [41] as a separate Cricket bus device, and I was able to transplant the code into the Brick with very few problems. In the end I only needed to discard three Bricks due to nonworking software.

2.5.4 Tiles

Inspired by his earlier work with (1D) electronic Beads, Kwin Kramer developed the 2D Tiles [27] system. Each Tile is a $2^1/2''$ square and contains enough computational power to run a subset of Java. Kramer's goal was to demonstrate the power of "mobile code" in a toy network environment which is constantly being reconfigured. Each Tile communicates optically with its north, south, east, and west neighbors. Programs "jump" from tile to tile and display their behavior on the same bi-color LEDs used for communication. The two-dimensional, composable nature of the Tiles got me thinking about building new learning activities using this sort of technology.

---

1. In this case there was an additional bottleneck because the hardware could not be tested before the packaging and final assembly work completed. Because I was using a One Time Programmable (OTP) microprocessor, I had to discard any Bricks with intermediate or broken software.

2.6 Setting the stage

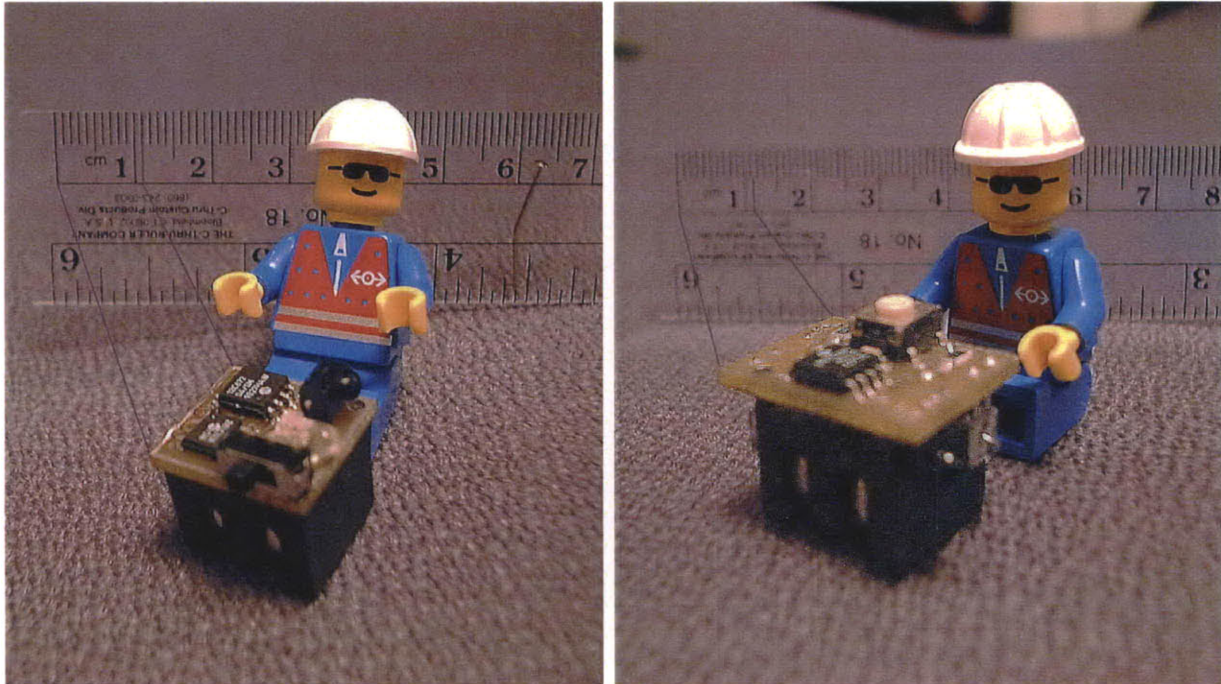
2.6.1 The tangible programming challenge

Ever since the Tiles project came to fruition, I had been thinking about how to use devices like the Tiles to physically build programs. The idea was to have tiles, or something like them but with less computational power, to serve as individual tokens from which program would be built. I had envisioned that these token tiles would be connected to a "compiler" node which would "parse" the structure of the tiles and use this structure to produce an executable program. As I got thinking more about snap-together elements it became more clear that this compilation step was not necessary. The tiles themselves could execute the program steps in order. This is especially interesting when the actuators and sensors are attached directly to the relevant Bricks in the program.

To understand this distributed model, consider a variant of Scrabble® where the tiles light up when words are formed. As children are learning how to spell, this toy would let them serendipitously discover new words. In the distributed model, rather than having a central overseer, each individual Brick or tile can have a dictionary inside it the. Alternately a tile might simply recognize sub-patterns and communicate with their neighbors to decide when to light up a section of letters without needing an entire dictionary.

2.6.2 Magnetic Programming Kit and microTiles

During the summer of 1998, shortly after Kwin Kramer debuted his Tiles project, Vennila Ramalingam, a Ph.D. student of Professor Mike Eisenberg from the University of Colorado, visited our lab. She was interested in developing a tangible programming metaphor based on refrigerator magnets. She dubbed it the "Magnetic Programming Kit" because it reminded her of the successful Magnetic Poetry Kit. In support of this project, I designed and prototyped a Cricket-based tile called the Micro-tile. Like Kwin's Tiles, the Micro-tile was designed to communicate with its four neighbors using short-range infrared light.



The Ladybug and the Ladybug2

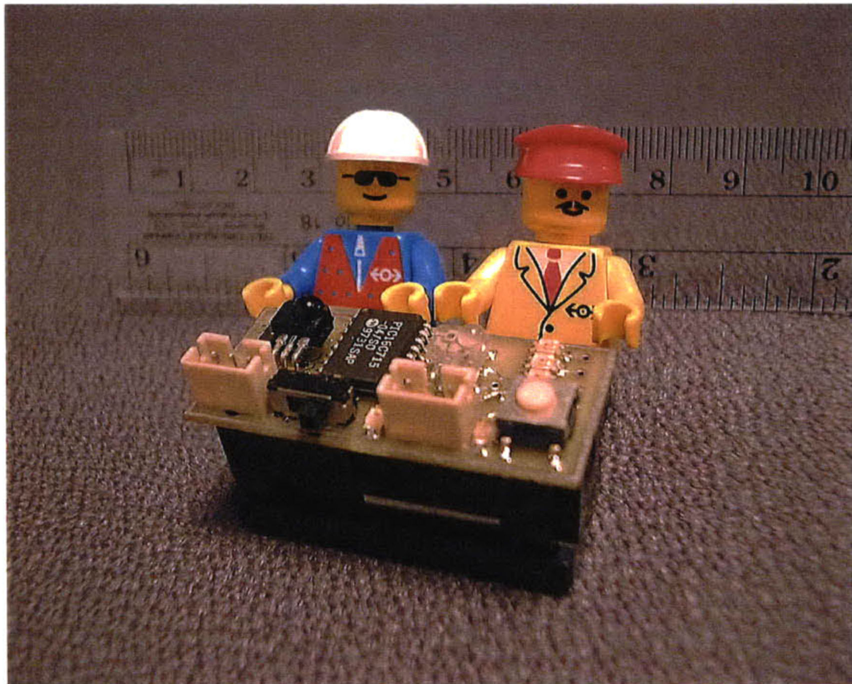
2.6.3 Ladybugs

As a project for Professor Hiroshi Ishii's Tangible Interfaces class, I designed and built the Ladybug,[1] a yet tinier version of the Cricket. It is shaped like a cube measuring about 1 cm on a side. I designed the Ladybug to be a remote "touch transponder". Using a minimalist capacitance sensor designed by Rehmy Post (which employs only a microprocessor and a mega-ohm range resistor), the Ladybug was designed to sense a person's touch and relay this sensor information, via infrared light, to a central user interface manager. As a follow-on, I built the Ladybug 2, which fixed problems with the touch sensor, added a push-button, and provided infrared communication in two (opposing) directions, much like Kramer's Beads communicated using inductive coupling [27]. Both Ladybugs were based on a 3 volt lithium "$^1/_3$ N cell" (e.g. 2L76), and an 8-pin PIC12C672.

1. Many thanks to Chris Hancock for contributing the name.

2.6.4 Blocks

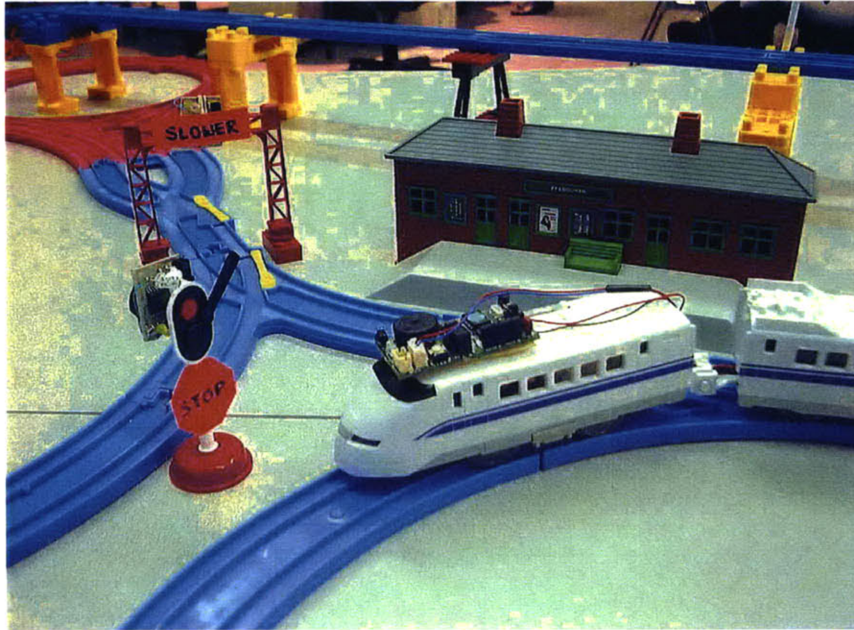
For additional printed circuit board real estate, I moved to a $^{2}/_{3}$ AAA-sized, 6 volt, silver oxide cell (e.g. 544). This line of Crickets were unceremoniously named Blocks. They had two capacitive touch sensors, three LEDs in place of motors, and two bus connectors: one master bus (the regular Cricket bus used to communicate to external sensors and other peripherals), and one slave bus. The additional bus connection allowed a Block to act as a "bus device" to another Cricket. This turned out to be a key architectural component of the Tangible Programming Bricks, as it allows a stack of Bricks to communicate with each other.



The Block version 1.2

2.6.5 Programmable toy trains

In the "Tangible Programming with Trains" project [33], Fred Martin and Genee Lyn Colobong built an interactive play environment based around a toy train set to allow young children to learn "pre-programming concepts." The train itself contained a small microprocessor and an infrared receiver. Around the track there were a number of signs and signals containing infrared transmitters. Children controlled the train, not with the

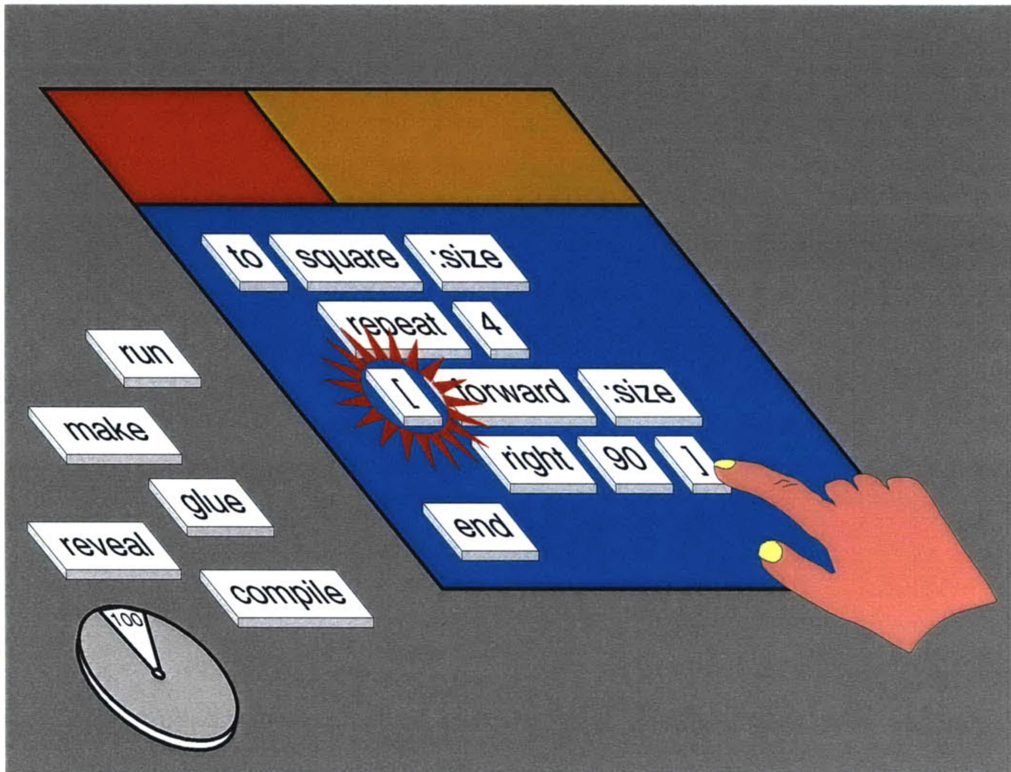


Learning "pre-programming" with a modified Tomy train set

traditional remote control console that came with the train set, but rather by moving the signs and signals around the track. The collection of signs included *stop*, *faster*, *slower*, *lights on*, *lights off*, and *sound horn*. The children found it fun to arrange realistic scenarios like making of the train turn its lights on when it goes into a tunnel, but in essence they were engaged in a kind of programming activity.

2.7 Tangible Syntax

My original motivation for pursuing tangible programming techniques was to provide tools for programming computers that were more conducive to learning and collaboration than traditional keyboard-, mouse-, and display-based programming environments. I imagined a new generation of young programmers building programs together with their hands instead one person typing on a keyboard and others looking over their shoulder. However, my notion of programming as an activity was narrowly focused on the construction and manipulation of traditional text-based programming languages such as Logo. Even though the inventors of Logo worked hard to simplify its syntax, syntactic elements such as brackets still remain. Here I show a concept example of a user



Balancing brackets with a "touch transponder"

balancing brackets with a "touch transponder." As she touches one bracket, the matching bracket lights up.

During the development of the LEGO Tangible Programming Brick I continued to concern myself with giving the user feedback about syntax. The three LEDs were designed to provide feedback about nesting of syntactic structures, for example a "repeat" surrounding an "if".

As I began to focus on simpler languages, syntactic issues assumed less importance, but never entirely disappeared. Logo Blocks uses virtual connectors deliberately shaped to indicate the type of blocks they are allowed to mate with. The Logo Blocks graphical program editor enforces these syntactic constraints by allowing only blocks with correctly mating connectors to be "snapped" together. This "puzzle pieces" approach to avoiding syntactic errors works even better in the physical world, because these constraints do not have to be simulated. Mismatched connectors literally won't fit. Color can also be used

to indicate syntax, but just like screen-based visual programming languages, too many colors can lead to an undesirable "tutti-frutti" effect.

2.8 Programming Styles

I considered a number of programming styles during the design of my tangible programming languages: functional programming, imperative programming (regular, sequential programming with state and side-effects), and rule-based programming.

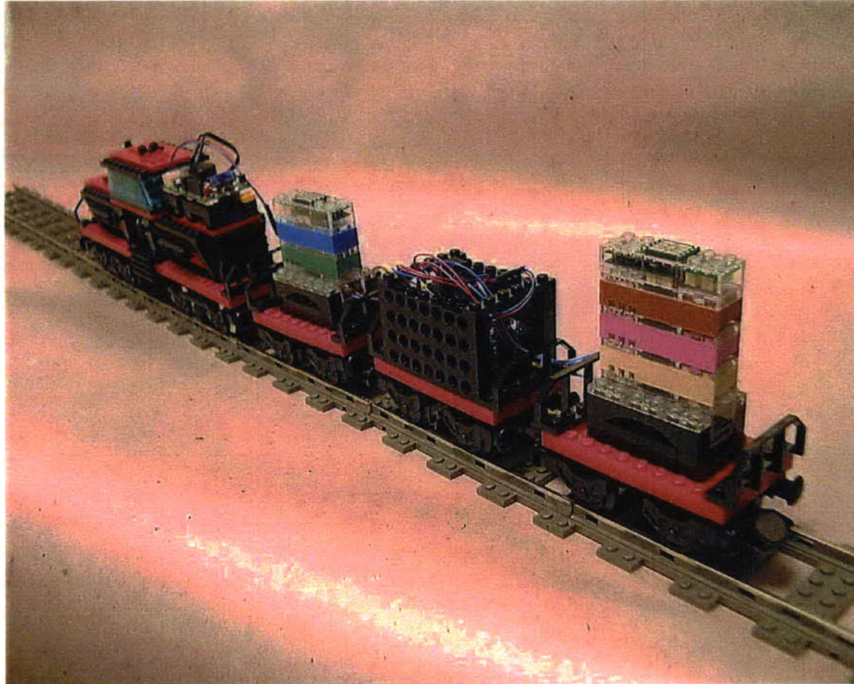
2.8.1 Imperative programming

Imperative programming is the oldest, least restrictive, most common style of programming. Sequential evaluation, state, and "side effects," the modification of variable, data structures, and objects in the outside world are all allowed. All of these issues are difficult for students to learn because they can have subtle repercussions. In fact, they are a major contributor to errors in commercial software.

2.8.2 Functional programming

Functional programming is a style of programming which does not allow sequential evaluation or side-effects. As such it is the most "mathematical" of programming styles. It has the advantage that it is easier for people to learn, and functional programs are easier for machines to manipulate, optimize, and prove properties about. It has a number of disadvantages, namely that it is difficult to make functional programs as efficient as imperative programs, and there are certain classes of problems which are difficult or impossible to solve with functional programs.

Mitch Resnick (et al.) proposed to the National Science Foundation a course of research which includes experiments with sensor blocks, function blocks, and effector blocks (motors, lights, musical instruments, etc.), to give younger children (K-3) the tools to learn about functions, integration, and derivatives, concepts usually taught in high school [48]. My Tangible Programming Bricks provide the literal building blocks to pursue the parts of this proposed research concerned with teaching about function blocks.



A train carrying tangible "rules"

2.8.3 Rule-based programming

Rule-based programming is a form of program organization. It is an attractive metaphor to implement in tangible form. One significant advantage is that programs are divided up into individual "rules" which can be modified independently of each other. This means that a small group of students can build a program together without getting in each other's way. In one language I designed (but did not implement), a rule consists of a message receiver Brick on top, followed by action Bricks which cause something to be performed when the message is received. A program is a collection of rule stacks. The message receiver Brick is similar to the LEGO RCX Code "sensor watcher," except that here the "sensor" is waiting for an infrared message to be sent. In RCX Code, a sensor watcher is a special kind of program building block which waits for a certain condition (e.g. temperature has risen above 100 degrees) and triggers a stack of code to begin execution.

2.8.4 Behavior mixing with priorities

In the spirit of Rod Brooks' subsumption architecture model of robot programming [8], several readers suggested that each Brick represent a single behavior, and that stacking Bricks together would cause the behaviors to be "mixed," with behaviors of the higher Bricks taking priority over the lower Bricks. This is an attractive programming metaphor, in particular because it reduces the chances that a program will be "wrong." One of the attractive features of mechanical construction kits is that they allow children to explore structural in mechanical systems without the fear that they might do something wrong. On the other hand, when children and adults learn how to use computers it is common to hear people described this very fear.

Brooks' original papers on the subsumption architecture might lead the reader to believe that mixing behaviors is easy. His graduate students and employees at IS Robotics who have been exploring the subtleties of robot programming tell a different story, that the architecture provides a good foundation, but getting good robot behavior requires careful fine tuning [11].

2.8.5 Database queries

With the growing popularity of the World Wide Web, professionals and nonprofessionals alike are finding themselves searching for information on a daily basis. Boolean expressions are frequently used to filter out the junk and hopefully leave us with what we were looking for. Whereas textual queries often require connective ANDs ORs, and NOTs, a tangible query might take a to the approach to syntax, representing AND as a vertical stack of filters, and OR as separate stacks in the horizontal dimension. For example,

Eyes: blue	Hair: brown
Height: average	Height: tall

would mean "eyes = blue AND height= average OR hair = brown AND height = tall".

The Tangible Programming Bricks are best suited for applications which are limited to a small vocabulary, so unrestricted web searches are not practical, but certain database

query applications are. For example, searching for clothing from a retailer's inventory database can be a fun, creative activity. It requires only a small set of filters and attributes that can be used over and over again: color, size, style, material, and category (pants, shirt, dress, socks, etc.). Color is difficult to describe as text, fortunately color "chips" can be used in a tangible database search. Searching a personal collection of documents can be a similarly bounded problem: authors in co-authors, subject, conferences, journals, and publishers, and recent years might only half a handful of choices each.

With a variety of programming languages in mind, I set out to design the elements of a tangible construction set that could be used to build programs. In the next chapter I describe the design of the Tangible Programming Brick, its associated plug-in cards, and the experiments I performed with the first set of prototype Bricks.

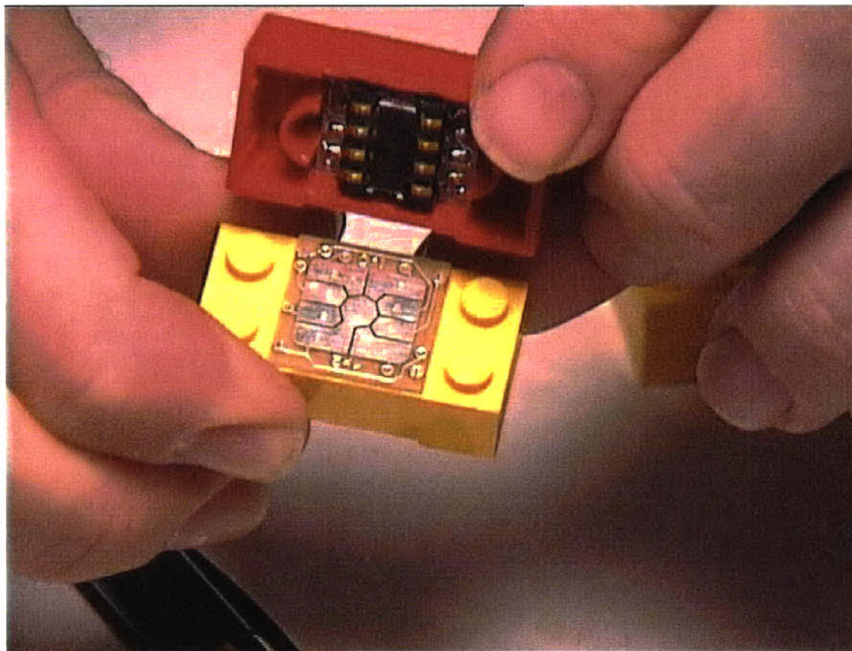
Chapter 3 — The Implementation

This chapter offers details of the Tangible Programming Brick. It begins with an overview of the first prototype that was built inside of a 4x2 LEGO System brick, it goes on to present the hardware and software design, and concludes with an overview of three applications I implemented using a set of the Bricks.

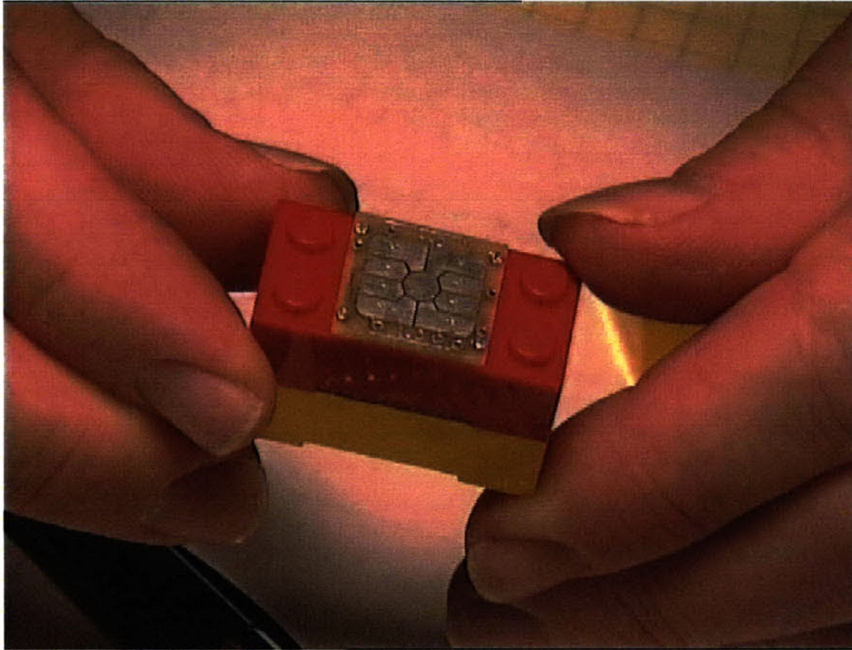
3.1 The Tangible Programming Brick Hardware

3.1.1 The early 4x2 design

The early prototype for the LEGO Tangible Programming Bricks was a 4x2 LEGO brick. The circuit design was based on the Ladybug. It contained a PIC16C672, a single LED, communication to neighbors above and below, and a "mode" input. The mode input is a global signal which passes through all of Bricks. This design was abandoned because it promised to be difficult to debug, in part because it didn't have the infrared communication ability which is used to program members of the Cricket family. But the main reason



A prototype of the early 4x2 Design



for moving to a 6x2 design was that there seemed to be a need for some sort of parameter mechanism, especially because my Bricks could be assembled only in a one-dimensional sequence, or stack. The extra space provided the option of using an 18 pin PIC and a ceramic resonator, avoiding the oscillator calibration problems that I experienced with the Ladybugs. Also I became interested in using LEDs to give feedback to user about the structure of the program being assembled, for example to show the syntactic extent of special forms like **if** and **repeat**, as shown in this diagram.

LEDs	Brick	Parameter
●	**If**	**Too Hot**
● ●	**Repeat**	**x3**
● ●	**OnFor**	**10 sec.**
● ●	**Wait**	**5 sec.**
● ●	**End Repeat**	
●	**End If**	

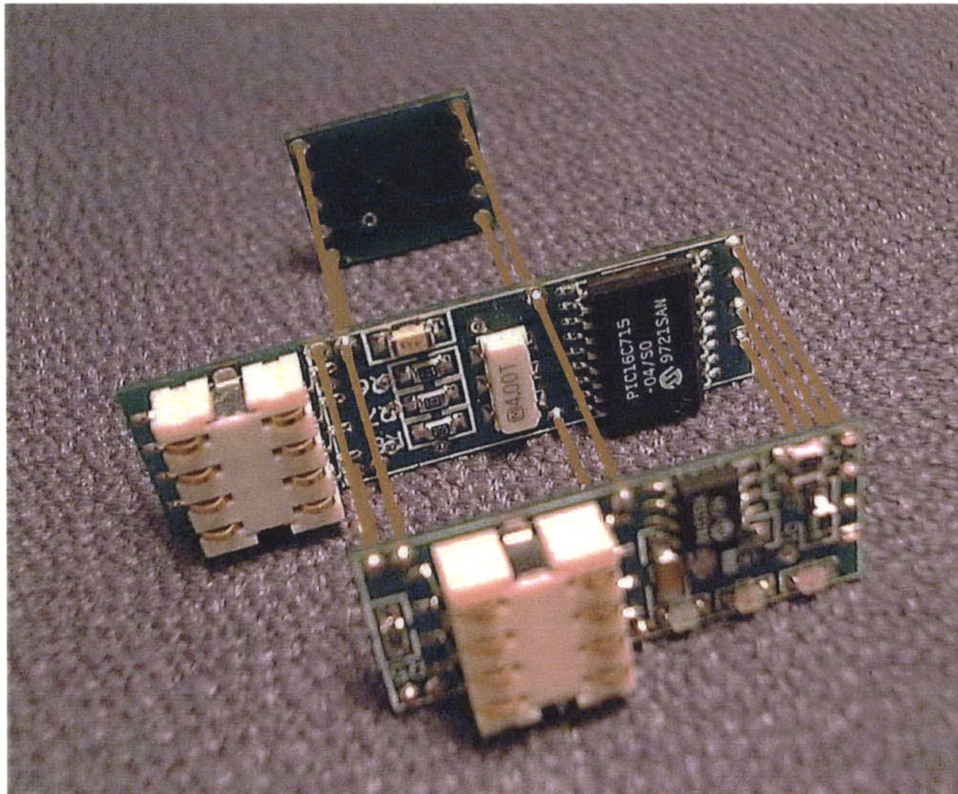
A stack of three Tangible Programming Bricks

3.1.2 The 6x2 Brick design

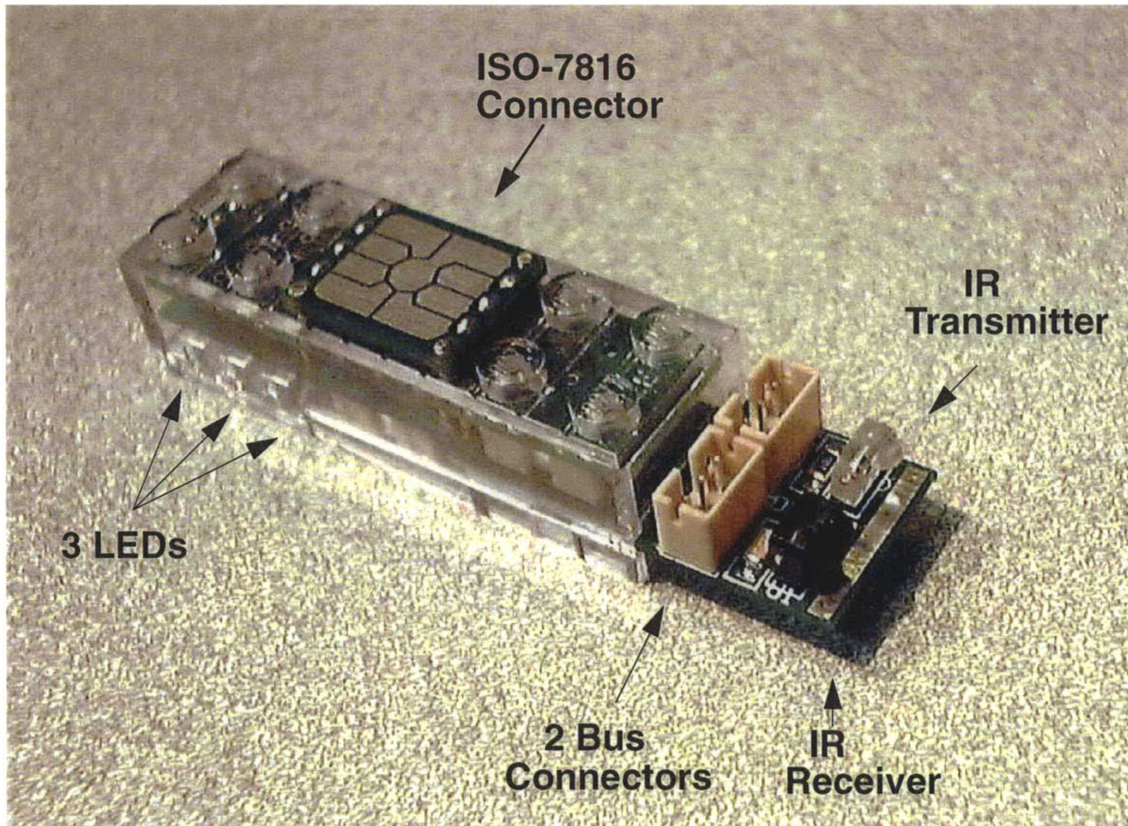
For my thesis, I implemented the Tangible Programming Brick, which was housed inside a custom 6x2 LEGO brick. Each Brick has a "smart card" (ISO-7816) connector, a female connector on the top, consisting only of metal pads on a printed circuit board, and a male connector on the bottom (a JST-ICC). This connector has eight gold plated springs which press against the printed circuit on the female connector. The connector system passes power, serial communication between neighbors, and a global "run/stop" signal. This global signal exactly corresponds to the Cricket run button functionality. It can be used to start and stop running programs on all of the Bricks in a stack simultaneously. In the user environment, this feature is not necessary because Bricks, like Crickets, can be configured to auto-start, running their embedded Logo program as soon as they are powered up. To allow for rotational symmetry, making it possible to snap together Bricks "backwards" (rotated 180 degrees front to back), the four power/signal lines are duplicated. This redundancy also provides an additional measure of fault tolerance for situations where the connectors are not seated properly.

Each Brick has three program-controlled LEDs on the front. Originally intended for user feedback, they have proven invaluable for debugging. Unlike the Cricket, there is no "run" light or power light, so there is no way of knowing whether the Brick is functioning correctly without making explicit provisions in the Logo program. The PIC processor has one unused I/O pin, enough to provide one additional LED, but there is no space available in the current design. Currently the LEDs are colored orange, amber (yellow), and green, reminiscent of a traffic signal. Red it was explicitly omitted because this color is reserved for signaling fault conditions by a number of national and international standards agencies.

The Tangible Programming Brick design was highly constrained by the size of the plastic capsule. The printed circuit boards inside are very small. This left little space for interboard connectors, so I had to use free-standing header pins to make electrical connects between the three PC boards.



Exploded view of Tangible Programming Brick electronics



A Tangible Programming Brick + IR/Bus Card

The design of the Tangible Programming Bricks is modular in several ways. (1) The Bricks stack together to form a system or program, and (2) a selection of cards is available to customize or augment the function of a single Brick.

3.1.3 The Cards

Each Brick has a slot on the side which accepts smart cards of nonstandard size that can be inserted into this slot to extend or customize the Brick.

I designed and built three types of cards that slide into the card slot on the side of the Tangible Programming Bricks.

- The **IR/bus Card** consists of an IR transmitter and receiver, a transmit LED for monitoring and debugging IR communications, and two Cricket bus connectors. This card allows the Brick to act as a member of the Cricket family. It can be used

for communicating with Crickets and other Bricks, for downloading Logo software from a desktop computer, and for communicating with Cricket bus devices such as sensors and motors.

- The **EEPROM Card** was designed for providing parameters to a Brick, but it can also be used for permanently recording information for later recall over for passing physically to other Bricks.
- The **Display Card** was designed for representing variable names or less common parameters, but it can be used for displaying any alphanumeric information, such as the temperature recorded by a LEGO temperature sensor.[1]

For maximum flexibility, two power and six signal lines are available to the cards.

- I^2C serial data
- I^2C serial clock
- Touch sensor/Beeper
- IR transmit
- IR receive
- Cricket serial bus

The I^2C serial bus signals were extended to the card so that the EEPROM Card could be implemented using a single chip. In retrospect this was a mistake because the I^2C bus is used by the Logo interpreter almost continuously. When a the EEPROM Card is inserted or removed, the communication between the PIC processor and the internal EEPROM is disrupted. This crashes the Cricket Logo interpreter. By contrast inserting or removing the IR/bus Card does not disrupt the processor. This makes it easy to program or customize an entire stack of Bricks sequentially with a single IR/bus Card by moving the card from Brick to Brick.

The IR transmit and receive lines are necessary to program the Brick using Cricket Logo running on a desktop computer. Originally I tried to fit the IR components inside the Bricks, but there was not enough space inside the plastic capsule. Fortunately, once a program has been downloaded into a Brick there is no need for IR communication.

1. Several Display Cards were manufactured, but I didn't have time to write the firmware.

Every card that plugs into a Tangible Programming Bricks has the option of including a capacitive touch sensor with no additional components. The card merely needs about one square centimeter of exposed metal on the top of the printed circuit board, preferably plated with gold for electrical and health reasons (lead-based solder is best if not touched). All three cards described here have a capacitive touch sensor pad.[1]

The Cricket bus is a bidirectional serial bus that can be extended to remote to devices or used to communicate to a Cricket bus device implemented directly on the card. An example of one such device is the Display Card.

3.1.4 The Evolution of Intra-Stack Communication

The Tangible Programming Brick is a descendant of the Cricket, and incorporates ideas that were first tried in the Block 1.2. The Block was designed to have the regular Cricket bus, which I shall call the Master Bus, and an additional bus connector, which I shall call the Slave Bus. Although this allows communication in two directions, it is not a symmetrical arrangement. The Master Bus can initiate communication with a bus device and can wait for a response, whereas the Slave Bus must passively listen for communication from another member of the Cricket family, and may respond only when "spoken to." This arrangement allows any device with a Slave Bus connection to serve as a "bus device" to any member of the Cricket family, but it constraints control and information to flow primarily from below (see Kitchen Brick in Appendix B for details). In the Tangible Programming Bricks, I decided to bias control flow so that Bricks on the bottom could initiate communication with Bricks above. This was motivated by the "compiler model" of tangible programming. The reverse is required to implement the "method model" where actions are directed from a top "listener" Brick. For this reason, future versions of the Bricks may provide a more symmetrical communication architecture.

1. The IR/Bus Card has a capacitive touch pad of minimal size included only for testing purposes.

3.2 The Tangible Programming Brick Software

3.2.1 Firmware

The firmware used in the Tangible Programming Brick is a modification of the "Blue Dot" Cricket Logo interpreter and operating system. The most significant modification to this code was the addition of the interrupt-driven "slave" bus facility. To the Cricket Logo user who is familiar with the IR receiver primitives **`new-ir?`** and **`ir`**, the new bus primitives will be easy to use. These new primitives are **`new2?`**, **`bus2`**, which corresponds directly to the IR primitives, and **`reply2`**, which is used to reply to a **`bsr`** sent from below.

The second major new facility is the capacitive touch sensor. **`sensora`** returns a number between 0 and 255 which represents the amount of capacitive loading on the touch pad of a card. **`set-touch-range`** allows the user to set a pre-scale parameter which can be used to adjust the dynamic range of the sensor. **`touch-range`** can be used to read back this parameter.

The third new facility is used to read and write data to the EEPROM on a parameter card. **`aget2`** and **`aset2`** correspond respectively to **`aget`** and **`aset`**, except that they access arrays stored on the card instead of the main Cricket EEPROM. It must be noted however, that there is still only a single **`array`** declaration, so naming of arrays can be slightly confusing. Where on a regular Cricket there is only one array of a given name, on a Tangible Programming Brick, an array name refers to one array that always exists, and a second array, which only exists if a card is inserted. If no card is inserted elements of the second array read back as **`-1`** (negative one).

LEDs on the Tangible Programming Brick are controlled using the Cricket motor commands, in a somewhat counterintuitive manner. This was done partly for expediency, and partly back compatibility with the Blue Dot Cricket.[1] The orange and yellow lights are controlled by motor a, and the green light is controlled by motor b. To light orange

---

1. A regular Cricket Logo programming environment can be used if no Brick-specific commmands are called. If they are, extra descriptors need to be appended to the compiler primitives list in `setup`: `new2? r 0 bus2 r 0 reply2 c 1 aset2 c 3 aget2 r 2 set-touch-range c 1 touch-range r 0`

and yellow together, you use the **brake** primitive. To light an individual LED of the "a" pair, you use the **on** primitive. The direction selects which LED. **thisway** selects the yellow light, and **thatway** selects the red and green lights. The **setpower** primitive sets the brightness of individual LEDs (Note: low power settings cause noticeable flickering).

3.2.2 Application code

The application code for the Tangible Programming Brick is written in Cricket Logo on a Mac or PC and downloaded into each individual Bricks using an IR/bus card. For the dancing cars and microwave oven demonstrations, each Brick ran a very simple program which allowed a stack of Bricks/cards to be scanned in a "bucket brigade" or "shift register" fashion by a Cricket connected to the base of the stack. The microwave oven demo also allowed the Cricket to remotely control the LEDs of each Brick. This was used to indicate which step of the program was being currently executed. Also, a mechanism was provided to remotely right data onto EEPROM cards. This was necessary because the IR/bus card could not be used while a EEPROM card was inserted.

The style of programming I adopted for the Bricks was strongly influenced by technical details of the Cricket bus protocol. Each Brick has two bus ports, a master bus which can communicate to a Brick above or through the card slot, and a slave bus which listens for communication from below, much like the existing Cricket IR receiver. The only way to receive data from above is with the **bsr** instruction, which sends a byte on the master bus, and waits for a byte in reply. The biggest constraint with this arrangement is that the **bsr** instruction only awaits 100 milliseconds for a reply. With ordinary Cricket bus devices, this is not problem. Because their firmware is written in assembly language, they reply quickly, but the application code inside a Brick is written in Logo, which can interpret only about 30 lines of code within the time-out period. This means that after overhead reserved for the dispatch mechanism, very little time remains for a reply to be computed, and it is completely impractical to recursively poll Bricks further up the stack in order to make a reply. In the end, I adopted a two phase scanning mechanism which treated a stack of Bricks as a shift register chain. Once individual pockets in the shift reg-

ister were initialized, each shift operation could be done very quickly (For details see the "Kitchen Brick" section of Appendix B).

3.3 The Tangible Programming Brick Demonstrations

Using a small set of Tangible Programming Brick and parameter cards, I implemented three different scenarios to demonstrate the utility of tangible programming.

3.3.1 Dance Craze Buggies

In the fall of 1998, Rick Borovoy and Fred Martin created the first demonstration of Borovoy's "tradeable bits" technology: the Dance Craze Buggies—toy cars that teach each other how to dance. Borovoy writes:

> Imagine a child teaching her robotic toy a new dance step. Then, when she is playing with a friend, her toy can "teach" this new dance to her friend's toy. Later, her friend can modify this dance a little, and pass it on to another friend. The creator of the dance can check the Net to see how far her dance has spread ("150 toys know my dance!") [6].

The Tangible Programming Bricks were able to neatly complete the picture in Borovoy's story about the cars. His intention all along was that kids would initially teach their car a new dance. What was missing was a compelling, practical way to teach new dances without a computer. To solve this problem we built a "teacher" device (nicknamed the "phaser") that accepted a stack of my Bricks. When the "trigger" was pressed, it taught a car the dance steps specified by the stack of Bricks. Each Brick was labeled with a dance step (big step forward, big step back, little step forward, little step back, and wiggle tires), and accepted parameter cards which either specified the number of repetitions of that dance step, or that the dance step should be performed and reversed ("forward and back"). This demonstration worked beautifully for the Toys of Tomorrow exhibition in May of 1999. The great feature was that we could finally offer a visitor the collection of Bricks and parameter cards, and say "would you like to invent your own dance?"

This experiment raised some interesting questions. One of my motivations for working on tangible programming is the intuition that, if software is more visibly coupled with the devices being controlled, then it will be easier for young children to understand. Software that is invisibly stored inside a computer but the device being controlled is in the world can be truly cryptic. A stack of tangible Bricks on the "phaser" teacher device got closer. And finally a stack of Bricks directly on the dancing car is the most closely coupled, and I postulate is the easiest to understand (See "Robotic vehicles and "spatial" programming" on page 61.).

3.3.2 Kitchen appliances

Modern kitchen appliances such as microwave ovens, bread makers, and even coffeemakers can be programmed and personalized to a limited extent. These limitations come primarily from deficiencies in the user interface. A busy professional might want to communicate a simple program to his microwave oven: defrost for 10 minutes, then cook for 20 minutes, and have everything finished by 5:30. This sort of programming was once common on high-end microwaves, but the sequence of keystrokes required to express this can be confounding. Using tangible programming techniques, this same program can be expressed by assembling three Bricks and three cards:

[Defrost] [:10]
[Cook] [:20]
[Finish by] [5:30]

For the Fall 1999 meeting of the Media Lab Counter Intelligence consortium, I implemented a tangible programming language to control an actual microwave oven. Photographs of the Kitchen Bricks can be found on page 14 and page 15. Details of the software can be found in Appendix B.

3.3.3 Toy trains (revisited)

Although Martin and Colobong's train (described on page 33) demonstration was compelling and complete, I was inspired to go one step further by letting the child choose how the train would react to each signal. In this scenario each signal is given a distinct color, but the meaning of a signal is not predefined. In fact each train is free to respond

differently to a given signal. For every color of signal, each train has a corresponding car of the same color. Placing Tangible Programming Bricks on a car determines the train's behavior when it passes a signal of the same color (see figure on page 37).

Chapter 4 — Discussion

4.1 Design Issues

4.1.1 Communication issues

At runtime, traditional object-oriented programming environments provide a perfectly reliable message passing substrate. But when real objects are involved, and when communication between objects needs to be wireless, a number of issues arise. First and foremost is reliability. Closely related is predictability. The last thing we want in a toy or learning environment is the frustration and uncertainty of unreliable communications. Radio communication has the advantage that it is omnidirectional, but it is expensive and difficult to engineer. Infrared communication is inexpensive and uses small parts, but neither transmitters nor receivers are omnidirectional. Typically their broadcasts and reception patterns resemble a 30 degree wide cone. Infrared transmitters can be "ganged together" and arranged in a circular array, but receivers cannot. Tomy solves this problem in their infrared controlled Tomica World toy train system by using a conical mirror and an upward facing receiver to achieve a 360 degree reception pattern. Note that this does not solve line of sight issues.

In certain situations, global, omnidirectional communication is not desirable. In the train and signal scenario (discussed in sections 2.6.5 and 3.3.3), we want the train to exhibit certain behavior when it "sees" a signal tower. In this case, short range, directional communication is more desirable.

From the user's standpoint, a disadvantage of using infrared light is that it is limited in range, directional, and invisible.[1] This makes debugging difficult unless we provide a device to make the beam visible in some way. Rather than a debugging device which simply shows where a beam is projected it would be preferable to have a device which shows the messages being transmitted by a particular beacon.

---

1. Radio waves are also invisible, but it is easier to understand them because they are omnidirectional, at short range they go through most objects, and for small setups, range is not an issue.

Leading up to the design of the Tangible Programming Brick, I spent months researching power sources and connectors so the Brick would have just the right size and feel.

4.1.2 Power

During the development of the second generation of Kwin Kramer's Beads (see [27] for details), I investigated how to provide power to the beads so that we wouldn't have to put a battery in each one. I considered capacitive coupling, inductive coupling (transformers), and DC coupling (connectors). In the end, I rejected capacitive and inductive coupling, mostly for health reasons (people get nervous when high voltage electrostatic and high-power radio frequency fields are near their children). DC coupling appears to be the only viable means to provide external power, but to do a good job of providing reliable power requires good connectors.

4.1.3 Connectors

After extensive study of commercially available (as opposed to custom-designed) connectors systems, I have concluded that connectors that are good for children are bad for industrial applications, and vice versa. Industrial connectors which rely on friction to stay in place generally require too much force to insert and disconnect. Low insertion force connectors do exist, but they are expensive, and generally require a two-step process to connect or disconnect (insert + lock, unlock + remove). The most attractive connector systems appeared to be ones where the connector does not provide the mechanism for maintaining contact. An everyday example is the connector system on laptop, cell phone, and camcorder batteries. Here, a set of gold plated springs are held against a set of gold plated pads are held against each other by a latch mechanism built into the case of the portable device. This is the class of connectors I chose for the Tangible Programming Brick. Commercially available battery connectors were too large (contacts on 0.100" spacing). But, the ISO-7816 "smart card" the connector system proved to be perfect. The male connector provides 8 pins in a 4x2 grid, and the female connector is just a gold plated PC board pattern familiar to Europeans. I used this connector system both for Brick to Brick (stacking) connections, and for the card slot (a more traditional use for the ISO connector).

4.1.4 Signaling and power: To multiplex or not?

A very attractive prospect would have been to use a two contact connector like LEGO's standard 2x2 electric plate. This would require multiplexing power and signaling (data). I wanted two kinds of signaling: neighbor to neighbor and global signaling (bus). Multiplexing a global power and data bus is not difficult. Several schemes are in commercial use. LEGO uses a time division scheme for its rotation and reflectance sensors. Power is provided for part of the time, and data is returned in analog form during the remainder of each cycle. Other systems use DC for power and superimpose an AC signal for data. Reconciling neighbor to neighbor communication and bused power is another matter. Theoretically it is possible to de-couple Bricks from each other using diodes, but the voltage drop across each diode makes voltage regulation problematic. In the end, I chose to use four conductors: power, ground, global, and neighbor to neighbor communication. The "smart card" connector has eight conductors. I use the redundant conductors to allow 180 degree rotation of Bricks with respect to each other. In other words, electrically there is no wrong way to stack Bricks. Ninety degree rotations are not allowed, but they are prevented by mechanical means (the Bricks don't fit together this way).

Other signaling in power arrangements are possible. For example power could be distributed using a two conductor connector, and neighbor to neighbor signaling could be accomplished using infrared signalling (as in Kwin Kramer's tiles). I chose not to use this scheme to reduce space and part counts, but we may consider this system in the future.

4.1.5 Optical issues

The three LEDs in each Brick need to be visible from a large range of angles. For cosmetic reasons I chose to use "light pipes" to guide light from the surface-mount LEDs (which faced downward on the printed circuit board) to the exterior. Having separate molded plastic pieces for the light pipes would have provided more optical flexibility, but at LEGO's request I settled for a two piece case design, with the transparent bottom piece doubling as a light pipes. The design as it was produced is effective at directing light out the front of the Brick, but in actual use, it doesn't work very well. This is because a stack of Bricks is frequently viewed from above (at perhaps a 45 degree angle). In the next revi-

sion (if there is one), we plan to use an extra prism in front of the mirror to direct light slightly upward as it leaves the light pipe.

4.2 A Design Critique

Admittedly there are a number of valid criticisms of the (first generation) Tangible Programming Bricks:

They are uniform in shape. Shape is important semantic cue. When selecting and sorting objects uniform shades are more difficult to work with. Shape can also suggests function and program structure.[1]

They can only be stacked in one dimension. Screen based visual and textual programming languages are traditionally two-dimensional from the user's perspective, even though they are frequently "parsed" and interpreted as one-dimensional strings by compilers and other language tools. A specially designed "bridge" card would allow limited two-dimensional structure using the current Bricks. This way two stacks could be joined side-by-side.

They are expensive. The toy industry and school systems alike are extremely cost-conscious. Although I have eliminated one major source of cost by not requiring batteries in each Brick, the microprocessor and the connector system are expensive—too expensive for consumer and educational products consisting of dozens of Bricks. Even if one were to reduce the part count and use in expensive "gob of glue" chip packaging, the high-quality connector system required may still make productization infeasible.

4.3 User testing

At this writing I have not performed any *formal* user testing. However, Rick Borovoy and I were able to make *informal* observations of dozens of first-time users who visited our Dance-Craze Buggies demonstration. With each new group of participants, we gave a

1. Henry Lieberman suggests that Bricks representing nested commands like **repeat** and **end repeat** have offset connectors. This way the loop "body" will be indented automatically.

brief overview of our system, and giving an example of its use. We showed how the Bricks snap together, and we showed how the parameter cards slide in the slot on the side each Brick then we offered a handful of Bricks to a participant so they could create their own dance for the dancing cars. Without exception, we observed that the participants were able to effectively assemble a stack of Bricks and insert parameter cards to create a dance. Because the dances were highly creative and nature, there was no such thing as a "wrong dance," and consequently there was no opportunity to study whether or not users were able to correctly execute a goal.[1] One incidental observation we made was that most participants used every available Brick in their dance. I speculate that we would not have seen this behavior if the participants had a larger selection of Bricks to choose from (i.e. larger than a reasonable length dance).

During the "Kitchen Brick" demonstration there was almost no "audience participation," however the middle-school-aged son of one of my colleagues asked if he could try his hand at programming the microwave using the Bricks. He asked good questions and quickly discovered a shortcoming of my prototype: I had to explain that there were two types of bricks, ones that required parameter cards, and ones that didn't, but because my Bricks all had a functioning card slot, there was no way for him to determine this on his own. This suggests that I should have taped over the slot in the cases where no card was needed, thus removing the inappropriate affordance.

1. Fred Martin comments that this is "not necessarily a problem. Who says users have to have an explicit goal? If anything I see this as a big feature. Conventional programming systems don't typically allow program 'doodling.' If your system does, that's a big win."

Chapter 5 — Future Work & Conclusion

5.1 The near term

5.1.1 Formal user testing

My intention from the very beginning of this project was to build several dozen of the Tangible Programming Bricks and to test them with children between the ages of 7 and 14. I did a prototype run of 8 Bricks, but manufacturing difficulties and time constraints prevented us from embarking on the production run of 30 to 40 Bricks. Time constraints alone prevented us from performing limited user testing and the shortage of Bricks would have made it difficult to conduct the full-scale testing I had intended from the outset. Given additional time I would certainly do a production run and full user testing.

In this document I have described a number of programming metaphors, and it would be especially revealing to compare children's experiences with different types of tangible programming languages using the Tangible Programming Bricks I have already implemented. It would also be very interesting to study adult subjects across several age ranges.

5.1.2 Expanding beyond linear stacks

A major shortcoming of the current design as implemented is its strictly one-dimensional nature. Implementing a Brick which connects to neighbors in two or three dimensions is a natural next step, but one which is replete with design issues which would take months to explore. Another shortcoming of this design is the decision I made about power sources. To make individual Bricks and small assemblies "come alive" we may want to reconsider including batteries or providing some other way of powering Bricks while they are being handled and before they had been assembled into a program.

5.1.3 Issues of size and shape

The Tangible Programming Bricks were designed to be small based on the theory that you want be able to build programs out of many Bricks. For professional use by adults

the ability to construct large programs it is attractive, but for children learning about programming short programs and especially programs of low complexity are likely to be most common. I have considered but not pursued the implementation of larger Bricks suitable for preschool children and toddlers. This would be a natural course of follow-up research.

With only one shape of Tangible Programming Brick the user's ability to differentiate between among Bricks is constrained to color, icon, and text. I believe it would be valuable to explore Bricks of different shapes and Bricks with differently shaped connectors based on the semantics of the particular tangible programming language being implemented, much like the Logo Blocks system [49], which uses a "puzzle pieces" metaphor for connecting blocks, and makes a distinction between control flow and data flow connections.

5.2 Alternate implementation mechanisms

I explored a number of mechanisms for implementing tangible programs. Most of my efforts were spent pursuing building blocks containing an embedded microprocessor, but no batteries, and which communicated with each other and distributed power through electrical connectors. This system is expensive, surprisingly not due to the cost of the microprocessor, but because of the cost of the connectors and packaging.

5.2.1 Bar Codes

My original proposal was to place bar codes on the backs of plastic tiles, which would have a "feel" similar to dominos, and to scan these bar codes using a desktop augmented with a bar-code scanner. Flatbed scanners are becoming inexpensive consumer products, so the bar-code system is relatively inexpensive, but this system isn't nearly as portable as the Tangible Programming Bricks, because users are restricted to working on a fixed work surface containing an embedded scanner.

5.2.2 RFID tags

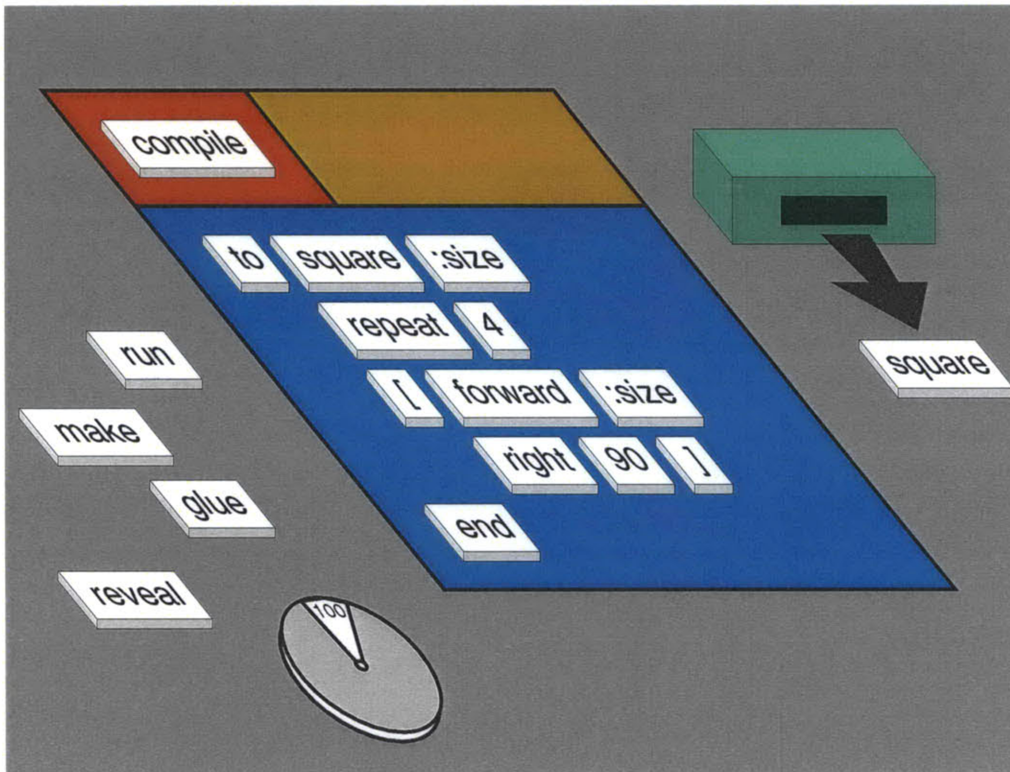
Radio Frequency Identification (RFID) tags offer an inexpensive, portable implementation strategy. The holy grail of RFID research is a tag which can represent a nontrivial number of bits of information (say, 8 or more bits) and which costs pennies per tag. For tangible programming purposes the tag needs to be able to sense the identity of neighboring tags. This way the topology of a tangible program can be determined using a single tag reader. Rich Fletcher and the MIT Media Lab Tangible Media Group have demonstrated a system using glass bottles as a metaphor for "tangible information containers." Most recently they used glass bottles outfitted with simple RC resonant tags wound around the necks and bottle-stoppers with embedded ferrite rods that modified the resonant frequency of the bottle's tag [22]. A table outfitted with an inexpensive tag reader [13] can sense the presence of particular bottles and determine whether each bottle is opened or closed (one bit).

To determine the suitability of this technology for tangible programming, I performed some informal experiments that confirmed that an LC resonant tag can be used to sense the identity of its neighbor. This allows a tag reader to determine the topology of an entire assembly of tiles or bricks using only one antenna. In my experiments, the neighbor's identity was encoded by the mass and geometry of a piece embedded of ferrite. Although this demonstrates a proof of concept, Fletcher estimates that this technique will not scale beyond approximately 8-16 unique tags. This means that the "fashion designer" database query (introduced on page 39) might be feasible, but expressing a seven-digit phone number would not.[1]

5.2.3 Specialized work surfaces

If we are willing to place our tiles on a specialized surface, a number of technical benefits can be derived. For example, a mat of conductive Velcro can be used for two purposes: it can serve as an anchor for keeping assemblies of tangible tokens or tiles from moving

1. For the tag reader to unambiguously determine the sequence of digits, this would require 7×10 = 70 uniquely tagged tiles (e.g. it is not sufficient for all 5s to have the same ID).



Block "printer" expands tangible vocabulary

around on the work surface, and it can provide a single strong electrical path for electrostatically coupled RFID tags.

5.2.4 Reusable/Re-printable blocks

Up until this point I have talked about tangible programming applications with fixed vocabularies. The original proposal for this thesis was to implement a system which included a "block printer" which could make new blocks (or recycle old ones) on demand. The presence of a block printer can significantly enhance the utility of a tangible programming system. In addition to simply creating new vocabulary as needed, the block printer can engage the user in a sort of dialogue, where the user constructs new structures and the block printer "creates" structures in response, closing the loop. Rewriting the "machine readable" part of an old block to create a new one is easy. Recycling of the "human readable" part is a bit more of a challenge. If power and expense are no issue, LCD displays are an option. There are even by stable LCDs which only require

power to change state. Electronic ink (a.k.a. "E-ink") is an area of active research but there exists a viable low-tech solution: thermochromic inks and plastics. These are materials which can be erased by cooling (e.g. using a Peltier junction) and printed or reprinted using a garden-variety thermal printer.

5.3 Applications

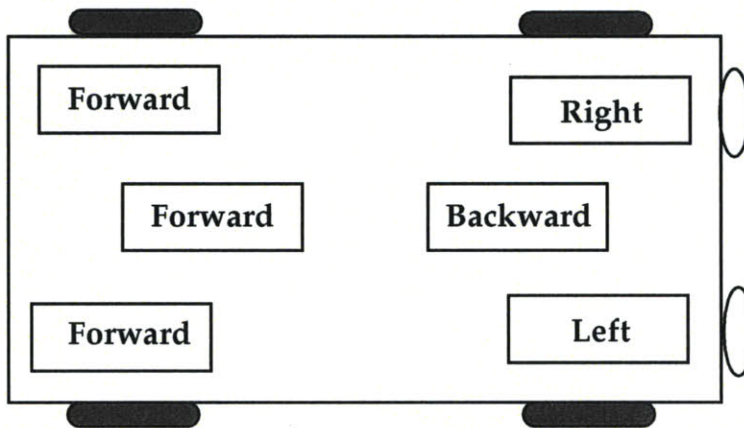
5.3.1 Music synthesizers

Flip to the back of the liner notes in the CD of a modern keyboardist like Chick Corea, and you are likely to find, along with credits for the session musicians, the name of someone acknowledged for "synthesizer programming." This is someone who combines the talents of good sound designer and a technician who has spent countless hours mastering the confounding user interfaces of modern synthesizers. Most musicians who use synthesizers in novel ways run up against complex devices and frustrating user interfaces.

Synthesizers are powerful reconfigurable devices. In the early days of electronic music, analog synthesizers were collections of signal-processing modules that were physically "programmed" with patch cords. By manipulating tangible modules that can be assembled into a physical representation of the synthesizer's signal chain, musicians will have greater understanding and control over their music-making machines: a throwback to the analog days, but without the noise and hum of real patch cords.

5.3.2 Robotic vehicles and "spatial" programming

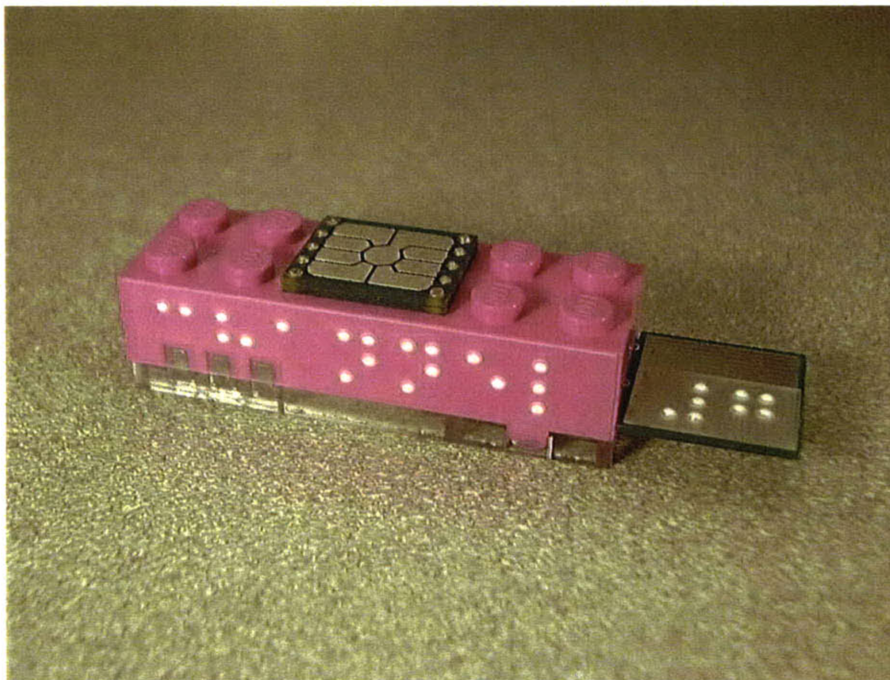
In the spirit of Braitenberg [7] and reminiscent of early LEGOsheets applications, this robotic vehicle scenario allows children to experiment with emergent behavior [44]. The vehicle is equipped with two motors and bump sensors on the 4 corners. Also at each corner is a tangible programming "pad." Placing programming Bricks on a corner determines what the vehicle will do when that corner bumps into something. Two additional programming sites are provided so the vehicle can be programmed to do something more appropriate when both front or both rear bump sensors are triggered at about the



same time. The vehicle in shown here could do a pretty good job of keeping itself from getting stuck with only very simple rules.

5.3.3 Tangible interfaces for the visually impaired

Human computer interfaces for the visually impaired have been available for years, and have made it possible for blind people fluent in Braille to get jobs as programmers and computer operators. A Braille "terminal" consists of a one or two line mechanical "dis-



A Braille Brick: "Channel 7"

play" typically with 40 characters per line. Each character is formed by six or eight solenoid-controlled pins. This mechanism is typically used to simulate a multi-line "TV typewriter" by providing a knob or slider that is used to select which line to "look at."

The heyday of Braille terminals was in the MS-DOS era of command-line interfaces and text-based user interfaces. Commercial software designed for the general public could be used unmodified by the blind. Window systems and graphical user interfaces, while a great advance for the sighted, was a serious setback to blind computer users, who were left without any way of using modern graphical software. Tangible user interfaces can solve this problem directly, but they do offer some exciting new interaction paradigms for Braille readers.

For limited vocabulary applications, sentences and computational expressions can be assembled out of Bricks labeled with Braille words. A Braille "block printer" to create new vocabularies or engage in a dialog with the user a cost it is significantly less than that of a Braille terminal.

5.4 Conclusion

In this thesis I have presented technical details of the Tangible Programming Brick, a stackable, programmable, electronic building block that I designed to conduct research in constructive tangible user interfaces in general and tangible programming languages in particular. I have discussed potential applications for this technology, and I have described two domain-specific languages that we implemented using a set of my stacking Bricks, one language for controlling toy cars and another for controlling microwave ovens. Although I conducted no formal user testing, I did have the opportunity to informally observe dozens of first-time users who, after only a few minutes of instruction, successfully used my system to control the toys and kitchen appliances. I expect that further testing will confirm my hypothesis that the tangible programming techniques developed in this thesis will lead to user interfaces that are easier to learn and easier to use when compared to the graphical user interfaces of general-purpose computers and the ad hoc, application-specific user interfaces found on today's digital appliances.

The Tangible Programming Brick is a general-purpose research tool based on the Cricket architecture that can be repeatedly programmed using the Cricket Logo software development environment. It features an innovative connector system that is both electrically reliable and easy to assemble and disassemble by hand, and a number of features which make it suitable for rich interaction with the user. Although this technology is probably too expensive for immediate commercialization, we have begun to experiment with techniques which promise to reduce costs to pennies per Brick (or tile).

My original motivation for pursuing research in tangible programming was to lower the age at which children can begin to learn about programming. It became clear in the course of my research, that tangible programming techniques have broad applicability beyond educational toys, to the control of everyday digital devices. Recently it has been suggested that the most exciting prospects for this research may be in the area of human computer interfaces for the visually impaired, where graphical user interfaces are completely ineffective and where tactile, constructive tangible interfaces hold great promise as a medium for communication and expression.

Bibliography

1. Abelson, H., Sussman, G. J., with Sussman, J., *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill, 1985. Second Edition, 1996.

2. Abelson H., diSessa, A., *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*, MIT Press, 1981.

3. Anagnostou, G., Dewey, D., and Patera, A., Geometry-defining processors for engineering design and analysis, in *The Visual Computer*, 5, pp. 304-315, 1989.

4. Anderson, D., Frankel, J., Marks, J., et al., Building Virtual Structures with Physical Blocks (demo description), to appear in *Proceedings of UIST'99*, 1999.

5. Begel, A., *LogoBlocks: A Graphical Programming Language for Interacting with the World*, S.B. Thesis, MIT Department of Electrical Engineering and Computer Science, `http://abegel.www.media.mit.edu/people/abegel/begelaup.pdf` 1996.

6. Borovoy, R., *Dance Craze Buggies*, `http://el.www.media.mit.edu/people/borovoy/cars/` 1998.

7. Braitenberg, V., *Vehicles, experiments in synthetic psychology*, MIT Press, Cambridge, Massachusetts, 1984.

8. Brooks, R. A., *A Robust Layered Control System For a Mobile Robot*, AIM-864, MIT AI Lab, 1985.

9. diSessa, A., Notes on the Future of Programming: Breaking the Utility Barrier, *User Centered System Design*, Norman, D. A., and Draper, S. W., eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

10. Drescher, G. L., Object-Oriented Logo, *Artificial Intelligence and Education*, Ablex Publishing, Norwood, NJ, pp. 153-165, 1987.

11. Ferrell, C., *Robust Agent Control of an Autonomous Robot with Many Sensors and Actuators*, AITR-1443, MIT AI Lab, 1993.

12. Fitzmaurice, G., Ishii, H., and Buxton, W., Bricks: Laying the Foundation for Graspable User Interfaces, in *Proceedings of CHI'95*, pp. 442-449, ACM Press, 1998.

13. Fletcher, R., *A Low-Cost Electromagnetic Tagging Technology for Wireless Identification, Sensing, and Tracking of Objects*, S.M. Thesis, MIT Media Lab, 1997.

14. Gershenfeld, N., *When Things Start to Think*, Henry Holt, New York, 1999.

15. Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O., and Repenning, A., LEGOsheets: A Pule-Based Programming, Simulation and Manipulation Environment for the

LEGO Programmable Brick, *Proceedings of IEEE Symposium on Visual Languages*, Darmstadt, Germany, pp. 172-179, IEEE Computer Society Press, September 1995.

16. Gorbet, M., Orth, M. Ishii, H., Triangles: Tangible Interface for Manipulation and Exploration of Digital Information Topography, *Proceedings of CHI '98*, ACM Press, 1998.
17. Green, T. R. G., Petre, M., Usability Analysis of Visual Programming Environments, *Journal of Visual Languages and Computing*, 7(2), pp. 131-174, 1996.
18. Hansen, W. J., "The 1994 Visual Languages Comparison," in *1994 IEEE Symposium on Visual Languages*, pp. 90-97, IEEE Computer Society Press, Los Alamitos, CA, 1994.
19. Harvey, B., *Computer Science Logo Style*, 2nd edition (3 volume set), MIT Press, 1999.
20. Hillis, D., *The Pattern on the Stone: the simple ideas that make computers work*, Basic Books, New York, 1999.
21. Hutchins, E., Hollan, J., & Norman, D. A., Direct Manipulation Interfaces. In Norman, D. A. & Draper, S. W., Eds., *User Centered System Design: New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.
22. Ishii, H., Fletcher, R., et al., MusicBottles, in *Abstracts and Applications of SIGGRAPH'99, Emerging Technologies*, ACM Press, 1999.
23. Ishii, H., and Ullmer, B., Tangible Bits: Toward Seamless Interfaces between People, Bits and Atoms, *Proceedings of Conference on Human Factors in Computing Systems (CHI '97)*, pp. 234-241, ACM, Atlanta, March 1997.
24. Jagadeesh, J., and Wang, Y., LabVIEW (product review), in *Computer*, February 1993.
25. Kahn, K., Drawings on Napkins, Video-game Animation, and Other Ways to Program Computers, *Communications of the ACM*, vol. 39, no. 8, pp. 49-59, August 1996.
26. Koch, C., Upitis, R., Is Equal Computer Time Fair for Girls? Potential Internet Inequities. *Proceedings of the INET'96 Conference*, Internet Society, Montreal, Canada, `http://www.isoc.org/isoc/whatis/conferences/inet/96/proceedings/c1/c1_3.htm` June 1996.
27. Kramer, K. H., *Moveable Objects, Mobile Code*, S.M. Thesis, MIT Media Lab, 1998.
28. Macaulay, D., *The New Way Things Work*, Houghton Mifflin, Boston, 1998.
29. Maes, P., ed., *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, MIT Press, 1991.
30. Martin, F., *Children, Cybernetics, and Programmable Turtles*, S.M. Thesis, MIT Department of Mechanical Engineering, 1988.

31. Martin, F., Resnick, M., LEGO/Logo and Electronic Bricks: Creating a Scienceland for Children, *Advanced Educational Technologies for Mathematics and Science,* (David Ferguson, ed.), Springer-Verlag, Berlin, Heidelberg, 1993.

32. Martin, F., *The Programmable Brick Handbook*, MIT Media Laboratory, 1997.

33. Martin, F. Colobong, G. L., Resnick, M., *Tangible Programming with Trains,* `http://el.www.media.mit.edu/projects/trains/` 1998.

34. Martin, F., *personal conversations*

35. Miller, D., KISS Institute for Practical Robotics `http://www.kipr.org/`

36. Norman, D. A., *The Psychology of Everyday Things*. Basic Books, New York, 1988.

37. Norman, D. A., *The Invisible Computer*, MIT Press, Cambridge, Massachusetts. 1998.

38. Papert, S., *Mindstorms: children, computers, and powerful ideas*, Basic Books, 1980.

39. Parigot, D., and Mernik, M. eds., *Second Workshop an Attribute Grammars and their Applications, WAGA'99,* Amsterdam, The Netherlands, INRIA rocquencourt, 1999.

40. Perlman, R., Using computer technology to provide a creative learning environment for preschool children, Logo Memo no. 24, AI Memo 260, MIT AI Lab, May 1976.

41. Post, R., *E-Broidery,: An Infrastructure for Washable Computing*, S.M. Thesis, MIT Media Lab, 1999.

42. Repenning, A., Sumner, T., Agentsheets: A Medium for Creating Domain-Oriented Visual Languages, *IEEE Computer*, v. 28 no. 3, pp. 17-25, 1995.

43. Repenning, A., and Ambach, J., Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition, and Sharing, in *Proceedings of Visual Languages 1996*, IEEE Computer Society, 1996.

44. Resnick, M., *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press, 1994.

45. Resnick, M., Behavior Construction Kits, *Communications of the ACM*, v. 36, No. 7, pp. 64-71, July 1993.

46. Resnick, M., Martin, F., Berg, R., Borovoy, R., Colella, V., Kramer, K., Silverman, B., Digital Manipulatives: New Toys to Think With, *Proceedings of CHI'98*, ACM Press, 1998.

47. Resnick, M., Berg, R., Eisenberg, Turkle, S. M., Martin, F., *Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Instruments*, Research Proposal, 1997.

48. Resnick, M., Eisenberg, M., Berg, R., Martin, F., *Learning with Digital Manipulatives: A New Generation of Froebel Gifts for Exploring "Advanced" Mathematical and Scientific Concepts,* Research Proposal, 1999.

49. Resnick, M., Silverman, B. S., Begel, A., Martin, F., Welch, K., Logo Blocks, `http://fredm.www.media.mit.edu/people/fredm/projects/cricket/logo-blocks/` 1998.

50. Shneiderman, B., Direct Manipulation: A Step Beyond Programming Languages, *Human-Computer Interaction: A Multidisciplinary Approach,* Baecker, R. M., and Buxton, W. A. S., Eds., pp. 461-467, Morgan Kaufmann Pub's, Los Altos, CA, 1989.

51. Smith, D. C., KidSim: Programming Agents Without a Programming Language, *Communications of the ACM,* v. 37, No. 7, pp. 55-67, July 1994.

52. Soloway, E., and Spohrer, J., *Studying the Novice Programmer,* Lawrence Erlbaum, Hillsdale, NJ, 1989.

53. Steele, G. L., and Sussman, G. J., Scheme: An interpreter for the extended lambda calculus, Memo 349, MIT Artificial Intelligence Lab, 1975.

54. Suzuki, H., Kato, H., AlgoBlock: a Tangible Programming Language, a Tool for Collaborative Learning, *Proceedings of 4th European Logo Conference,* pp. 297-303, Athens, 1993.

55. Suzuki, H., Kato, H., Interaction-Level Support for Collaborative Learning: AlgoBlock–An Open Programming Language, *Proceedings of CSCL'95,* 1995.

56. Tufte, E. R., *Envisioning Information,* Graphic Press, Cheshire, Connecticut, 1990.

57. Ullmer, B., *Models and Mechanisms for Tangible User Interfaces,* S.M. Thesis, MIT Media Lab, 1997.

58. Ullmer, B., Ishii, H., and Glas, D., mediaBlocks: Physical Containers, Transports, and Controls for Online Media, *Proceedings of SIGGRAPH'98,* ACM Press, 1998.

59. Ullmer, B. and Ishii, H., Emerging Frameworks for Tangible User Interfaces, unpublished, submitted to *CHI'00,* 1999.

60. Umaschi, M. SAGE Storytellers: Learning about Identity, Language and Technology, *Proceedings of ICLS '96,* pp. 526-531, AACE, 1996.

61. Underkoffler, J., Ishii, H., Illuminating Light: An Optical Design Tool with a Luminous-Tangible Interface, *Proceedings of CHI'98,* pp. 452-549, ACM Press, 1998.

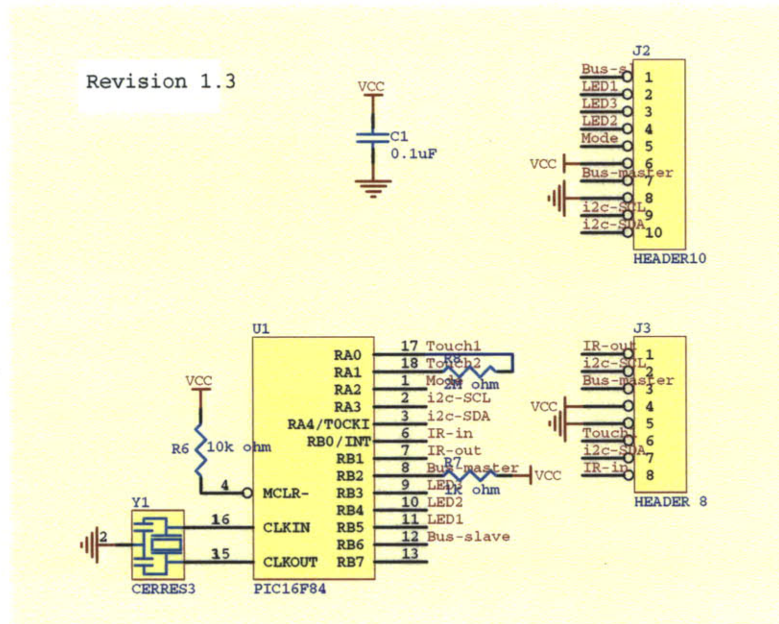
62. Weiser, Mark. The Computer for the Twenty-First Century, *Scientific American,* September 1991. pp. 94-104.

63. Zimmerman, T. G., Personal Area Networks: Near-field intrabody communication, *IBM Systems Journal,* Vol. 35, Nos. 3&4, 1996.

Appendix A — Schematics and Drawings

A.1 Schematics

A.1.1 6x2 Brick – Top board

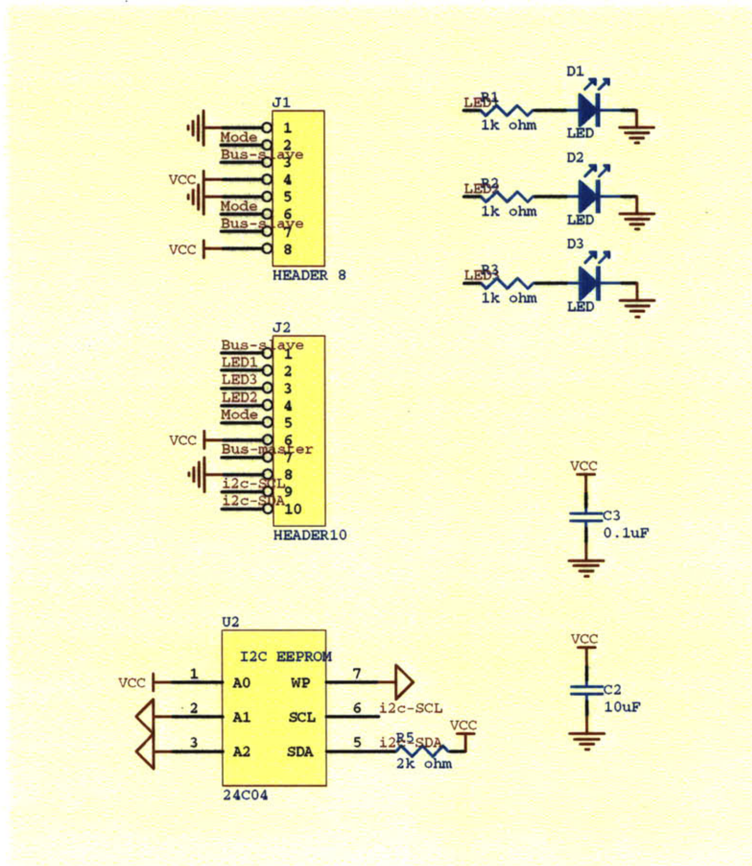


Notes:

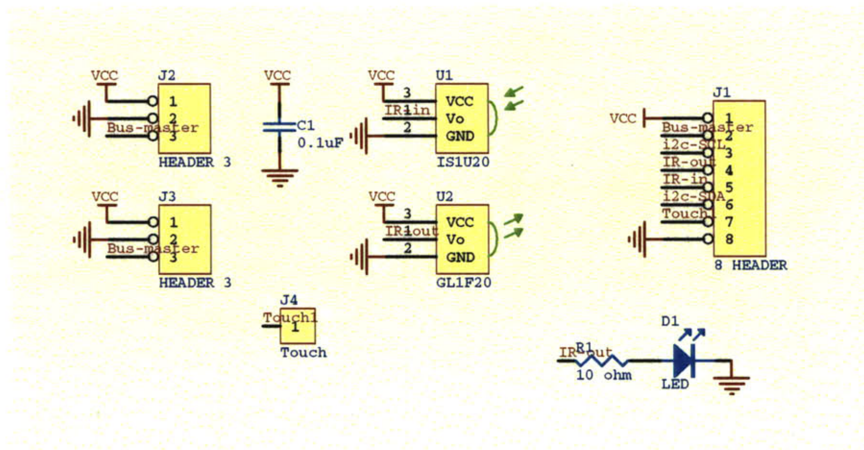
The two boards inside the 6x2 Tangible Programming Brick are connected using 10 free-standing header pins. In the schematics for the top and bottom board, these pins are grouped conceptually as a single "HEADER 10." The "smart card" connector at on top of the brick is connected to the top and bottom boards through 4 of these 10 pins (VCC, GND, Mode, and Bus-master).

The "HEADER 8" (Top Board J3) is the connector for the card slot. On the schematics, the pin numbers do not directly correspond to the mating connectors on the cards. Clearly this is confusing, but there is a practical reason why this became so. The "tops" of the cards inside the Brick actually face down, and this invalidated the pin numbering on the JST-ICC connector. This discrepancy became evident at the 11th hour, and it was easier to change the schematic than to change the component library.

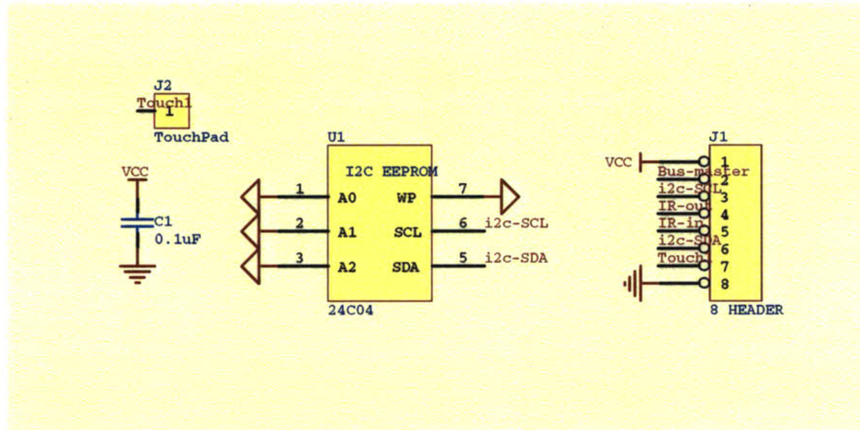
A.1.2 6x2 Brick – Bottom Board



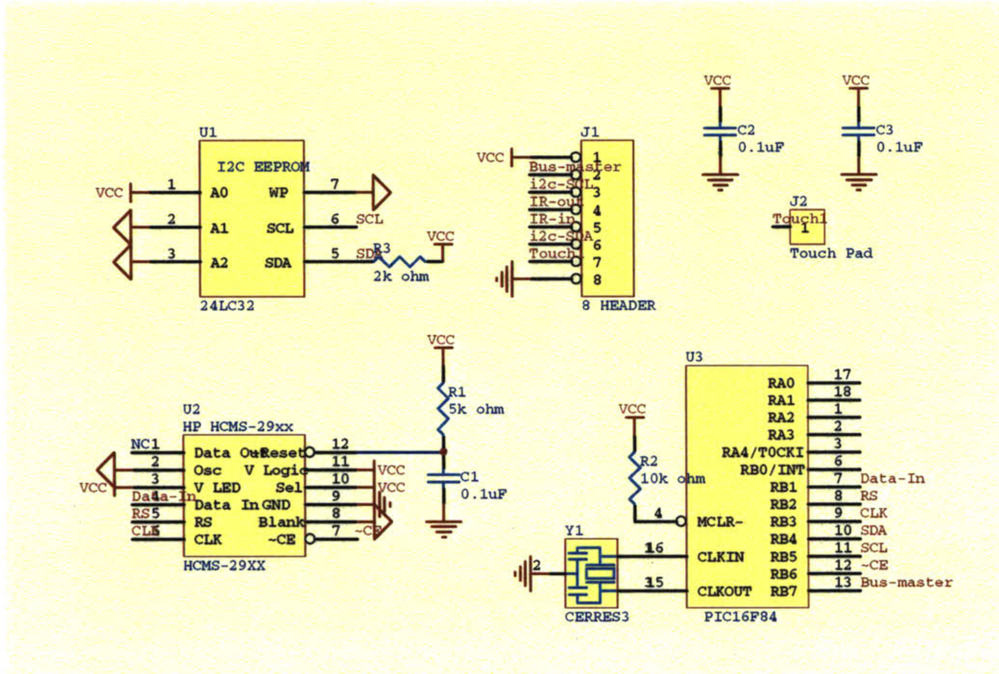
A.1.3 IR/Bus Card



A.1.4 EEPROM Card



A.1.5 Display Card



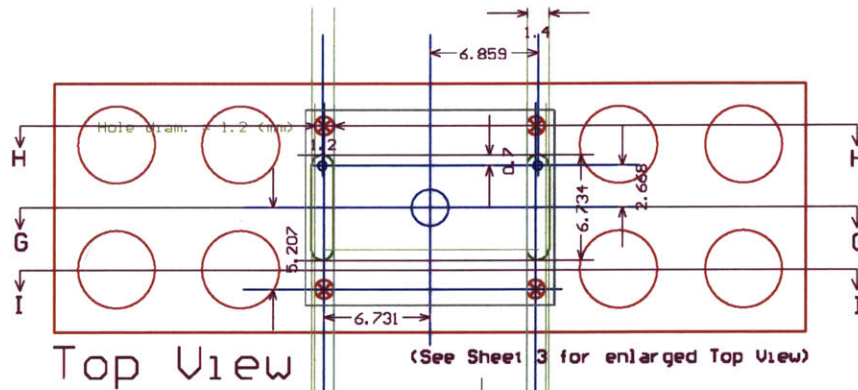
A.2 Engineering Drawings for LEGO Plastic Casing

The following three pages are the engineering drawings I sent to LEGO.

6x2 Tangible Programming Brick (Sheet 1 of 3)

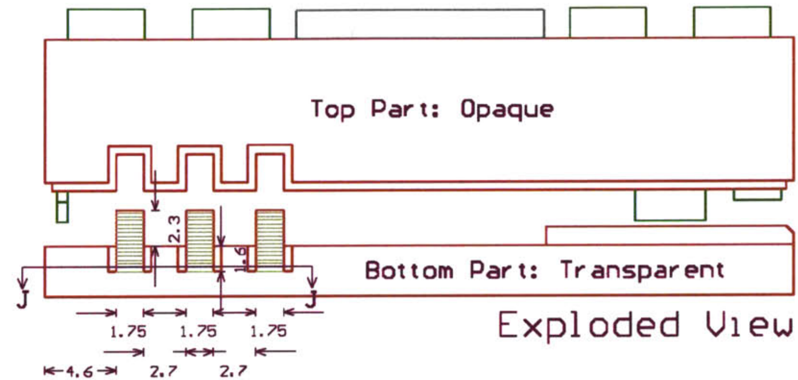
April 22, 1999 02:15

Tim McNerney <mc@media.mit.edu>

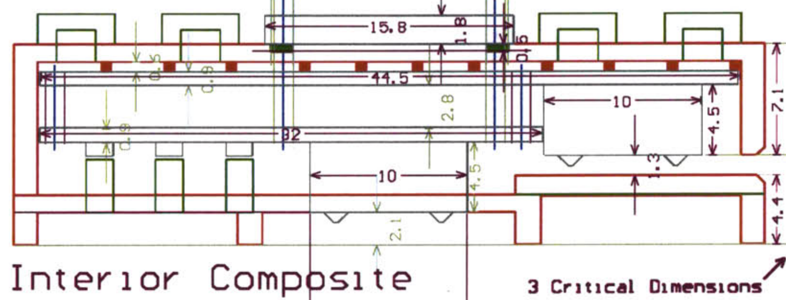


Top View

(See Sheet 3 for enlarged Top View)

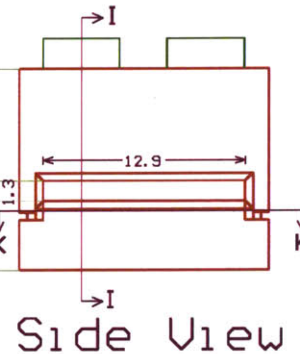


Exploded View

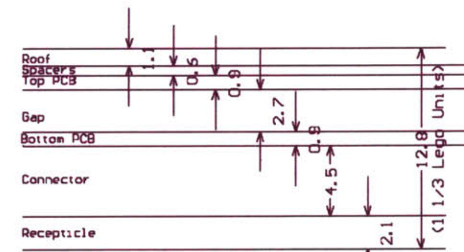


Interior Composite

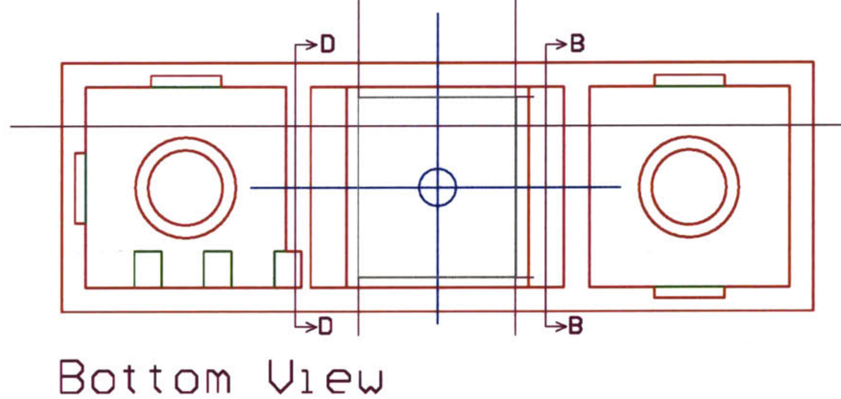
3 Critical Dimensions



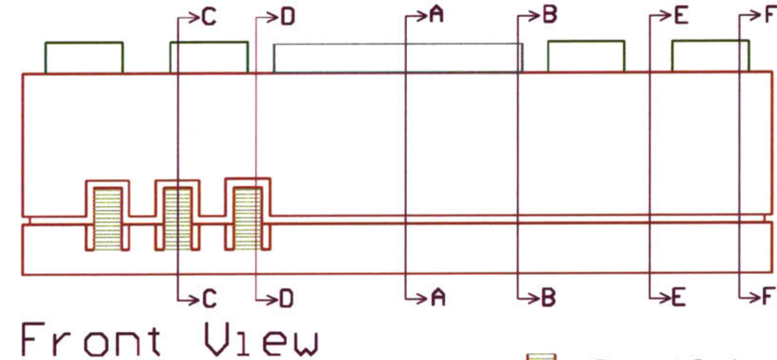
Side View



Elevations



Bottom View

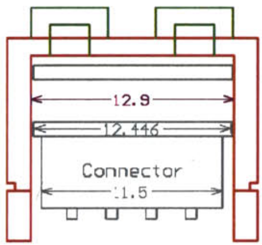


Front View

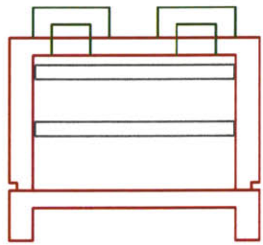
= Frosted Surface

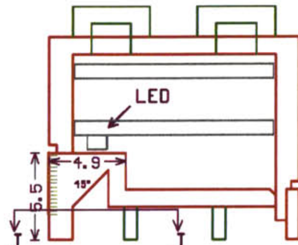
6x2 Tangible Programming Brick (Sheet 2 of 3)
Tim McNerney <mc@media.mit.edu>

April 22, 1999 02:15

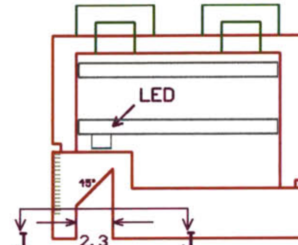


Section A

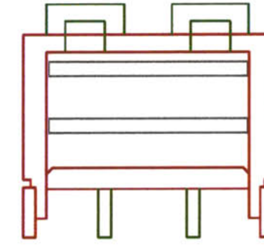

Section B

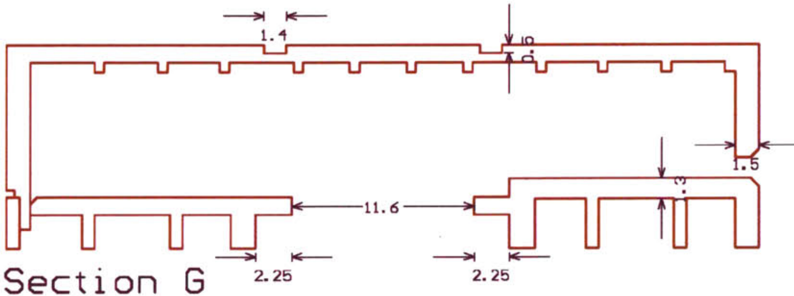


Section C

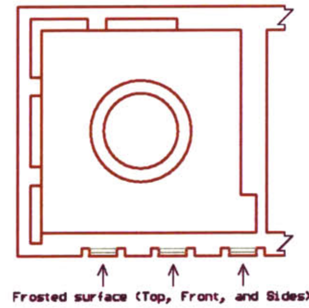


Section D


Section E

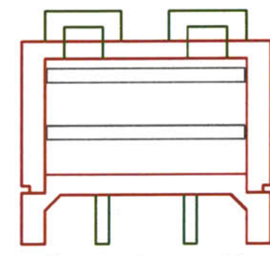


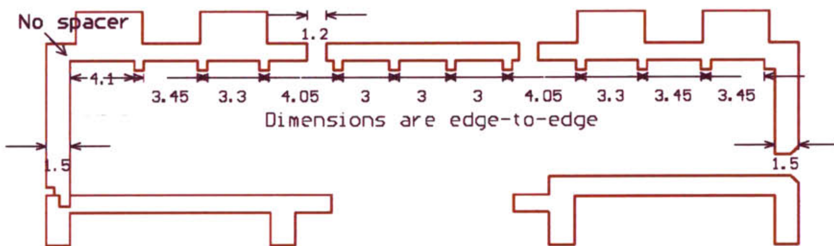
Section G



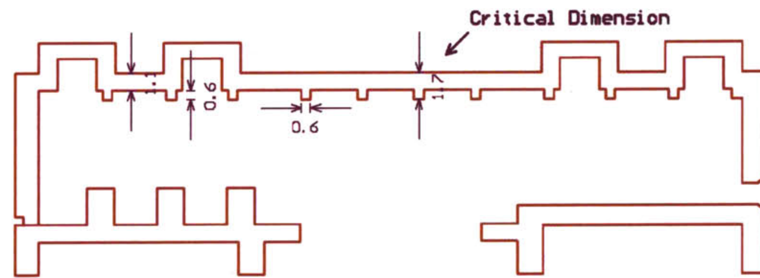
Section J

Section K

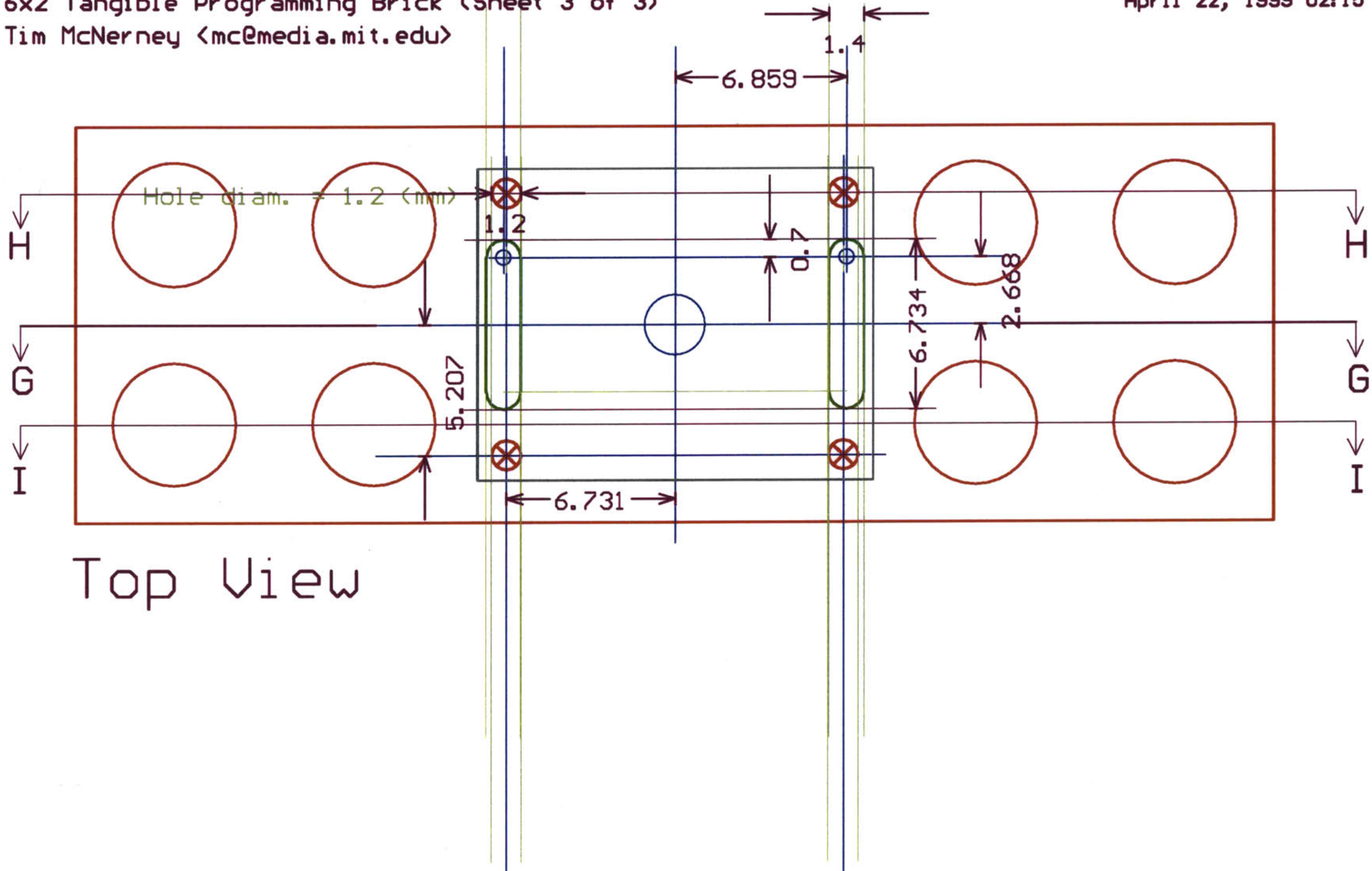

Section F



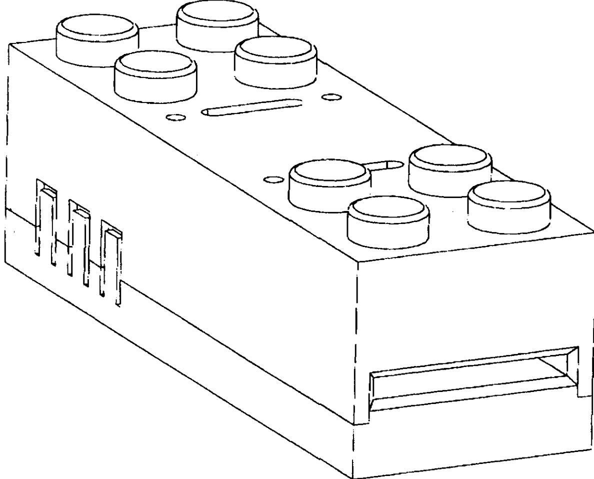
Section H

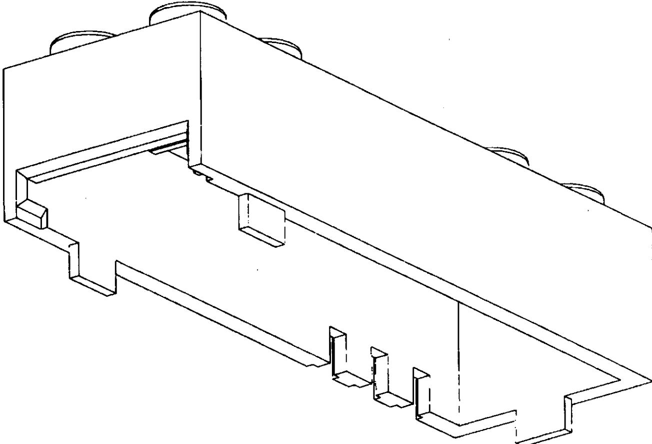


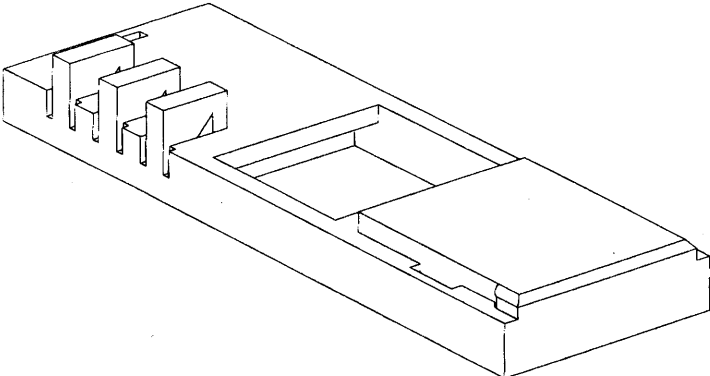
Section I


6x2 Tangible Programming Brick (Sheet 3 of 3)
Tim McNerney <mc@media.mit.edu>
April 22, 1999 02:15
1.4
6.859
Hole diam. = 1.2 (mm)
1.2
0.7
2.668
6.734
5.207
6.731
H
G
I
H
G
I
Top View

A.3 Proof Drawings faxed back to me by LEGO







Appendix B — Software

B.1 Kitchen Brick (Logo)

```
global [bucket counter temp]
array [dummy 2 brick-type 2 card-param 2 have-card? 2]

to kitchen-brick
  aset have-card? 1 1 ;true
  startup
  loop [
    waituntil [new2?]
    dispatch bus2 ]
end

to dispatch :bus2
  ;display :bus2
  if (:bus2 = $195) [ ;are you there?
    reply2 0 ;whereas 255 = no
    stop ]
  if (:bus2 = $196) [ ;read bucket
    reply2 bucket
    setbucket (bsr $196);recurse
    stop ]
  if (:bus2 = $197) [ ;write card
    waituntil [new2?] ;yes, be paranoid!
    settemp bus2
    if (temp > 30) [stop] ;must be valid
    waituntil [new2?]
    if (not (temp = bus2)) [stop] ;both must match
    aset2 card-param 1 temp
    stop ]
  if (:bus2 = $198) [ ;init bucket with card & type
    setbucket (read-card and $1f) + ((aget brick-type 1) * 32)
    bsend $198 ;recurse
    stop ]
  if (:bus2 = $199) [ ;flash/beep
    beep beep beep;more than a wink
    stop ]
  if ((:bus2 > $199) and (:bus2 < $1a9)) [ ;not us
    bsend :bus2 - 1 ;ask upstream
    stop ]
end
```

```
to read-card
  ifelse (aget have-card? 1) [
    output get-card (aget2 card-param 1)] [
    output 0 ] ;dummy out for old firmware
end

to get-card :val
  ifelse (:val = -1) [;card in the slot?
    output 0 ] [; NIL can't be 255 because that's end marker
    output :val ]
end

to startup
  wait (aget brick-type 1) - 1 ;random / 3000
  ab, brake wait 1 off
end

to setbrightness :value ;0-7
    bsend $171
    bsend :value
end

to display :n
   bsend $170
   bsend :n / 256
   bsend :n % 256
end
```

B.2 Microwave oven controller (Logo)

```
array [list 20]
global [count foo rep-loc rep-count mm ss]

to uwave
  waituntil [not bsr $195] ;there
  wait 10
  bsend $198 ;init buckets
  wait 1
  setcount 1
  fill-list
  wait 1
  call-reversed
  off ;make darn sure
  waituntil [bsr $195] ;gone
  uwave
end
```

```
to fill-list
  loop [
    setfoo bsr $196 ;get-something
    wait 1
    if (foo = 255) [
      stop]
    aset list count foo
    setcount count + 1 ]
end

to call-reversed
  setrep-count 1
  loop [
    if (count = 1) [
      ifelse ((rep-count > 1) and (rep-loc > 1)) [
        setrep-count rep-count - 1
        setcount rep-loc ] [
      ;else
        stop ] ]
    setcount count - 1 ;count points past end
    setfoo (aget list count)
    ifelse (foo / 32 = 7) [ ; = repeat
      setrep-loc count  ;point past 1st rep'd instr
      setrep-count default (foo and $1f) 2 ] [;n
    ;else
      ifelse (foo / 32 = 5) [ ;end repeat
        if ((rep-count > 1) and (rep-loc > 1)) [
          setrep-count rep-count - 1
          setcount rep-loc ] ] [
        ;else continue
      ;else
        do-something foo ] ] ]
end

to default :n :default
  ifelse (:n = 0) [
    output :default ] [
    output :n ]
end

to do-something :thing
  light-up count
  if (:thing / 32 = 6) [ ;stir
     repeat 3 [ ;alert
       note 25 2
       wait 1]
     waituntil-door-open
     waituntil-door-closed
     stop ]
  if (:thing / 32 = 3) [ ;beep
    repeat 4 [
      light-up count
      note 20 5
      note 15 5 ]
```

```
    stop ]
  ;; the rest require a card
  if (:thing and $1f = 0) [ ;no card -> complain
    repeat  3[
      note 10 3
      light-up count
      wait 2 ]
    stop ] ;no card
  if (:thing  / 32 = 1) [ ;stand
    seconds-timer (:thing and $1f) * 10 ;sec
                          [light-up count]
    stop ]
  ; cook or defrost
  if ((:thing / 32 = 4) or (:thing / 32 = 2)) [ ;cook
    on
    wait 1 ;clear bus
    seconds-timer (:thing and $1f) * 10 ;sec
                          [light-up count
                           ;assure-door-closed
                             if (switcha) [
                                 off
                                 stop-counting
                                 waituntil-door-closed
                                 start-counting
                                 on ] ]
    off
    stop ]
end

to waituntil-door-open
     loop [
        wait 3
        light-up count
        if (switcha) [ ;door open
          stop ] ]
end

to waituntil-door-closed ;code dup
     loop [
        wait 5
        light-up count
        if (not switcha) [ ;door closed
          stop ] ]
end

to light-up :n
  bsend $199 + :n - 1
  wait 1
end

to setbrightness :value ;0-7
    bsend $171
    bsend :value
end
```

```
to display :n
   bsend $170
   bsend :n / 256
   bsend :n % 256
end

to seconds-timer :ss :tick    ;code dup!
  setbrightness 6
  setmm :ss / 60
  setss :ss % 60
  display 100 * mm + ss
  wait 1
  set-timer mm ss
  loop [
    if (minutes = 0) [stop]
    display 100 * (59 - minutes) + (59 - seconds)
    if (hundredths > 79) :tick
   wait 2]
end
```

B.3 Device Driver for Timekeeper bus device (Logo)

```
; $Bx - i2c write: 1st byte=addr, 2nd byte=data
; $Cx - prepare i2c read: 1st byte = addr
; $Dx - return unique id of this bus device
; $Ex - get i2c-read value

to i2c-write :addr :data
  bsend $1B7
  bsend :addr
  bsend :data
end

to i2c-read :addr
  bsend $1C7
  bsend :addr
  output bsr $1E7
end

to bcd-decode :bcd
  output (((:bcd and $F0) / 16) * 10)
         + (:bcd and $0F)
end

to bcd-join :high :low
  output (:high * 16) + :low
end

to bcd-encode :x
  output ((:x / 10) * 16) or (:x % 10)
end
```

```
to set-time :hh :mm
  stop-counting
  i2c-write 4 bcd-encode :hh
  i2c-write 3 bcd-encode :mm
  i2c-write 2 0
  i2c-write 1 0
  start-counting
end

to set-timer :mm :ss
  stop-counting
  i2c-write 4 0
  i2c-write 3 bcd-encode (59 - :mm)
  i2c-write 2 bcd-encode (59 - :ss)
  i2c-write 1 bcd-encode 99
  start-counting
end

to hundredths
  output bcd-decode i2c-read 1
end

to seconds
  output bcd-decode i2c-read 2
end

to minutes
  output bcd-decode i2c-read 3
end

to hours ;24-hour mode only
  output bcd-decode i2c-read 4
end

to day
  output bcd-decode ($3F and i2c-read 5)
end

to month
  output bcd-decode ($1F and i2c-read 6)
end

to year
  output ((i2c-read 5) and $C0) / 64
end

to weekday
  output ((i2c-read 6) and $E0) / 32
end

to time
  output (hours * 100) + minutes
end
```

```
to stop-counting
  i2c-write 0 ($80 or i2c-read 0)
end

to start-counting
  i2c-write 0 ($7F and i2c-read 0)
end

to set-date :m :d :y :weekday
  stop-counting
  i2c-write 5 (bcd-encode :d)
              or ((:y % 4) * 64)
  i2c-write 6 (bcd-encode :m)
              or (:weekday * 32)
  start-counting
end

;;;;;; Examples

;; Clock hh:mm
to clock
  loop [display 100 * hours + minutes wait 10]
end

;; Stopwatch ss:cc
to stopwatch
  loop [display 100 * seconds + hundredths]
end

;; Backwards (don't bring on airplanes)
to egg-timer :mm :ss :tick
  display 100 * :mm + :ss
  wait 1
  set-timer :mm :ss
  loop [
    if (minutes = 0) [stop]
    display 100 * (59 - minutes) + (59 - seconds)
    if (hundredths > 82) :tick
   wait 1]
end
```

B.4 Cricket Logo interpreter/OS firmware (Assembly)

The following pages document the changes I made to the Cricket Logo firmware ("Blue Dot" version 1.2, written by Brian Silverman, and maintained by Robbie Berg, Fred Martin, Bakhtiar Mikhak and the author) to create the Tangible Programming Brick. My major contributions to the code include the "slave" bus, the capacitive touch sensor, and support for a second EEPROM (the one on the card).

```
;;;;;;;;;;;;;;;;;;;;;
;
; Blue Dot Classic Cricket
;
;;;;;;;;;;;;;;;;;;;;;
; 5-24-99 0.3 Added touch sensor range
; 5-24-99 0.2 Changed adcon1 to all digital, added touch sensor (sensora), stubbed
                                          sensorb
; 5-23-99 Removed Run Light, added visual beep, greeting
; 5-23-99 changed default EEPROM address to 1
; 5-23-99 changed I/O pins to match 6x2 brick
; 5-18-99 added external eeprom select (aset2 aget2)
; 5-2-99 added bus-slave port (new2? bus2 reply2)
; 10-12-98 (v1.2) changed prim-output to not be stupid.
•
•
•
; demons
        [const dbits $3d]
          [const running 7]
          [const last 0][const active 1]
          [const bus2f 2][const bus2bit9 3]
          [const ee-select 4][const ee-select-mask $10][const ee-default 1]
          [const touch-range0 5][const touch-range1 6];[const unused 7]
        [const dcondl $3e][const dcondh $3f]
•
•
•
; i/o pin assignments
        [const touch-port porta][const touch1 0][const touch2 1]

        [const ir-in-port portb][const ir-in 0]                 ;was b0
        [const ir-out-port portb][const ir-out 1]               ;was b1
        [const beeper-port touch-port][const beeper touch1]     ;was b3
        [const button-port porta][const button 2]               ;was a4
        [const ee-port porta][const sck 3][const sdat 4]      ;was a2,a3
        [const motor-port portb]
          [const motora-l 5][const motora-r 4]                  ;was b5,b6
          [const motorb-l 3][const motorb-r 7]                  ;was b7,b7
        [const bus-port portb][const bus-port-ddr [sum :bus-port $80]]
          [const bus 2][const bus2 6]                           ;was b2,b4

        ;Mode = button (was for testing, may well stay this way)
                ;tie high for normal operation

        ;Capacitive touch sensor
            ;touch1 = beeper (for testing, but shouldn't interfere)

        [const pad-direct-port touch-port][const pad-direct touch1]
        [const pad-resistor-port touch-port] [const pad-resistor touch2]
•
•
•
;;;;;;;;;;;;;;;;;;;;;
; Main loop
;;;;;;;;;;;;;;;;;;;;;

start   [bsr io-init]
        [bsr read-autostart]
        [btsc autostart bits][bsr run-startup1]
        [bsr greeting][bclr autostart bits]
        [ldan $01] [bsr bus-tyo]        ; broadcast reset message on bus
•
•
•
        [bra prim-eb][bra prim-db]
        [bra prim-new2?][bra prim-bus2][bra prim-reply2]
```

```
        [bra prim-aset2][bra prim-aget2]
        [bra prim-set-touch-range][bra prim-touch-range]
•
•
•
prim-beep
        [bsr beep]
        [bra switch]

beep    [ldan aon-mask][xorm motors]    ;visible
        [ldan 35][sta t0]       ; so beep to show it
bp20    [bset beeper beeper-port]
        [bsr delay-loop]
        [bclr beeper beeper-port]
        [bsr delay-loop]
        [decsz t0]
        [bra bp20]
        [ldan aon-mask][xorm motors]
        [rts]

prim-note
        [bsr pop-byte][sta t0]
        [bsr pop-byte][sta t1]
        [bsr note]
        [bra switch]

greeting                        ;visible assumes motors off
        [ldan 20][sta t1] [ldan 1][sta t0]
        [bset motora-r motor-port]      ;yellow
        [bsr note]
        [bclr motora-r motor-port]
        [ldan 250][bsr delay-loop]
        [ldan 15][sta t1] [ldan 1][sta t0]
        [bset motora-l motor-port]      ;orange+green
        [bset motorb-l motor-port]
        [bsr note]
        [bclr motora-l motor-port]
        [bclr motorb-l motor-port]
        [rts]

;alarm  [ldan 21][sta t1] [ldan 2] [sta t0]
note    [lda ticksl][addn 100][sta r0l]         ;was label nt20
nt30    [bset beeper beeper-port]
        [bsr note-delay]
        [bclr beeper beeper-port]
        [bsr note-delay]
        [lda ticksl][sub r0l][andn $f0]
        [btss z status][bra nt30]
        [decsz t0][bra note]
        [rts]
•
•
•

prim-aset2
        [bsr pop-r0]
        [bsr pop-array-addr]
        [ldan num1h][sta @@]
        [ldan ee-select-mask] [xorm dbits]
        [lda r0h] [bsr ee-write-and-delay]
        [lda r0l] [bsr ee-write-and-delay]
        [ldan ee-select-mask] [xorm dbits]
        [bra switch]

prim-aget2
        [bsr pop-array-addr]
        [ldan num1h][sta @@]
```

```
        [ldan ee-select-mask] [xorm dbits]
        [bsr ee-read][sta r0h]
        [bsr ee-read][sta r0l]
        [ldan ee-select-mask] [xorm dbits]
        [bra return-r0]
•
•
•
prim-set-touch-range
        [bsr pop-byte]
        [sta t0]
        [bclr touch-range1 dbits]
        [bclr touch-range0 dbits]
        [btsc 1 t0] [bset touch-range1 dbits]
        [btsc 0 t0] [bset touch-range0 dbits]
        [bra switch]

prim-touch-range
        [clr r0h] [clr r0l]
        [btsc touch-range1 dbits] [bset 1 r0l]
        [btsc touch-range0 dbits] [bset 0 r0l]
        [bra return-r0]

prim-switchb
prim-sensorb
        [bra return-false]      ; false = 0

prim-switcha
        [bsr get-sensor]
        [bra return-false-if-small]

prim-sensora
        [bsr get-sensor]
        [bra return-r0]

;Parameters (delay=2 and n-measurements=30 worked well with cutoff of 50 in bus device)
        [const delay 2]
        [const n-measurements 32]

;Temp variables
        [const integral t0]             ;alias to existing blue-dot temps
        [const counter t1]

get-sensor
gsinit  [bclr gie intcon]                       ;interrupts off
        [clra][sta integral]
        [ldan n-measurements]                   ;loop counter
        [sta counter]
        [btss touch-range1 dbits][bra gsinit2]  ; prescale counter
        [ror counter][ror counter]              ; X1=1: div by 4
gsinit2 [btsc touch-range0 dbits][ror counter]  ; X0=1: div by 2

measurement-loop
init-up [bclr pad-direct pad-direct-port]               ;discharge the pad
        [bsr bang-mode]                                 ;(when enabled)

        [bset pad-resistor pad-resistor-port]   ;pull resistor high (when enabled)

        [bset bank2 status]                     ;"measure-mode"
        [bset pad-direct pad-direct-port]       ;set direct to input
        [bclr pad-resistor pad-resistor-port]   ;set resistor to output
        [bclr bank2 status]

cnt-up  [ldan delay][bsr delay-loop]            ;delay 4us * delay
        [btsc pad-direct pad-direct-port][bra init-dn]  ;bra if above cmos threshhold
        [incsz integral]                        ;if wrapped to 0 -> overflow
        [bra cnt-up]
```

```
oflow1  [ldan 255][bra return-a]

init-dn [bset pad-direct pad-direct-port]       ;charge the pad
        [bsr bang-mode]                         ;(when enabled)

        [bclr pad-resistor pad-resistor-port]   ;pull resistor low (when enabled)

        [bset bank2 status]                     ;"measure-mode"
        [bset pad-direct pad-direct-port]       ;set direct to input
        [bclr pad-resistor pad-resistor-port]   ;set resistor to "output"
        [bclr bank2 status]                     ;(i.e. sink to ground)

cnt-dn  [ldan delay][bsr delay-loop]            ;delay 4us * delay
        [btss pad-direct pad-direct-port][bra end-cycle];bra if below cmos threshhold
        [incsz integral]                        ;if wrapped to 0 -> overflow
        [bra cnt-dn]

oflow2  [ldan 255][bra return-a]

end-cycle
        [decsz counter]
        [bra measurement-loop]

end-measurement
        [lda integral]
return-a
        [bclr pad-direct pad-direct-port]       ;discharge the pad
        [bsr bang-mode]                         ;(when enabled)
        [sta r0l]
        [clr r0h]
        [bset gie intcon]                       ;interrupts back on
        [rts]                                   ;touch1/beeper left as output

bang-mode
        [bset bank2 status]
        [bset pad-resistor pad-resistor-port]   ;tristate resistor
        [bclr pad-direct pad-direct-port]       ;set direct to output
        [bclr bank2 status]
        [rts]
•
•
•
;;;;;;;;;;;;
;;; "Slave" bus port

prim-new2?
        [btss bus2f dbits]
        [bra return-false]
        [bclr bus2f dbits]      ;clear flag
        [bra return-true]

prim-bus2
        [clr r0h]
        [btsc bus2bit9 dbits][bset 0 r0h]       ;set the 9th bit of r0
        [lda bus2-data][sta r0l]
        [bra return-r0]

prim-reply2
        [bsr pop-r0]
        [lda r0l]
        [bsr bus2-tyo]
        [ldan 60][bsr delay-loop]
        [bra eval]

; send a byte down the bus. the "stop" bit is always 0
; of a data byte
```

```
; input in a
bus2-tyo[sta t0]
        [bclr gie intcon]       ;interrupts off
        [ldan bus-port-ddr][sta @@]
        [bclr bus2 bus-port][bclr bus2 @]       ;configure bus2 port as output
        [ldan 24][bsr delay-loop] ; give receiver a chance to sync (about 100us)
        [bset bus2 bus-port]            ; start bit
        [ldan 8][sta t1]
        [bsr an2-rts][nop][nop]
b2tyo50 [ror t0]                ; bit -> carry
        [bclr bus2 bus-port]
        [btsc c status]
        [bset bus2 bus-port]
        [nop][nop][nop]
        [decsz t1]
        [bra b2tyo50]
        [nop][nop]
        [bset bus2 @]           ;re-configure bus2 port as input
        [bset gie intcon]       ;interrupts back on
        [rts]
 •
 •
 •
;;;;;;;;;;;;;;;;;;;;;;
; eeprom
;;;;;;;;;;;;;;;;;;;;;;

; i^2c protocol
; see the data sheet for details

fetch   [ldan iph][sta @@]
; read a byte from the eeprom
; @@ should have a pointer to the address
; result in a
ee-read [bsr ee-addr-match?]            ; can we skip the write cmnd?
        [btsc z status][bra eer30]
        [dec @@]                        ; back to the high word
        [bsr ee-start]                  ; send a start bit
        [ldan $a0]
        [btsc ee-select dbits][orn $02] ; maybe select i2c address 1
        [bsr ee-send]           ; send write command (to set addr)
        [lda @][bsr ee-send]            ; then addr high
        [inc @@][lda @][bsr ee-send]    ; then addr low
eer30   [bsr ee-start]
        [ldan $a1]
        [btsc ee-select dbits][orn $02] ; maybe select i2c address 1
        [bsr ee-send]                   ; send current addr read command
        [bset bank2 status]
        [bset sdat ee-port]             ; data pin to input
        [bclr bank2 status]
        [bsr eein1][bsr eein1][bsr eein1][bsr eein1]
        [bsr eein1][bsr eein1][bsr eein1][bsr eein1]
        [bset bank2 status]
        [bclr sdat ee-port]             ; data pin back to output
        [bclr bank2 status]
        [bclr sck ee-port]
        [bset sdat ee-port]             ; send a nack
        [bset sck ee-port]
        [bsr ee-stop]
        [bsr inc-ee-addr]
        [lda ee-data]
        [rts]
 •
 •
 •
; write a byte to the ee-prom
; @@ contains a pointer to the addr
```

```
; ee-data has the data
ee-write[bsr ee-start]
        [ldan $a0]
        [btsc ee-select dbits][orn $02] ; maybe select i2c address 1
        [bsr ee-send]                   ; send write command
        [lda @][bsr ee-send]            ; then addr high
        [inc @@][lda @][bsr ee-send]    ; then addr low
        [lda ee-data][bsr ee-send]      ; then the data
        [bsr ee-stop]
inc-ee-addr
        [linc @][sta @][sta ee-addrl]   ; inc the low word
        [decsz @@]                      ; point back to the high word
        [btsc z status][inc @]          ; propogate carry
        [lda @][sta ee-addrh]           ; keep a copy of the pointer
        [rts]
•
•
•
;;;;;;;;;;;;;;;;;;;;;;
; interupt handlers
;;;;;;;;;;;;;;;;;;;;;;

; dispatched to either a timer int or an ir int
int-routine
        [sta int-a]
        [lda status]
        [bclr bank2 status]     ; who knows what state it was in...
        [sta int-status]
        [btsc intf intcon][bra int-ir]
        [btsc rbif intcon][bra int-bus2]

•
•
•
; after we receive a high-to-low transition on the bus2 port
; we block, wait low-to-high transition, and receive 9 bits of bus data
; return result in bus2-data    [Note: no timeout provisions]
int-bus2
        [bclr rbif intcon]
        [btsc bus2 bus-port][bra iret]  ;ignore low-to-high transitions
        [bsr bus2-tyi]
        [bset bus2bit9 dbits]
        [btss c status][bclr bus2bit9 dbits]
        ;[lda bus2-shift][sta bus2-data]   ;no need for double-buffer so just alias
        [bset bus2f dbits]
        [bclr rbif intcon]              ;does this prevent endless interrupts?
        [bra iret]

; return a byte in bus2-shift
; also return the inverse of the stop bit in the carry
; commands have a 0 stop bit -> carry set
; data has a 1 stop bit -> carry clear
bus2-tyi;[btsc bus2 bus-port][bra bus2-tyi]     ; already taken care of by int handler
b2tyi20 [btss bus2 bus-port][bra b2tyi20]       ; wait for sync edge
        [ldan 8][sta bus2-count]
        [nop][nop][nop][nop]    ;was [bsr an2-rts] ;(4 cycles, right?)
b2tyi30 [nop][nop][nop]
        [ror bus2-shift]
        [bclr 7 bus2-shift]
        [btsc bus2 bus-port]
        [bset 7 bus2-shift]
        [decsz bus2-count]
        [bra b2tyi30]
        [nop][nop][nop][nop][nop]       ;was [bsr an2-rts][nop]
        [bset c status]
        [btsc bus2 bus-port][bclr c status]     ; no stop bit -> carry clear
an2-rts [rts]
```

```
•
•
•
; set z on a button transition
; the button and the run light share a pin
button-edge?
        [bclr z status]
        [btsc autostart bits][bra be50] ; no stopping on autostart
        [btsc but bits][bset z status]  ;old but -> z
;       [bset bank2 status]
;       [bset button button-port]       ; switch to input
;       [bclr bank2 status]
        [bclr but bits][btsc button button-port][bset but bits] ; button -> but
        [btss run-light bits][bra be170] ; is not running, keep it as an input
be50    [bclr button button-port]
;       [bset bank2 status]
;       [bclr button button-port]       ; back to output if run light on
;       [bclr bank2 status]
be170   [btsc but bits]
        [bclr z status]
        [rts]

; set up the pins and initialize some ram variables
io-init [ldan 7][sta adcon]             ; *** only for 622 ***
        [bset bank2 status]
        [bclr motora-l motor-port][bclr motora-r motor-port]
        [bclr motorb-l motor-port][bclr motorb-r motor-port]
        [bclr ir-out ir-out-port]
        [bclr sck ee-port][bclr sdat ee-port]
        [bclr beeper beeper-port]
        [ldan $83][sta option]  ; set timer to / 16 = 250 hz
;       [ldan 2] [sta adcon1]    ; set ra0,ra1 to analog, ra2,ra3 to digital
        [ldan $3] [sta adcon1]    ; set ra0,ra1,ra2,ra3 to digital
        [bclr bank2 status]
        [bclr ir-out ir-out-port]       ; leave ir led off
        [clr bits][clr wait-counterh]
        [clr motors][clr dbits]
        [ldan ee-default][orn 0]       ;If ee-default=1
        [btsc z status][bra init20]   ;Set default ee bank to 1 (else 0)
        [ldan ee-select-mask] [xorm dbits]    ;(this should really be an #if)
init20  [clr ticksl][clr ticksh]
        [bclr motora-l motor-port][bclr motora-r motor-port]
        [bclr motorb-l motor-port][bclr motorb-r motor-port]
        [ldan $ff] [sta aspeed] [sta bspeed]
        [bset awho motors] [bset bwho motors]
        [bset inte intcon]
        [bset rbie intcon]      ;New: enable interrupt on portb change
        [bset t0ie intcon]
        [bset gie intcon]
        [rts]
```