Programming Computationally Enhanced Craft Items

by

Thomas E. Wrensch

B.S., University of Wisconsin, Parkside, 1986

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2002

This thesis entitled:
Programming Computationally Enhanced Craft Items
written by Thomas E. Wrensch
has been approved for the Department of Computer Science

Michael Eisenberg

Elizabeth Bradley

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Wrensch, Thomas E. (Ph.D., Computer Science)

Programming Computationally Enhanced Craft Items

Thesis directed by Assistant Professor Michael Eisenberg

This thesis is a study from a builder's point of view of materials that blend physical objects with computation. Specifically, this study is placed in the context of craft, where the builders are crafters and the materials are computationally enhanced craft items (CECIs). CECIs are traditional craft materials enhanced with behaviors and sufficient computation to control behaviors and to provide end-user programmability.

The question guiding this work is: *What are the minimal set of technical solutions that need to be developed in order to make general CECIs programming feasible?* An examination of this question falls into two areas.

The first area—creating CECIs—is discussed by considering them as variants of traditional craft materials and examining how they differ from those materials. The prototype CECI created as part of this study are used as concrete examples to identify and explain these differences.

The second area—programming by crafters—is illustrated using a scenario. The issues are expanded and generalized in a discussion of the unique features of the environment in which CECI programming occurs. These features are then used to support a design for a programming language and environment.

The results of this study are presented as detailed specifications for the technical solutions needed to design and develop CECIs and their programming system. Other results of this study include a model for CECIs, an understanding of the role of behaviors as a bridge between computation and physical objects, and an example of a programming language that may apply to user control of other objects that combine physical and computational materials.

Dedication

This work is dedicated to those who have, during my life, acted as mentors. Those who have encouraged and sometimes forced me to become more than I was: Gene Korienek, Timothy Fossum, Mike Ricker, and most of all my father Kenneth Wrensch.

Acknowledgements

This work would not have happened without the assistance of many people. In particular I would like to acknowledge Mike Eisenberg for starting me down this slippery slope and Glenn Blauvelt for conversation, encouragement, and his countless helpful suggestions. Others who provided much needed (if not always appreciated) comments were Eric Scharff, Elizabeth Bradley, Mark Gross, Andee Rubin, Gerhard Fischer, Robbie Berg, Fred Martin, Carol Strohecker, and Kathi Mundigler among many others.

I would also like to thank my family for their patience and understanding during this process.

This work was supported in part by National Science Foundation grants CDA-9616444 and REC-961396.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Computation is slowly invading the physical world. Smart toasters, web-enabled soda machines, and a truly incredible range of smart toys are only the most recent entries in a long series of products that combine some amount of computation with a physical object. The ubiquitous computing community, which studies the increasing amount of computation in our environment, predicts that computation will "fade into the background" as it becomes so common that we no longer notice it. There is every indication that this prediction is coming true.

However, there are times when background technology brings itself forcefully to our attention. One such time is when it breaks, requiring us to repair or replace it. Another is when we change our relationship with that technology from that of a user to a builder. A builder views objects and materials as something to manipulate to build larger structures, and so they are interested in controlling those objects and materials to a level far beyond that needed by a causal user.

This thesis is a study of the builder's point of view of materials that blend physical objects with computation. Specifically, this study takes place in the context of craft, where the builders are crafters and the materials are computationally enhanced craft items (CECIs). CECIs are variations of traditional craft materials like ceramic tiles, hinges, and thumbtacks that are enhanced by the addition of *behaviors*—specific ways to interact with the world—and sufficient computation to control those behaviors.

This study of CECIs and how crafters may use them breaks down into two areas: first, an examination of the unique properties of these enhanced materials and second, an examination of how those properties can be used and shaped by crafters. This second area is particularly interesting. Crafters, as builders, need considerable control of their materials. With traditional craft materials this control is accomplished by choosing particular materials or by painting, weaving, cutting, or otherwise modifying the materials directly. With CECI the behaviors and computation are another aspect of the material that crafters will need to control. This level of control over computation is in fact nothing other than programming. Thus this thesis is one of understanding what a CECI is and how it can be programmed.

1.1 Motivating Problem

Crafters are an admirable group of people. They spend time—often their free time—designing and building objects that are useful, beautiful, and often both. However, their creations rarely include computation. Crafts have not, for the most part, shared in the revolution that computers have caused in areas like graphic arts and writing. This is not terribly surprising since computers, in their most obtrusive configuration of boxes with screens and keyboards, are not very good tools for working with physical objects and seem an even worse candidate for a building material.

Thus the problem is that crafters have not been given an effective way to include computation in their craft. That they would be willing and able to make use of computation is suggested by their willing adoption of the new materials that have been made available over the last half century. These materials, ranging from polymer "clay" to electronic "jewels" that light up in patterns, have been adopted by by crafters because of their extra capabilities or reduced cost. This willingness to master new materials suggests a solution to the problem of introducing computation into craft. If the materials crafters use included a computational component, crafters might adopt them as easily as they have other new materials.

While there has been considerable general study in combining computation with physical artifacts in the ubiquitous computing community [2, 62, 115] and study of more specific applications in areas like wearable computing [67, 103] and smart toys [41, 64, 69, 102], these communities have not taken into account the level of control need when using these artifacts to build larger structures.

1.2 Background

One of the first questions that comes up in this work is "What is craft?". Unfortunately "craft", like "art", is fiendishly difficult to define. Here, a set of examples and

a discussion of the important characteristics of craft will be used in place of a definitive definition. Table 1.1 lists some crafts and materials commonly used in those crafts.

Craft	Some Commonly Used Materials
Primitive Dolls	Fabric (Muslin), Fiberfill
Mosaics	Ceramic Tiles, Grout
Furniture	Wood, Glue
Flower Arrangement	Dried and Artificial Flowers
Pottery	Clay, Glaze
Stained Glass	Colored Glass, Lead
Jewelry	Wire, Silver Solder, Glass
Origami	Paper
Quilting	Cotton Fabric, Thread
Paper Making	Wood (Pulp), Cloth, Dyes

Table 1.1: A list of crafts and some materials commonly used in those crafts.

There are three characteristics that separate craft from other activities:

- Craft has as its purpose the construction of physical artifacts.
- The artifacts being built should have some utility.
- The work should be done “by hand”.

These characteristics are themselves somewhat vague. The word craft is sometimes applied to activities—such as software development and creative writing—that do not produce physical objects. The idea of utility must be stretched a bit to fit strictly decorative crafts such as flower arranging and crafts that may be considered play, such as the building of model railroad scenery. However, it is the idea of something being built “by hand” which causes the most difficulty.

By hand may be meant literally, as in the way a potter shapes clay. It may also mean with the use of hand tools in crafts like woodworking, where materials are too hard or tough to be worked directly. Then there are power tools like table saws and computer controlled tools such as routers and laser cutters. Are objects built with these tools still built “by hand”? This is a difficult question. One answer is put forth by

McCullough in his insightful book *Abstracting Craft* [78]. He suggests that "by hand" describes a situation where the crafter is, at all times, in control of the construction process. This view neatly separates craft from factory-style construction where the builder serves the process of construction rather than being in control of it.

CECIs must fit these characteristics to be used in the context of craft. This need lead to another set of characteristics of the use of CECIs in crafting. This second set of characteristics both shaped this investigation and differentiated it from other work in computationally enriched artifacts. These characteristics are:

- CECIs have embodied computation.
- Crafters are end-user programmers.
- Programs for these objects are small.
- The programming environment is separate from the CECI.

Descriptions of and justifications for these characteristics can be found throughout this thesis, and in particular in Section 3.3. What follows in the balance of this section is a synopsis.

The computation in a CECI is embodied in the sense that the physical craft object and its behaviors are the computation's sole interface to the outside world. In other words, the different ways that the computation can see and affect the world is limited to what the CECI can do. This is in contrast to a desktop computer systems whose suite of input and output channels are much richer and more extensible, and which are designed to transmit information rather than to directly interact with the physical world.

In general, only crafters have sufficient knowledge of their work to know how a craft material will be used: how it should be shaped, placed into some larger structure, or otherwise manipulated. This applies as much to enhanced materials as it

does to traditional materials. Only the crafter has sufficient knowledge to know how a CECI should behave. And so the end user—the crafter—should be the programmer. Moreover, CECIs and their programming system must be designed for the majority of crafters who have little or no programming experience, not for the few who are experienced programmers.

Programs for CECIs are small. Experience with the prototypes CECIs built during this investigation suggests that most programs will be under a dozen lines of code, with more than forty lines being rare. There are several reasons for this. The most prominent is that CECIs are very simple objects, with limited inputs and outputs. There is no need for smart hinges to execute complex algorithms or handle database transactions; they are limited to a domain where motion in one degree of freedom, single-value sensors, and other simple behaviors are the subject of their programs. These restricted domains do not call for complex programs.

The programming environments for CECIs, like other very small computer systems, are separate from the object itself. The reason for this is simple; there is no place to fit a keyboard and monitor on a craft material such as a ceramic tile. Of course there are other ways to communicate programs, but they all have the same set of problems: development tools need a set of computational resources and user interactions than cannot be supported by a CECI.

1.3 Research Question

The research question that provided focus for this work is:

> What are the minimal set of technical solutions that need to be developed in order to make it feasible for crafters to program CECIs?

We define a technical solution as a piece of technology created to solve a particular problem. The technical solutions created in the course of this research are related

to the goals of building CECIs or of enabling end users to programmatically control them. Technical solutions for these goals might include a programming language, a hardware architecture for CECI control electronics, and the development of software for editing CECI programs.

While many technical solutions will be required to design and build CECIs and enable crafters to program them, most of those solutions will come from simple applications of well-known techniques, or even the "borrowing" of existing solutions. An example of this is the editing of the text in a source program, a well known problem with many solutions available. Technical solutions of this type may be described here, but are not included as part of the minimal set. Only technical solutions requiring developments unique to CECIs or meaningful innovations are reported. The technical solutions fitting this criteria are listed in Chapter 4.

"Minimal" as used in the research question does not refer to the theoretical minimum possible set. For the theoretical minimum set, all that is necessary to program a CECI is a programmable device and some means of communicating a program to it. A smart tile that can sense when it is being touched, for example, could be programmed by tapping on it in a particular sequence. It is safe to say that nearly all users would dislike this approach to a programming user interface. Thus "minimal" is taken here to mean the minimum necessary for crafters to be both able and willing to create programs for CECIs.

1.4 Approach

This investigation made heavy use of scenarios. Three were originally developed, one for each type of prototype CECI. These were useful, but working with the actual prototypes uncovered additional ways of interacting with the CECI that were not covered in the initial scenarios. Additional scenarios were developed as needed to cover these situations. Most of these were partial scenarios, focusing on a single task,

but a few were more general and covered a wide range of tasks. One of the general scenarios can be found in section 3.2.

The general approach taken to investigate the research question in the previous section included an iterative cycle of design, construction, testing, and analysis. One or more scenarios were used to guide the design and testing. The results of this cycle were artifacts (both physical objects and software), design guidelines, and technical solutions. The results were tested for generality by generating ideas for new kinds of artifacts and attempting to map them onto the existing set of technical solutions.

Each cycle began with a design for a new artifact—either a type of CECI or programming system software. The design was guided by at least one scenario describing how the artifact would be used by crafters. The design was used to develop a prototype, which was tested for basic functionality and for usability against the scenario. The information from the tests was analyzed to produce new design guidelines and extract the technical solutions created during the process.[1] .

A difficulty with this technique is that it is difficult to describe the progression of ideas that lead to the finished artifact. In Chapter 2, some of this design and development process is described by presenting the sequence of CECI prototypes developed, and listing the major design changes that each prototype prompted in the next iteration.

In summary, the approach was one of iterative refinement of physical and software artifacts interleaved with the iterative development of design ideas and the technical solutions needed to make the construction and programming of CECIs feasible.

[1] This description does not properly convey the richness—and chaos—of this method. Phases of the cycle overlapped, multiple prototypes were being worked on simultaneously, and construction often had to be stopped and the object redesigned when some flaw was found.

1.5 Chapter Guide

Chapter 2 discusses the first major area of study in this investigation: defining computationally enhanced craft items (CECI) and how they are constructed. This chapter places CECIs into the context of traditional craft and craft materials and how that effects their basic design. Each of the three prototype CECI is described, along with the lessons learned from their design, construction, and use. Finally the architecture of the final prototypes computation is presented.

Chapter 3 covers the second major area of study: the programming of CECIs by crafters. A usage scenario and a list of the affordances of programming CECIs are used to develop an understanding of why and how crafters would write programs for CECIs. A general discussion of the characteristics of programming systems and a discussion of programming systems with needs similar to CECI programming provide the background for a description of the programming system developed for this study. The design of the programming language and development environment are presented and explained.

Chapter 4 details the important technical solutions developed in the course of this investigation and introduced in Chapters 2 and 3. In Chapter 4 they are described in detail, and where necessary their development and use is justified.

Chapter 5 is a review of related work. This chapter presents previous work in end-user programming, including work on programming by example, visual programming, and programming using natural language. Various techniques that have been developed for combining physical objects with computation are summarized, with particular emphisis on the use of small control computers and microcontrollers. The chapter concludes with a discussion of existing research and commercial efforts in applying computation to the world of craft.

Chapter 6 summarizes the key findings of this research, including the results of

user testing. It then discusses the future directions of this research with emphasis on specific goals and the next steps toward those goals. The chapter concludes by enumerating the core contributions this work has made to the field of computer science.

Chapter 2

Computationally Enhanced Craft Items

Computationally enhanced craft items (CECI) were defined in Chapter 1 as craft materials enhanced by behaviors and a sufficient amount of computation to control those behaviors. This chapter explores and refines this definition, acting as a necessary foundation for the next two chapters, which cover programming and the technical solutions needed to make both CECIs and their programming possible.

CECIs are a blend of traditional craft materials, behaviors, and computation. They are intended to be a new kind of building material for crafters. To take advantage of crafters' knowledge and experience, each type of CECI described in this thesis is based on an existing craft material: hinge, ceramic tile, and thumbtack. Each of these was created using a different approach to combining behaviors with the traditional materials: automating an existing behavior, automating the change of a visual attribute, or adding a completely new behavior. Each prototype was designed and built using lessons learned from the previous attempts in an iterative refinement process. The result of this process is an understanding of the structure of CECIs in general, and the roles each of the three elements—traditional craft material, behavior, and computation—have in that structure.

This chapter talks about CECIs from three different points of view. The first is in terms of craft and existing craft materials. The second is as the sequence of prototypes built as part of this work and the lessons learned in the design and construction of

each. The third is a general architecture for CECIs.

2.1 CECIs and Craft

In the last century, there has been a dramatic increase in the variety of materials available to crafters. Some of this is due to the introduction of high-tech materials such as microprocessors and fiber optics. However, many of the new materials for crafters are middle-tech [32]. These are materials such as glossy paper, temperature-sensitive film, and kevlar fibers, which may not be considered high-tech themselves, but which are the result of high-tech processes and techniques. These materials have had a strong impact on many crafts in that they provide either a truly new kind of material or inexpensive replacements for expensive or difficult-to-find low-tech materials.

CECIs are intended to be another addition to the library of materials available to crafters, a blending of familiar craft materials and computation. They differ from existing craft materials in their ability to interact with their surroundings in a controlled way. The goal in the development of CECIs is to give crafters materials that are analogous to those that they currently use, but that also give them an extra dimension in the construction of their craft—the ability to easily build active—or even reactive—artifacts.

2.1.1 Craft Materials

A craft materials is any material used in the construction of the physical artifacts crafters build. These may be common building materials such as wood, paint, or nails. They may be materials specifically created for crafts such as clockworks, ceramic tiles, or doll hair. They may even be found or scavenged materials like rocks, soda bottles, or old toys.

Given this definition, the idea of traditional craft material seems meaningless. If *anything* can be a craft material, then there is no reason to single out any particular

material and call it “traditional”. While this is true to some extent, traditional craft materials do have in common that they rarely include any kind of autonomous behavior and even more rarely have any sort of computational control.

Recently, and in particular in the last few decades, crafters have seen a huge increase in the selection and variety of available materials. Technology has produced inexpensive and easy-to-use replacements for expensive or difficult to use materials like silk, oil-based paints, and even wooden planks[1] . In addition, completely new high- and middle-tech materials have been introduced such as shape memory alloy, temperature sensitive film, and compact low-power electronics—none of which were available thirty years ago.

These new materials allow crafters to do things that they could not do with traditional materials. This may be due to cost or availability—rayon is less than a third the cost of silk—or it may be due to attributes of new materials, such as the strength of epoxy resin or the phosphorescence of special paints. CECIs can fall into either of these categories, as a replacement for an expensive actuator and control system or as a variation of some traditional material with new properties of behavior and computation.

2.1.2 Existing Behavior and Computation in Craft

CECIs differ from traditional craft materials in that they interact with the world, have autonomous behaviors, and in that those behaviors are controlled programmatically. A careful search of a large craft store will turn up a handful of craft items that have behavior: that move, change, or interact with their environment without being driven by an external force.

Programmable crafts are more difficult to find, but not completely unknown.

[1] Plywood was invented as a way to produce large sheets of wood that did not warp; a single sheet of plywood replaced many planks which had to be tightly fitted to do the same job.

Two areas of crafting where programmable systems have had some impact are model railroading and the animated displays often found in store windows during the holiday season. Model railroaders use control systems to run the engines, switch tracks, and control scenery elements of complex layouts. Many of these control systems are either directly programmable or allow an attached desktop computer and associated software to provide some level of programmability. [36, 42, 104]

The automata found in store displays and homes of the more energetic holiday decorators may also contain programmable controllers of some kind—though many are animated using strictly mechanical devices. Over the last few years a new element has been added to this particular craft: prebuilt automated holiday decorations. These take the form of—for example—a reindeer doll which sings holiday songs when it detects movement. While these objects currently lack programmability of even the most primitive kind, they do contain internal computation that controls autonomous behavior. While not a promising start, they do show that the technology exists to produce computationally enhanced objects at a reasonable cost. [57, 59]

Computation has been slowly creeping into other areas of crafting as well, but in the form of tools rather than materials. One example of this is HyperGami and its successor JavaGami, which let users design three-dimensional paper shapes [33]. The software unfolds the shapes into a form that can be printed and folded into a three-dimensional shape. Other examples of this type of tool can be found in Section 5.3.

Craft and computation can also be combined without being blended into into the same object. There are small, easy-to-use control computers available that could be combined with sensors and actuators to produce craft projects that include all of the capabilities provided by CECIs. These small and easy-to-use computers, like the Lego Mindstorms RCX [69], the Cricket [74], and the BasicStamp [93], seem like a reasonable alternative to CECIs for introducing computation into craft projects.

The idea of using small control computers to add computation to craft is appeal-

ing. Their use might suggest a way to package computation as a sort of craft material in and of itself. However, small and inexpensive control computers have been available for many years without making a meaningful impact on the world of crafting. Examinations of craft stores and general craft magazines have turned up no mention of, or advertising for, these devices (with the exception of model railroad magazines). A hint as to why this may be the case turned up during a university course taught by Michael Eisenberg and Mark Gross called Things that Think. In this course, nine teams of students—a mix of computer science, architecture, and other disciplines—created mechanical automata, some of them quite beautiful. At the end of the project the students were asked to use Cricket control computers and a set of standard sensors and actuators to add computation to their projects. With one exception, the results were rather pathetic. Upon reflection this is not terribly surprising. Computation is *different* from wood, glue, and string in a very basic way. Integration of computation with traditional materials takes planning and effort.

2.1.3 CECI

Consider a crafter building a tile mosaic of a blooming flower. Using CECI variations of ceramic tiles that are able to change color, the traditional tiles making up the flower could be replaced in such a way that the flower appears to bloom or fade (see figure 2.1). The crafter may wish this process to take place over time—perhaps to have the flower bloom over the course of the morning only to wither in the late afternoon. Or the blooming and fading could be triggered by touching the tiles, by the level of sunlight, or some other change in the environment.

Note that this description of the tile mosaic that uses enhanced tiles does not discuss computation directly. Instead, it talks about how the mosaic and individual tiles behave over time. To the crafter, a CECI's behavior—the way it can interact with the world—is more important than the computation. The computation is just a means

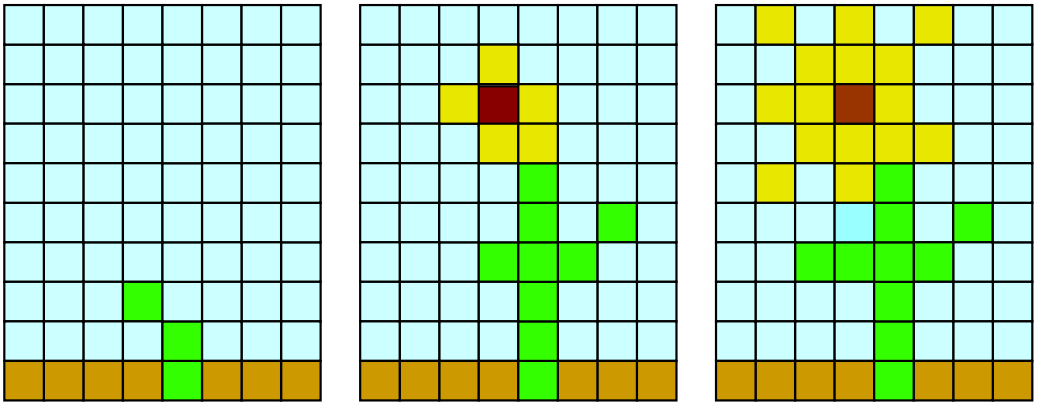


Figure 2.1: A sequence of diagrams of a tile mosaic using smart tiles. The mosaic shows a flower, which can appear to bloom or fade by changing colors of enhanced ceramic tiles. The sequence above shows the flower just after sprouting, just beginning to bloom, and in full bloom.

for the crafter to control the behaviors. It also provides a way to integrate the actions of a CECI that contains multiple behaviors, like a ceramic tile that can change color and sense touch, into a craft project. See Section 2.3.1 for a more detailed description of behaviors.

The flower mosaic example also shows that traditional craft materials and CECIs can be mixed in a single project. In this case the CECI tiles are used to create the flower, but the background tiles can be normal ceramic tiles. This ability is desirable to provide maximum flexibility (and minimum costs) in material choice. For this to work the enhanced and non-enhanced tiles must be interchangeable, at least so far as being the same height and width. Physical similarity between CECIs and their non-enhanced counterparts also makes it easier for experienced crafters, who are familiar with the traditional materials of their craft, to use CECIs.

Finally, this example illustrates an important attribute of the artifacts that crafters build—they tend to be one-of-a-kind objects. While the particular color and size of the CECI tiles might be chosen from a craft store shelf or ordered from a catalog, the detailed behavior of the tiles is too complex and idiosyncratic to be chosen from a list.

The behavior of the mosaic, and thus of its constitute tiles is as unique as the mosaic itself. Only the crafter knows how the CECIs should behave, and so only the crafter can decide how they should be programmed. Thus without some element of crafter programmability, the extra capabilities of CECIs are of limited use.

It is fairly easy to imagine a programmable ceramic tile, silk flower, or even a programmable nail, but what of materials like string and glue? Craft materials can be divided into two categories based on what Russell and Norvig call *individuation* [105]. In this scheme "objects" are things that have an individual identity while "stuff" is continuous, with no well-defined border between individual pieces. Thus a brick is an object while paint is stuff. This distinction is important because only craft materials that are objects can be used as the basis for a CECI. The reason is simple: stuff does not have any convenient place in which to put the computation.

The difference between stuff and objects is largely a matter of scale. Sand is stuff, but a single grain of sand is an object. While it is beyond current technology to create a CECI on the scale of a grain of sand, advances in MEMS technology suggest that it is not far off [11, 27, 37]. Perhaps more importantly, the technology to program a large (tens of thousands or more) collection of smart objects does not exist, though that too may not be far off [1]. For the purpose of this thesis, CECIs will be constrained to craft items that are large enough to contain the computational components, with the hope that technological advances will extend these capabilities to a larger set of craft materials.

2.2 Prototypes

While it is possible to study something without actually having the object available for direct observation—something that Cosmologists and Quantum Physicists do all the time—it is foolish to do so when it is unnecessary. We found the development and use of prototypes an invaluable part of the process of understanding CECIs and

their programming. In this section, each of the three major prototypes is described in detail.

Each of the prototypes pushed one particularly difficult problem to the fore. The programmable hinge emphasized the need for purpose-built computation, both to match the physical limitations of the CECI and to provide domain-specific programming primitives. The smart tile highlighted the importance of communication—both between two CECIs and between CECIs and the programming platform. The Rototack provided a deeper understanding of what was needed for CECIs to fit into the larger world of crafts, from design and manufacturing issues to the effects of subtle variations in behavior.

One thing this section can*not* convey is the sheer joy of bringing these working physical objects into existence. While the excitement of getting a complex piece of software to work after much effort is exhilarating, it is an essentially solitary event. A working physical artifact is different. It can be carried from room to room and shown to people who do not understand the esoteric world of research software development. Other people understand the excitement and joy. It is a shared experience rather than a solitary one, and that makes a tremendous difference.[2]

2.2.1 Programmable Hinge

The programmable hinge [120], the first CECI prototype, was built on a cabinet hinge. A hinge is conceptually easy to turn into a CECI as it already has a standard, single behavior—opening and closing—that can be automated. Hinges are also quite common; they can be found in cabinets, doors, and box lids.[3] A CECI hinge could also be used in other situations where a source of pivoting open/close motion is useful, such as automating the movement of doll arms or lifting the flaps of model airplanes.

[2] Not that getting these objects working was an experience that happened with great frequency.
[3] My house contains 27 door hinges, 32 cabinet hinges, and at least 14 hinges used for box lids.

The first variation of the hinge prototype was built from a medium-sized cabinet hinge using Shape Memory Alloy (SMA) actuators. SMA is a metal alloy, usually in the form of wire, that remembers a set shape and returns to it when heated (see [52] or [106] for an overview of SMA). The hinge CECI's actuator uses two SMA wires, one wire set to the hinge's open position and another to its closed position. The hinge can then be opened or closed by heating one or the other of the wires. SMA wire has sufficient resistance to allow it to be heated by running a current through it, a convenient but not particularly power-efficient technique that was used in this prototype.

There were four major variations on the hinge, three that used different designs for the SMA actuator and one that used a DC motor with its speed reduced by gearing. Each new design iteration increased the force exerted by the hinge and the level of control, though the level of control was never great. The SMA actuators were very sensitive to environmental conditions such as ambient temperature and battery condition[4]
.

Computation was supplied by a Cricket [74] a small, general-purpose control computer. Because the Cricket could not directly supply enough current to heat the wire, a second battery and a pair of switches (reed relays, later replaced with solid state devices) supplied power to heat the wire. The prototype consisted of four pieces (hinge, Cricket, switches, and extra battery), connected by wires. This awkward arrangement can be seen in Figure 2.2.

The Cricket had a number of problems as a source of computation for the programmable hinge: among other things it is very large relative to the hinge, and it requires additional circuitry to drive the actuators. However, it is easy to program and re-program, a capability that was very useful in experimenting with hardware control and other programming issues. One of those issues turned out to be a mismatch be-

[4] They also seemed to be sensitive to the phase of the moon, day of the week, and importance of the person watching the demo.

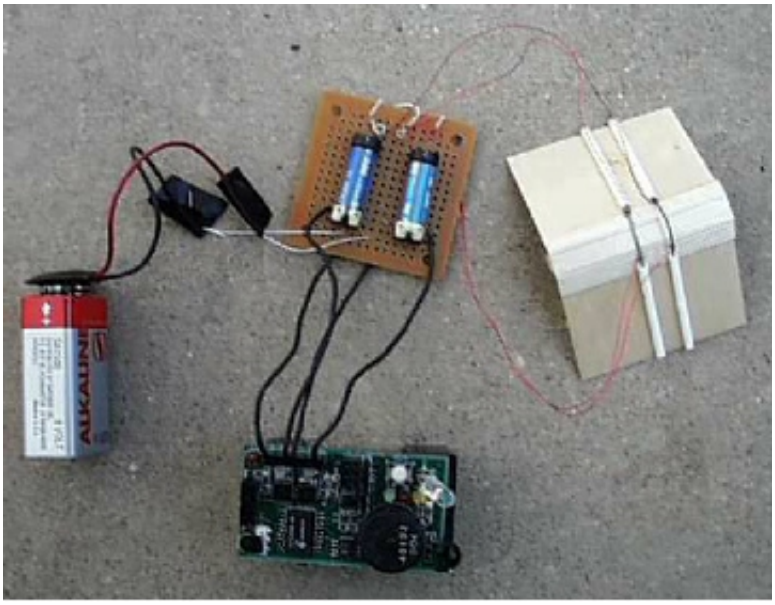


Figure 2.2: An early programmable hinge prototype. The SMA wires can easily be seen on the back of the hinge (right side). The circuit board at the center top includes power switching for the SMA using the 9 volt battery at the left. The Cricket control computer is at the bottom.

tween the Cricket's general-purpose control primitives and the operations of a hinge. Here is a Cricket Logo program to open and close a hinge:

```
to run
    a, on
    wait 25
    a, off
    wait 50
    b, on
    wait 22
    b, off
    end
```

It is impossible to tell what this program will do unless one knows exactly which actuators are being used and how they are connected to the Cricket[5] . A better version can be constructed by defining Logo words for open and close operations:

[5] The MetaCricket [75], an improved version of the Cricket with an intelligent device bus, would use primitives appropriate for the actuator or sensor. While this is a dramatic improvement, the primitives would still not directly relate to the craft item.

```
to run
    openByDegrees 120
    wait 50
    closeByDegrees 120
    end

;;
;; Definitions for hinge 'primitives'
;;
to openByDegrees :degrees
    b, off
    a, on
    wait 5 * :degrees / 36
    a, off
    end
to closeByDegrees :degrees
    a, off
    b, on
    wait 2 * :degrees / 15
    b, off
    end
```

While the word definitions that encapsulate the open and close operations in the above program are still difficult to understand, the intended operation of the overall program is clear. This program hides the low-level details by creating operations that makes sense for a hinge. A better language for manipulating CECI hinges would eliminate or hide the definitions of the open and close operations completely. This would hide the ugly hardware-specific details and provide crafters with domain-specific primitives to use in expressing the actions of the hinge.

Experiments with CECI hinge programs also showed the important role that time plays in programs that interact with the real world. Specifically, all programs needed to wait, usually by a fixed amount of time but sometimes for a particular condition to occur. While the Cricket Logo's `wait` primitive was very useful, it was limited to times expressed in tenths of a second. Some applications required wait times of minutes or hours, prompting the creation of new waiting operations similar to the open and close "primitives".

While the programmable hinge was an awkward and difficult to use implementation of a CECI, it was a excellent first prototype, as it led to a careful reexamination of programming and control of actuators. It also made it obvious that the three basic elements of a CECI—the traditional craft material, the behavior, and the computation—needed to be more tightly integrated to produce useful CECIs. The next prototype—the smart tile—was an attempt to address these programming and integration issues.

2.2.2 Smart Tile

The smart tile was based on a ceramic tile of the kind found in bathrooms and kitchens. One popular craft using ceramic tiles is the creation of tile mosaics, pictures and designs made from arrangements of different colored tiles (see Figure 2.3). Smart tiles are different from the other two CECI prototypes in that their behaviors do not involve motion and they are almost always used in large groups. The primary behavior of the prototype smart tiles is the ability to change appearance. The first prototype did this by illuminating a tinted window set into the tile. A later variation, which was designed but never fully constructed, would have changed color using a liquid crystal. Both the original prototype and the later design had the secondary behavior of being sensitive to touch.

The smart tile prototype is formed from a three inch by three inch tile made from a polymer clay with a tinted plastic window set in the center. The tile is mounted on a frame containing the electronics, a light, and a pressure switch (see figure 2.4). When the light is on, the window in the tile glows. Pressure applied to the tile's surface activates the switch. The frame is two inches high and strong enough that the assembly can be used as part of the tiling for a floor.

The electronics for the smart tile included on-board computation provided by a small microcontroller (a Microchip PIC16C84 [58]) buffering for the LED and switch, and primitive communication support. The communication system provided eight

Figure 2.3: Picture of a tile mosaic. This mosaic of a lion's face uses patterned and irregularly shaped tiles rather than the more common single-colored square tiles.

Figure 2.4: The smart tile prototype. The base is three inches square and stands two inches tall. The tile is made from a polymer clay with a tinted plastic window set in the center. The electronics, communication, and power connections are mounted in the base under the tile.

"ports" which could carry a simple on or off signal to anther smart tile.

While ceramic tiles have many uses, smart tiles were intended for mosaics. Tile

mosaics contain from dozens to thousands or even hundreds of thousands of tiles, and though not all the tiles in a “smart” mosaic would need to be CECIs, even a simple mosaic would likely use at least a dozen. This suggests that CECIs, or at least those based on ceramic tiles, will need to communicate with each other. The scenarios developed here for using smart tiles required communication with non-adjacent tiles and show that the communication system would need to carry more information than the simple signal originally supported by the smart tile’s electronics. In addition, the need to reprogramming smart tiles that had already been built into a mosaic means that the communication system needed to be able to convey new programs as well as the information necessary for tiles to cooperate. More details on the communication system that evolved from these considerations can be found in Section 4.3.

Reprogramming smart tiles in a mosaics is made more difficult because mosaics are not terribly portable. They are usually built into the floor, wall, or ceiling of a building or used as the surface of a table or counter. In all of these cases the mosaic is both heavy and fixed, and there is no reason to suppose that incorporating CECIs would change that. This is a problem when the smart tiles in a mosaic need to be reprogrammed due to programmer error, adjustment for changes in the environment, or artistic reasons. Techniques to adjust or reprogram tiles “in the field” may be necessary for smart tiles and other CECIs which cannot be brought back to the workshop.

The smart tile was a major step forward in creating a usable CECI. It integrated the physical craft material, computation, and behaviors in a package very similar to a traditional ceramic tile. Indeed, the smart tile’s biggest drawback was that it was *too* interesting and versatile, at least to a computer scientist. Smart tiles that could be connected to larger computer systems provide an easy way to turn almost any surface into a human-computer interface. While this is an exciting idea, it detracts from the idea of blending computation and craft. The smart tile showed that communications systems for CECIs needed to be more flexible, both the allow for reprogramming tasks

and to allow a richer interaction between CECIs used in a craft project.

2.2.3 Rototack

The Rototack [119] is based on a thumbtack or pushpin of the kind typically used to post notices on corkboards. Pushpins do not have interesting existing behaviors to automate, and the idea of changing the appearance of an object had previously been explored via the smart tile, so the Rototack was created by adding a new behavior to the object, namely rotation. Three versions of the Rototack were created in an attempt to produce a prototype of near-production quality that could be tested with crafters in real craft projects, an attempt that met with mixed success.

The Rototack is shaped like a thumbtack with an oversized head (see figures 2.5 and 2.6) with a small gear protruding from the top. It is the gear that turns, rather than the entire head of the tack. The case for the first Rototack was made from two plastic bottle tops, and was later changed to a more aesthetically pleasing wood design. The rotation was produced by a tiny stepper motor, one centimeter in both diameter and length. The final version of the Rototack also included a bi-color LED on the side, used primarily for testing programs, and a switch sensor to indicate when the tack was pushed into a corkboard.

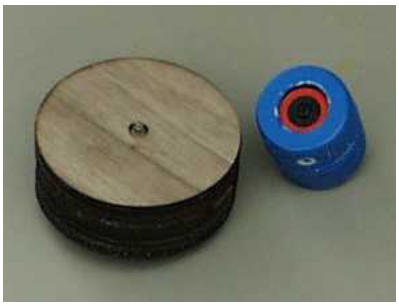


Figure 2.5: Two versions of the Rototack. The figure on the left is the final version (Mark III) and the version on the right is the original version of the Rototack (Mark I). The Mark III's size allows for larger batteries.

There were two main differences in the three versions of the Rototack. The first

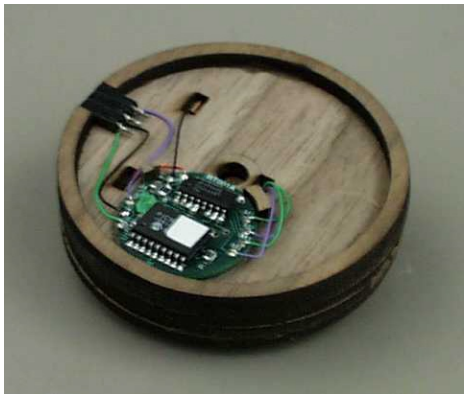


Figure 2.6: A Rototack with the bottom removed, exposing the electronics. The small circular circuit board contains the microprocessor and stepper motor driver.

difference was the motor: the initial prototype used a low-speed DC motor while the later versions used a small stepper motor. The DC motor provided smooth motion and reasonable torque, but could not provide accurate rotational control. The stepper motor was smaller and could be precisely controlled, but required a much more sophisticated motor driver.

The other major difference was power. The first two prototypes were small, about one inch in diameter and somewhat less than an inch tall (excluding the spike). This small size required the use of small batteries, which limited the tacks to about fifteen minutes of motion on a single set of batteries—an unacceptably short time, even for a prototype. The final version of the Rototack prototype was two and a half inches across, allowing room for batteries that lasted several hours and drove the motor at a higher torque. Even with the larger batteries the Rototack was restricted to turning lightweight foam or cardboard pieces, limiting the kinds of projects for which it can be used. More careful matching of components—particularly the motor—to the application should be sufficient to at least double the torque supplied without reducing the battery life.

The Rototack was an attempt to create a prototype that could be used in real craft projects; because of this the design issues were different from the programmable

hinge and the smart tile. The engineering of the previous prototypes was limited to making a working object. The Rototack engineering had to address issues of usable case design, dependability, and variations in the behavior that affected usability, such as torque and battery life. The final version of the Rototack was used in limited user testing (see section 6.1), so the effort was at least a partial success.

2.3 Architecture of Enhancement

The previous section discussed CECIs as individual examples, focusing on the details of each. This section focuses on the structure of CECIs in general, in particular the structure of the elements that make a CECI different from traditional craft materials: the behaviors and computation. The material in this section is an important foundation for the next chapter, which talks about how CECIs are programmed.

This section starts by discussing behaviors and the different kinds of descriptions of behaviors needed for developing and programming CECIs. This is followed by a discussion of how behaviors are controlled by a CECI's computation and the overall structure of the computation for CECIs, focusing on the software architecture rather than the hardware.

2.3.1 Behavior

Behaviors are ways of interacting with the world such as rotation, switching, light sensing, and color change. When a behavior, such as the rotation in a Rototack, is used in a CECI it must be implemented by a some device or set of devices. Examples of these devices are a two-phase stepper motor, a push button, a solar cell, or an LCD sheet. Table 2.1 lists some behaviors and the devices that might implement them.

Behaviors can also serve as a classification scheme for the devices that implement them. Various classification schemes for the devices used in human-computer interaction produce taxonomies that define properties and constraints over large groups of

Behaviors	Devices
linear motion	linear actuator, worm drive
rotation	DC motor, stepper motor, servo
color change	liquid crystal
light	incandescent bulb, LED
switch	push button, reed switch
light level	solar cell, CDS cell

Table 2.1: A list of some behaviors, which are high level descriptions of interaction with the physical world, and the lower level devices which might be used to implement those behaviors.

similar interaction devices [16, 17, 18]. A behavior plays a similar role for the devices used in CECIs; defining a set of attributes and operations that describe—in a general fashion—the constraints on devices that implement that behavior.

This idea of constraints works both ways. When a CECI is designed, the devices that implement its behaviors place constraints on how that behavior is expressed. Table 2.2 is an example of the limitations placed on the rotation behavior for the Rototack. This information on the constraint over a behavior was sufficiently useful that a notation was developed specifically to capture and use such information. (see section 4.1 and Appendix C).

Attributes	State	Operations
range (360 deg) minSpeed (0 rpm) maxSpeed (75 rpm) direction (both) stepSize (9 deg)	speed position direction	turn setDirection setSpeed getDirection getPosition

Table 2.2: Some attributes, state information, and operations for the rotation behavior. The values after the attribute names are the values used for the Rototack.

In summary, behaviors are an important element of CECIs, more important in many ways than the computation. The crafter is more likely to be concerned with the exact color of a color-change behavior than the clock speed of the underlying processor. After all, the behaviors touch on the physical world that crafters are used to dealing

with, while the computation is buried, hidden by the physical and functional structure of the CECIs. However, the computation is what controls the behaviors, and gives a CECI its flexibility and usefulness.

2.3.2 Computation

A CECI's form and behaviors encapsulate the computation both physically and functionally, limiting the CECI's access to the outside world. The computation's inputs and outputs consist solely of the information sensed by behaviors, actions produced by behaviors, and the CECI-to-CECI communication system. The first two let the computation view and manipulate the world in a very constrained way; the last allows it to coordinate its activities with other CECIs.

The CECI's software internally mirrors this structure. It has an "outside" layer consisting of device and communication drivers which manage all input and output. Encapsulated by the drivers is the kernel, which is responsible for overall management of the computation and the execution of user programs. This structure is described in more detail in Section 4.2.

The kernel must schedule user program instructions and system tasks such as device management, event polling, and handling communication requests. These are placed into fixed length execution slots called cycles. Though somewhat wasteful of computational resources, fixed size slots greatly simplify the management of time dependent tasks. This reduces the need for hardware timers and allows simpler hardware to be used to implement a CECI's computation. If no system task has consumed a cycle's execution slot, the kernel executes a user program instruction. More details on this process and how it manages tasks can be found in Section 4.2.3.

The kernel is implemented as a virtual machine, which tightly integrates interpretation of user program instructions, task management, and scheduling. The integration reduces overhead, leaving a maximum amount of computing time for device

driver operations. The use of a virtual machine to execute user programs has several advantages, including insulating users from changes in the underlying hardware, reducing the memory requirements of user programs, and allowing complex domain-specific operations to be implemented as single instructions. The disadvantage is lack of speed. For example, the Rototack executes about 150 user program instructions per second. This is very slow compared to high-performance workstations, but is sufficient for the prototype CECIs. Future CECIs that require additional computing power will also require more powerful hardware or a machine-dependent compilation technique.

2.4 Summary

Computationally Enhanced Craft Items (CECIs) are created by taking a traditional craft material and adding behaviors and computation. The behaviors are specific ways of interacting with the world such as rotation, switching, and light sensing. A CECI's computation controls its behaviors, and that computation is in turn guided by a user program. Thus it is the user— a crafter—who is ultimately in control.

As part of the investigation of CECIs, three kinds of prototypes were created: the programmable hinge, the smart tile, and the Rototack. These prototypes exhibit progress in improved integration, communication, match to the non-enhanced version of the craft material, and sophistication of construction. The final prototype, the Rototack, was sufficiently robust to be used in limited user tests.

The enhanced part of a CECI, separated from the physical craft item, can be thought of as a set of behaviors surrounding a core of computation. These behaviors, along with the CECI-to-CECI communication system, are the sole inputs and outputs of the computation, which in turn manages and schedules the tasks that control the behaviors. The computation is also responsible for executing the user programs which ultimately determine the CECI's actions.

Chapter 3

Programming

In Chapter 1 the research question was divided into two areas of investigation: the development of computationally enhanced craft items (CECIs) and how such items could be programmed. The previous chapter focused on the structure and design of CECIs. This chapter focuses on the programming process and the tools necessary to support that process. The next chapter will describe the technical solutions developed to support both the design of the physical craft items and the programming system and tools.

The programming of craft items is different in many ways from other kinds of programming, and the language and tools developed for it must reflect those differences. This is evident at several levels: crafters cannot be assumed to have prior programming experience, the programs are small and concretely expressed, and the object being programmed has few user interaction capabilities—too few to support programming directly. While a number of existing programming systems share at least some of the attributes of CECI programming—most notably those for microcontrollers and small control computers—they tend toward traditional approaches to development, even when those approaches are not appropriate for the users or the kinds of programs being written.

The programming language and tools developed for CECI programming make use of as many traditional techniques as possible. These techniques—such as text-

based languages and integrating editing and compilation—are well understood, and using them reduces the development effort as well as providing a well tested approach. However, some modifications to existing techniques as well as new approaches are necessary to properly support crafter-programmers.

In Section 1.2 several important characteristics of CECI programming are listed. The first part of this chapter broadens and extends that list by talking about the elements that affect the design of programming systems in general, and by then presenting a usage scenario and a detailed discussion of the affordances found in the CECI programming process. This is followed by a description of the programming system developed for CECI programming. The programming system—actually a "programming environment"—is first discussed in terms of the programming language and then as a set of tools used to develop programs in that language.

3.1 Characteristics of Programming Systems

Programming systems come in many different forms, from the batch-mode punch-card systems common in the 1960's and 70's to the GUI based programming environments often used today. This process of advancement has introduced object-oriented, functional, and logic programming as well as other new ways of thinking about the content of a program. It has also introduced new ways of expressing programs, such as visual programming languages (VPLs) and programming by example (PBE).

However programming systems support the conceptualization or expression of programs, they have in common that they are essentially a form of communication—a way for programmers to express their intentions and to translate that expression into something the target computer can execute. Thus, every programming system has a structure that matches that of a simple communication system. This includes a speaker and listener—the programmer and computer—as well as a message with some content and form. These three basic elements are shared by all programming

systems, no matter how idiosyncratic, and so may serve as a basis for comparison and evaluation.

In this section these basic elements are examined as general background for evaluating the overall situation in which the programming of CECIs takes place. This background is also useful in understanding the design choices made for the CECI programming language and environment.

3.1.1 Programmers

The programmer is the speaker in the communication system of programming. The characteristics of the programmer are critical to the design of programming system that supports that communication. The characteristics of programmers that most influence programming system design is their level of programming experience and domain knowledge. Of these two the level of programming experience has the most impact.

Programmers with little or no programming experience need different kinds of support from programming systems than experts [49, 87]. Systems that must support both novice and expert programmers need to find ways to assist novices without alienating more experienced programmers [48, 97]. The issues that impact this are the complexity of the programming language, the type and availability of help, and the use of external libraries. These effect the “friendliness” of a programming language—the perception of how difficult it is to learn to use and how much assistance it provides when there are problems. Novices prefer a more friendly system, while expert programmers will tolerate very unfriendly systems in exchange for speed and power.

Another reason a programmers level of programming knowledge is so important to programming system designers is that novices are much more open to novel environments and languages than experienced programmers. Novices must learn a system from the ground up, while experienced programmers make use of their knowl-

edge of existing systems. A novel programming environment is no more difficult for a novice to master, but fails to take advantage of much of the existing knowledge of the expert. In fact, many important innovations in programming systems were originally designed for novice programmers. For example, Smalltalk and Logo were originally intended for children, and BASIC for college students. Programming by Example systems were intended for end users. Most visual programming languages are intended for children or novices. Section 5.1 discusses some of these systems.

The programmer's domain knowledge is also important to the language designer. A programmer with strong domain knowledge can more easily take advantage of primitives and structures that mirror the domain. This is particularly important if the programmers are novices, who can use their domain knowledge to understand programs that are written using domain terms.

3.1.2 Form and Media

The form for communication between the programmer and target platform is a program. This is not always obvious, as some systems blur the lines between different programs, expose only part of the program to the user, or otherwise hide the fact that a program is being generated.

Programs may be created and represented in many ways. In the past, programs were written in assembly language or a sequence of octal numbers and might be stored on punch cards or tape [53, 90]. Currently, there are a wide variety of ways to create and store programs, although text-based descriptions—usually written in a high-level language and stored in computer files—are by far the most common. Some alternative forms for programs are graphical representations, mixed graphics and text, programs expressed by demonstration and stored in a variety of ways, and programs generated from natural language input.

However the program is expressed, the characteristics of its content are also an

important factor in the design of the programming system, particularly with regard to the expected size and complexity of the programs. Large programs must be broken down into pieces small enough to be understood and written by a single programmer and to provide a way for multiple programmers to work on it simultaneously. Thus, a programming system for large, complex programs must stress modularity and other complexity control mechanisms above all else.

Smaller programs do not need to be broken into pieces to be written, understood, or maintained. The programming system may still support various kinds of modularity because there are reasons to do so even when it is not strictly necessary, such as Logo's use of subroutines (called "words") to build a vocabulary with which to solve a problem [91, 92]. Thus programming systems designed for smaller programs are not forced to include modularity at the cost of other features, giving them a larger range of choices in the structure of programming notation and tools. For example, spreadsheets—which are certainly the most successful end-user programming systems [85] and arguably the most successful programming system of any kind—are modular, but the modularity is tied to their underlying grid structure. This modularity is not designed to help the programmer understand large, complex programs, but to support the underlying structure of the spreadsheet.

While there are situations in which a program is used just once, more commonly programs are kept and used repeatedly. When this is the case it is often necessary to repair or modify the program from time to time. A program that is difficult to read, or has no human readable representation at all, cannot be maintained. Most programs need to be represented in a form understandable—at a minimum—by the original programmer. The larger and more diverse the group that will need to read and understand a program, the more comprehensible it must be.

3.1.3 Target Computer

A designer of a programming system must be concerned with the target platform's capabilities and the context in which it will be used. Of direct interest is the capability to support programming tools. This is influenced by the processor speed and memory needed to handle compilation, editing, a help database, and other tools. It also includes the ways in which the target platform can interact with the user. For example, a programming system on a keyboardless PDA will be quite different from one on a powerful workstation.

The target computer may be so computationally impoverished that it is unable to support programming in any form. In these situations a separate programming platform must be used, eliminating the problems associated with limited resources, but adding its own set of problems involving communication between the two systems, testing, and longer feedback loops for programming changes. Some separate programming environments that exhibit these problems are programming systems for single chips [82, 121] and small control computers [29, 74, 75].

A separate programming platform also makes sense in situations where the target platform needs all its resources to perform some other task. This will happen both in very small processors used in applications where they are barely sufficient and in extremely powerful systems like supercomputers, whose time is too valuable to waste compiling code. In these situations the programming system designer may choose to trade efficient use of a programmers time for increased program efficiency. This can be done by, for example, choosing to have the programmer deallocate memory rather than using garbage collection or by using a language that closely mirrors the machine architecture.

3.1.4 Summary

Designers of programming systems must take into account the programming and domain knowledge of programmers, the form and media used to express those programs, and the characteristics of the programming environment. The next two sections define these characteristics for a programming system for CECIs, first informally as a use scenario and then more formally by listing the affordances of CECI programming. The design decisions make to handle those characteristics and affordances are described in section 3.4.3

3.2 A Programming Scenario

This section presents a scenario to illustrate how a CECI is programmed using the notation and tools developed for that purpose.

3.2.1 Getting Started

Among the many craft objects adorning a particular craft shop is an attractive wooden turntable divided into three compartments in the shape of pie wedges. Each compartment is filled with small craft objects, usually primitive cloth dolls or holiday decorations, demonstrating the kinds of projects customers can build using kits and materials for sale in the shop. They also serve a secondary—but important—role as proof of the competence of the shop owners who designed and built the objects on display.

The turntable was stored in the front window of the store, an area visible from every side. A problem arose when it was moved to a corner behind the counter. In its new location only one of the three sections could be seen at a time. The original solution was to have the person behind the counter remember to turn it regularly to display the crafts in the different sections.

In a world filled with CECI, other solutions become feasible. A Rototack is a source of programmable rotation, and could provide a physically simple and (hopefully) inexpensive way to automate the turntable. The crafters who own the shop need to describe the actions they want performed, translate that into a program, and make sure the program does what they want.

After some thought the crafter who wants to modify the turntable comes up with the following informal description:

> Turn the tack head so the turntable moves one-third of a rotation, then pause for a few minutes. Repeat this until turned off.

Given the crafter's skill and enjoyment of such work, the physical construction is likely to happen first. The Rototack is mounted against the edge of the turntable. Figure 3.1 is a diagram of how this might be done.

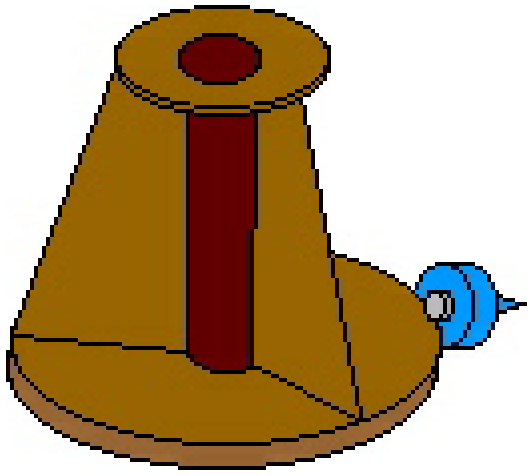


Figure 3.1: The Rototack is mounted near the edge of the turntable and drives it by turning a small rubber tire against the rim. Note that the Rototack will have to turn many times to rotate the turntable one revolution.

The layout is a rim drive, using friction to transmit rotation between the Rototack and the turntable. The advantages to this are that it requires no modifications to the (rather attractive) turntable, and reduces the torque needed by the Rototack. The disadvantage is that the relationship between the number of rotations of the Rototack and

the number of rotations of the turntable is not obvious. Careful measurement would provide a good approximation, and certainly the owner of a craft store that specializes in fabrics would understand the need to measure carefully. However, let us assume no measurement or calculation, just eyeballing and estimation by someone who is used to building physical artifacts.

Now that the physical structure of the object is in place, the crafter moves the entire assembly to a table containing the shop's computer, attaches the computer to the Rototack and starts the programming environment. Her next step is to rough out program, make it work, then try it on the "enhanced" turntable.

3.2.2 First Attempt

Now the focus of the crafter moves from the physical world to the computer screen. On startup, the programming environment presents a small menu window with buttons to start basic tasks: getting language help, looking for an existing program, modifying a program, or creating a new program. Here let us assume that the user wants to modify a similar program rather than writing a completely new one, so she looks for an existing, appropriate program to modify.

She opens a window called the *behavior browser* (see Figure 3.2). It shows the behavior of up to five programs at a time with a small schematic-level display of the running program (see Section 4.4). The display allows users to see what each program does without the need to read and understand program code or rely on vague or non-existent comments.

In this example, the window does not immediately show any programs that are close to what the crafter wants, so she uses an arrow button to see more programs. One of the new set of programs seems close to what she wants. Called "quarter" it turns one-fourth of a rotation, then pauses for a few seconds.

Having found a candidate program, the crafter now wants to examine it more

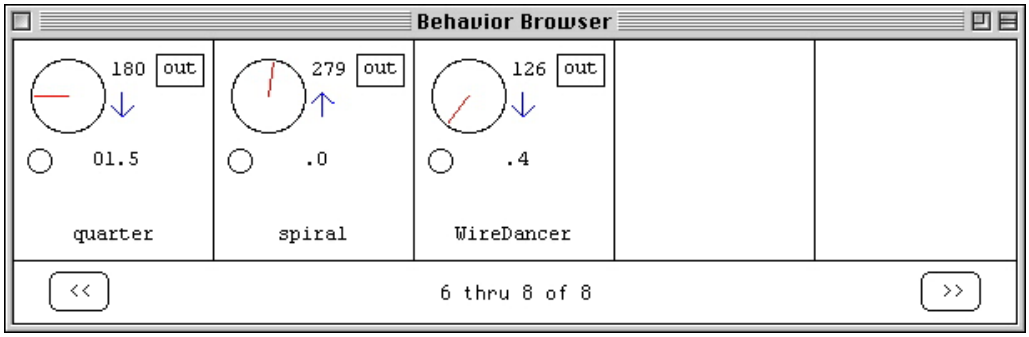


Figure 3.2: A behavior browser window. Each of the five tiles displays an active animation of a program. The arrows at the bottom are used to scroll through the library of programs.

closely and understand it sufficiently to make necessary changes. The user has several choices: she could use the menu window to open the program, but being somewhat familiar with the programming environment she knows that clicking on the name of the program in the browser is a shortcut that opens the program in the editor-animator. The editor-animator window with the original “quarter” program is shown in figure 3.3.

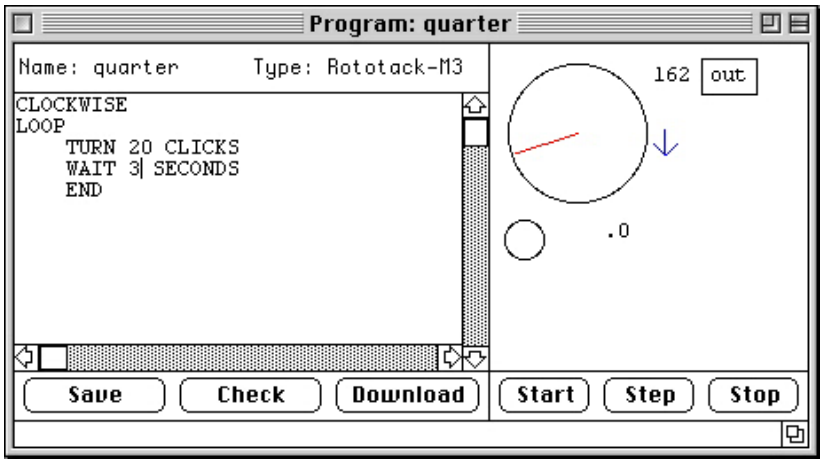


Figure 3.3: The editor-animator tool window showing the original ‘quarter’ program. Note the controls along the bottom to save, check, and download the program. The right hand side is used to animate the current program.

The user starts by looking at the program code, trying to understand what the

program does and how it works. To check her understanding she clicks on the “start” button at the bottom of the window, which starts an animation of the program in the left side of the editor-animator window. This animation is a larger version of what she saw in the behavior browser. The only differences here are the level of detail, the focus on a single program rather than a set of programs, and the user’s greater level of understanding now that she has seen the program code.

At this point the user believes she understands the program well enough to modify it. She understands that the Rototack will turn many times for each rotation of the turntable and she knows the pause time must be longer. The resulting program is:

```
CLOCKWISE
LOOP
    TURN 80 CLICKS
    WAIT 1 MINITE
    END
```

The user now wants to see how this program works. To the user this piece of text is the program and “compilation” is a mysterious process they are neither interested in or comfortable with; the programming environment compiles programs and reports errors when a compiled program is needed. The user’s next action, starting an animation, is one of those times when compilation is needed.

The user clicks on the start button again to begin animating the modified program. The existing animation (which was still running) stops and the compiler tries to compile the program. Finding a syntax error—‘minite’ is unknown—it reports the error by inserting the error message directly in the text editing area, highlighted for visibility, and reports the problem in a feedback area on the bottom of the window. Figure 3.4 shows how this reporting works.

The problem is simple and easily corrected. The user fixes the program and presses the start button again. After watching the animation for a few minutes and performing a hand simulation by moving the turntable manually, she decides that the

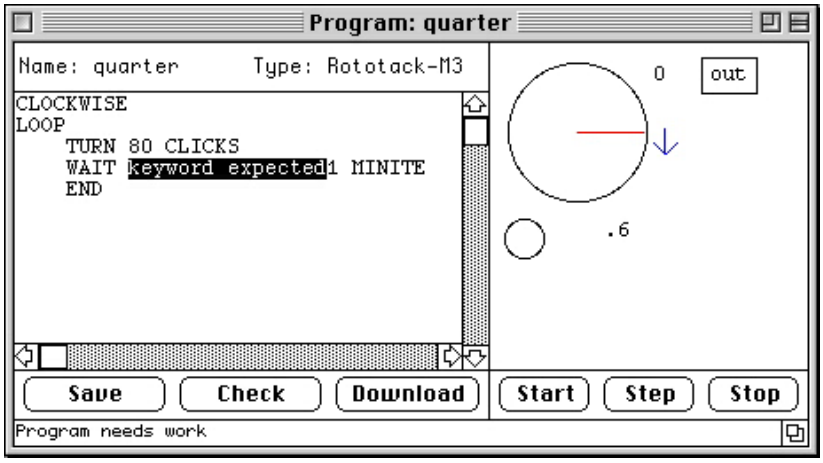


Figure 3.4: The editor-animator window reporting a syntax error. Note that the error message is highlighted so the programmer can backspace or type over it, leaving the faulty code in place. Also note the message in the feedback area reporting that the program needs attention.

delay time is not long enough, and that the number of turns of the Rototack is probably insufficient. She modifies the program and presses the download button to move the program to the attached Rototack.

```
CLOCKWISE
LOOP
    TURN 100 CLICKS
    WAIT 3 MINUTES
    END
```

Since a download needs a compiled program, the compiler is automatically invoked. This time, there are no syntax errors, and the programming system starts the download process.

3.2.3 Refinement

The Rototack-enhanced turntable is now ready for a physical test. The crafter turns it on and watches the action of the turntable. This is a likely point for improvement of the physical setup, such as ensuring sufficient traction between the tire on the Rototack and the rim of the turntable. Once those problems are solved, the user will

also be very familiar with a shortcoming of the program—namely that the number of turns by the Rototack is only about half of what it needs to be to rotate the turntable the correct distance. A few more iterations of writing and testing produces the following program.

```
CLOCKWISE
LOOP
    TURN 198 CLICKS
    WAIT 4 MINUTES
    END
```

This works well enough for this application[1] .

3.3 Affordances of CECI Programming

CECI programming occurs in an environment that includes the CECI, the crafter, the craft project, and tools for users to communicate programs to the CECI. This model leads to the characteristics of CECI programming listed in Section 1.2. In this section those characteristics are refined and extended into a collection of affordances that are unique to CECI programming. These affordances are:

- Analogs to Familiar Objects
- Small Programs
- Limited Scale of Operation
- Small Set of Concrete Primitives
- Wide Range of Programming Knowledge
- Programs as Elements of Communication
- Separate Programming Environment

[1] Actually, experiments suggest that the turntable would require a regular (at least once a day) adjustment as it "drifts" such that parts of two sections are shown rather than just one.

The remainder of this section explains and justifies these affordances.

3.3.1 Analogs to Familiar Objects

CECIs are traditional craft objects with added behavior and the computation to programmatically control that behavior. Despite their additional capabilities they are still craft items and should be able to take the place of their non-enhanced counterparts. The users are crafters who are familiar with traditional craft items; thus the physical structure of a CECI is familiar to the user.

The physical similarity may be the only similarity for items like the Rototack, which has a new and perhaps unnatural behavior added. Other types of CECIs, like the programmable hinge, will be functionally as well as physically similar. Still others, like the smart tile, can appear to be different variations of the same craft object—in this case differently colored tiles—that are otherwise functionally and physically identical to their non-enhanced counterparts.

3.3.2 Small Programs

CECI programs are small. There is no technical reason why this must be the case[2] but it is difficult to come up with a situation where a large program would be useful. Take, for example, animated holiday displays and model railroad layouts, two existing crafts where computational control is sometimes used (see section 2.1.2). Focusing on the activities of a single object in these displays shows a fairly simple set of behaviors. A Santa Claus that nods its head, a train engine that starts and stops at one point each time around the track, and a turkey perpetually running in a circle away from an ax-wielding farmer are all simple behaviors that can be described in a few lines of program code. Though our experience with CECI programs is limited, nearly

[2] The current prototypes are limited to about forty lines of code. A limitation that can be overcome with a modest increase in cost or by the relentless march of Moore's Law.

a hundred programs have been created for the prototype and hypothesized CECI, and the largest of these programs is thirty-five lines. CECI do not need to have complex behaviors and so do not need large programs.

It could be argued that the reason there are no complex behaviors for existing craft objects is that current automation and programming techniques make them too hard to implement. If that is the case, then the simplicity of using CECIs will allow crafters to more easily produce complex behaviors, and therefore they will need to write larger programs. It is impossible to predict whether this is the case, at least without a considerable amount of experimentation. However, since there is room for maximum program sizes to double or triple and still be considered small, small programs will be a basic attribute of CECI programming, at least for the foreseeable future.

3.3.3 Limited Scale of Operation

Small embedded computers such as microcontrollers have a number of attributes in common with CECIs. They are small, have limited program space, and their I/O is limited to a small set of operations. The most difficult problem in programming microcontrollers is the inability to see the program operate. The shift of voltage levels on tiny pins changing in thousandths or even millionths of a second are simply beyond anything humans are capable of sensing. Simulations of these operations that slow things down and map voltages to something a human can sense are of limited usefulness (see Section 4.4 for a more complete description of this problem).

CECIs are different. Since they are based on an existing craft material, they share the limitations on the size of those materials. This ranges from the small beads and stones used by jewelry makers to large pieces of wood used by crafters who make furniture, a size scale from millimeters to meters. Similarly, the time scale of the behaviors in a CECI will tend toward something the crafter can deal with directly. There is not much use for a Rototack that turns at thousands of RPM, or a hinge that takes months

to open. Thus the time scale of a CECI, like the size scale, is one humans can deal with directly.

It is more difficult to judge the user's ability to directly observe a CECI's behaviors. Certainly the prototypes listed in chapter 2 work with behaviors that humans can observe directly: motion, light, color change, and tactile feedback. Still, it is not hard to imagine a CECI magnet, whose level of magnetism could be changed programmatically. Even here one could argue that the force of a CECI magnet would fit the scale of the user. It is hard to imaging a crafter using a magnet, computationally enhanced or otherwise, that could produce a ton or more of force, or one that was limited to lifting a fraction of a gram. In general, even behaviors that the crafter cannot directly sense will tend to be of a scale that fits the user.[3]

3.3.4 Small Set of Concrete Primitives

The computation in a CECI consists of two kinds of primitives: general control primitives that are used to implement control structures and data handling, and primitives that control behaviors. The first set is small, largely because CECIs cannot do many different things, and so do not need complex data or control structures (see Sections 3.5 and 4.5 for details). The second set is based on operations on the CECI's behaviors.

The operations in this second set are concrete because they correspond to aspects of a behavior's actions in the real world. For example, a full set of operations for the rotation behavior is: turn, set speed, set direction, test speed, and test direction. These are all directly trigger a real world action or they examine or change a setting that affects that action. While some behaviors' are harder to observe than rotation they are

[3] Actually a high powered CECI magnet and high-speed Rototack could be combined to produce part of a simple MRI (Magnetic Resonate Imaging) system. This seems laughable until one realizes that the first prize in the 1997 Westinghouse Science Competition was a high school student who built a scanning tunneling microscope out of Legos, plasticine, and similar materials [21].

still based on real world actions. Only a few “internal” behaviors, such as data storage, have no direct physical effect.

Based on the behaviors identified so far, it seems that any single behavior is controlled by from one to five operations. Since CECIs are intended to be raw materials, they will tend toward few behaviors—no more than three in those we have built or considered building. Few, if any, types of CECIs will have more than a dozen or so primitive operations for controlling behaviors.

Although it is possible that the set of primitives that implement general control and data handling will be much larger than those used to control behaviors, it is still unlikely to be large when compared to traditional programming systems. One reason for this is that the majority of primitives in modern high-level languages are used for input and output [25], something that is limited in CECIs to a half dozen communication primitives and the primitive operations for controlling behaviors already discussed.

3.3.5 Wide Range of Programming Knowledge

There are crafters who are expert programmers, crafters who are computer-phobic, and crafters at every possible level of experience between those extremes. Most will fall somewhere in the middle, ranging from those who use computers only in very limited ways to sophisticated users who have “done a little programming”. Any programming system for CECIs will have to deal with this wide range of programming knowledge and experience.

3.3.6 Programs as Elements of Communication

In chapter six of her book *A Small Matter of Programming* [87] Nardi discusses the emergence of communities of end-user programmers when a programmable application is used in a social setting such as an office or lab. The same phenomenon is

likely to emerge in crafts that are practiced in a social setting, such as quilting groups, sewing circles, or craft classes.[4] However, not all crafting is done in groups; most are solitary activities pursued by an individual working alone. This is not to say that these crafters are completely cut off from a larger community. There are many books and magazines, and crafters meet face-to-face at craft fairs and stores. These too constitute communities, though somewhat different from those Nardi studied.

In any of these communities, programs can serve as a form of communication. This is particularly important in the more dispersed communities of the less social crafts, where the richness of face-to-face communication is unavailable. Short program listings are especially appropriate for print-based communication such as magazines and books. Such printed program listings were a common way to share programs in computer hobbyist magazines until the late 1980's, when the programs became to large to easily print or read. Some books from that period also included program source code, with a few being entirely devoted to program listings (such as [80]).

3.3.7 Separate Programming Environment

CECI cannot act as their own programming environment: they contain too few ways of interacting with the user, and more importantly, they offer no way to interact in abstract, symbolic terms. CECIs could be enhanced by plugging in additional hardware when programming was needed, increasing the user interaction capabilities to the point where the object could serve as its own programming platform. Once additional hardware must be attached for programming, there is no reason not to make those separate controls a full fledged programming environment. A separate programming environment need not have the same limitations as the CECI. This makes it possible to consider languages and development tools that do not fit the computational

[4] Craft classes rarely teach basic crafting skills. Instead they are aimed at experienced crafters who want to learn a new technique or how to use a new material. They are more peer-to-peer social groups than a classroom setting.

and user interface constraints of a CECI.

Some non-traditional programming systems use a form and media that change the kinds of user interaction needed to express a program. Most of these require modes of interaction or hardware support far outside that likely to be available on a CECI (such as the techniques described in [7] and [65]). However, programming by example (PBE) (see [24] for background) seems to require little more in the way of user interaction capabilities than those already present in a CECI, and so are worth a deeper look.

PBE systems create programs based on user examples of how the program should execute. To program a Rototack using this technique one would hopefully be able to simply pick it up[5] and turn it back and forth in the desired pattern, the object itself would then generate the program needed to create that pattern. Some further thought suggests some possible problems with this scenario:

- How does the Rototack know how many times to repeat a pattern?
- How does it know when to *stop* repeating a pattern?
- How can the Rototack infer conditional execution?
- What if one of the behaviors is too fast or too slow to be demonstrated effectively?

The problem of inferring loop constraints and conditionals is know in PBE literature as the problem of generalization. (see [77] and [117] for some discussion and examples). The problem of compressing or extending time is one that is inherent in PBE systems in general, but is more often a problem when applied to systems that interact with the physical world, where time and timing are often important elements of a program.

[5] Hopefully avoiding the spike that has savaged so many hands.

A less obvious problem is how users would create examples for programs that use behaviors that cannot be directly manipulated, such as the color change behavior of a smart tile. It would be necessary to add controls that users *could* manipulate to trigger these behaviors. Here we come back to the idea of plugging in additional hardware when programming was needed, though in this case that additional hardware may be minimal. Other reasons for deciding against PBE for CECI programming can be found in Section 3.4.3.

3.4 Design of a CECI Programming System

The design of the programming system was an iterative process. The initial prototypes used a general purpose control computer and its existing programming system. The mismatches of this programming system to the CECIs were then used as a basis for the design on the next attempt. The new ideas for programming were also used in the design of the next set of CECI prototypes. The process was as much one of discovery as purposeful design.

Although much of the design process was meshed with building and testing, certain goals were used as rough guides for the refinement process. These included the affordances listed in Section 3.3, a strong preference for simplicity over complexity, and a need to support as much of the crafting community as possible. Other guidance to the design process came from examination of existing systems that are similar to CECIs and from a number of important design choices that had to be made early on in the design processes.

This section starts by discussing the design goals. Then two types of systems, embedded systems and small control computers, whose programming is similar in a number of ways to CECI programming, are described. Finally, the most important design choices made for the CECI programming system are explained and justified. Section 3.5 and 3.6 discuss the specific design and implementation decisions, the first

covering the programming language and the second the set of tools created to enable and assist the crafter in the development of programs for CECIs.

3.4.1 Design Goals

It is important to conform to and, when possible, take advantage of the affordances listed in Section 3.3. However, these affordances are insufficient in themselves to fully guide the design of the programming system. Other goals come from the overall focus of the research and personal aesthetics. These include a preference for simplicity over complexity, making use of successful design elements from other systems, making the system usable by as many crafters as possible, and supporting a full range of programming activities.

The preference for simplicity over complexity is motivated by several factors. The first is that the basic question guiding the research described here is an attempt to find a *minimal* set of technical solutions for programming CECI (these are enumerated in the next chapter). Another is a preference for elegant solutions. While a simple solution may be neither minimal or elegant, a minimal or elegant solution will almost always be simple. A final reason to prefer simple solutions is strictly practical: simple solutions are easier to implement than complex ones.

These practical issues also apply to the goal of borrowing elements from successful existing designs. Unfortunately, it is difficult to be sure that an individual element of a larger design actually contributes to its success. Since only an entire design can be easily judged successful or not, individual elements of that design may or may not be advantageous in and of themselves. System success may occur because the overall design was successful *despite* that element, or because the aggregate benefited from useful interactions between the features, not any individual feature. To address this possibility, elements of other systems were considered only if they come from a system with parallels to CECI programming (such as embedded computation or small control

computers, as described in the next section), or which appear in a large number of successful systems.

For this research to have applicability, the programming system should support as much of the crafting community as possible, despite the wide range of programming skill and knowledge that crafters have. A programming system designed for the subset of crafters who already program, or who have degrees in computer science, would be both unnecessary and uninteresting. Likewise a system that only supported the most novice of programmers—while somewhat interesting—likely could not support those novices as they become more skilled. Thus the goal is to create a programming system usable by both experienced and novice programmers.

For similar reasons, the entire programming process must be supported, not just the creation of new programs. This process includes understanding programs, program modification, and debugging. Because CECIs are built into a physical system, their behavior must often be adjusted slightly to match the rest of the system. This "tweaking", while similar to program modification and debugging, often takes place while the object is in place in a larger system. Thus, special support for in situ program modification is needed.

3.4.2 Similar Programming Systems

There are a number of existing programming systems designed for objects and situations similar to CECI—similar in that they share at least some of the affordances listed in Section 3.3. This section discusses two of these, embedded systems and small control computers, that influenced the development of CECIs and their programming system.

An embedded system (also sometimes called "embedded computation") is computation placed inside a physical object and acting as a controller. This is typically done by building a microcontroller and related circuitry (for examples and details see

[43, 54, chapter 11]) into the object. Embedded computation is common, and there is an almost limitless set of examples. Two of the better known are replacements for mechanical and electrical controls in kitchen appliances and enhancements to toys.

While embedded computing bears a strong similarity to CECIs, the situations in which it is used are quite different. The programming of embedded systems is normally done by professional software engineers, and it is very rare for end users to have any meaningful control over the computation[6] . Designers of objects containing embedded computation must work with "raw" computation and program it using abstract operations that have no obvious connection to the object or the way it is used.

There are several reasons for the strong surface similarity between CECIs and embedded systems. One is that both involve placing computation inside of physical objects and another is that both use a separate programming platform. The third, and most interesting, is that CECIs are a type of embedded system. However, while the designer of a new CECI must use embedded computation, the crafter who will eventually program the CECI is doing something quite different.

Embedded systems influenced the design and development of the prototype CECIs mostly as an example of a situation where the programming platform was separate from the object being programmed. While this kind of separation is historically quite common—punch cards and coding forms were in use through the late 1970's and early 1980's—embedded systems are one of few areas of programming that combine that separation with more modern programming tools and languages.

Small control computers are another area where programming is separated from execution. These are tiny, general purpose computers designed to be used as controllers in much the same way as microcontrollers are used in embedded systems. One difference is that these small control computers are separate from the physical objects rather than being built into them. Another important difference is that some—like

[6] A famous—or perhaps infamous—exception is the minimal programming available on VCRs.

the Cricket [75], Basic Stamp [29], Handy Board [73], and Lego Mindstorms [69]—are designed for end users rather than professional programmers.

Because these computers are created for end users, and because some of them are used in educational settings, the programs for these systems often serve as a means of communication. Also, while they are unable to provide the level of domain specificity of a CECI programming system, small control computers can express their control primitives in real world terms. For example, the Cricket uses primitives like "on" and "off" to control its motor ports, while the equivalent commands in a microcontroller might be something like `BSET PORTB.1` and `BCLR PORTB.1`. Thus, despite their physical separation from the objects they control, the programming of small control computers has a greater similarity to CECI programming than embedded systems programming.

The languages, tools, and techniques used in embedded systems and small control computers were an important point of reference during the design and development of the CECI programming system. They provided both a source of ideas and, in some cases, a demonstration of how things should not be done[7] .

3.4.3 High-Level Design Choices

The affordances of the domain and other aspects of the design goals suggest a design that focuses on simplicity, use of domain knowledge, and tools to support users' tasks. A number of well studied programming techniques address these points. Primary among these are visual programming languages, programming by example, automatic programming based on informal specification, and systems that use text-based languages. Others—tangible programming for example—do not fit the design goals or there is as yet an insufficient depth of practical implementation knowledge.

[7] This was brought painfully to the fore by the use of embedded system development tools to construct the prototype CECIs.

In her wonderful book on end-user programming languages, Nardi does a through and telling critique of alternative programming techniques [87, chapter 4] including those listed above. It seems unnecessary to reiterate her points here, though a summary in her own words will give a flavor of her arguments:

> Many interaction techniques have been heralded as *the* answer. But upon further inspection we find that these techniques all have their limitations. And no single technique by itself addresses the semantic issues of designing a programming language that taps into user knowledge and leverages users' skills, interests, and familiarity with their domains to make programming more interesting and intelligible. However, any of these techniques may prove useful as a local solution to a problem of user interaction within a task-specific language.

As Nardi suggests, the programming techniques just mentioned have advantages and disadvantages, some of which are discussed in more detail later in this chapter. Two specific issues, one strictly practical and one based on the domain, made a text-based language a better choice for programming CECIs. First, text-based languages are easier to implement and create tools for because of the large pool of existing implementations, reference materials, and tools available for building such systems. The second is that text-based programs are more suited for use as a means of communication. While most craft work is a solitary activity, crafters share information via books, magazines, and in person at craft stores and shows. Many of these books and magazine articles are devoted to "how-to" information: how to use materials, techniques, and tools; usually in the context of a particular craft project. Magazines and books can easily include listings of textual programs—it was a common practice in computer hobbyist publications in the 1980's.

Programming by example systems are especially problematic for use as a means of communication, as they often produce no human-readable program at all. In the cases where they do, the correspondence between the program and the way it was created is not strong (see, for example, [50] and [94]). Visual programming languages

are awkward to use as a means of communication because a printed representation takes substantially more space and is harder to reproduce. Though the first problem is less important for CECIs, which tend to have small programs.

One alternative programming technique that rivals traditional text-based languages as a means of communication is natural language programming (or, more formally, automatic programming using an informal program specification where the informal specification is in a natural language). This approach has had some early successes in highly-focused domains, including the venerable LUNAR program [118], but has not seen the same level of success in general-purpose programming. Among other problems natural language programming systems, like PBE systems, cannot accurately infer loop constraints or conditional execution without considerable additional domain knowledge.

Thus the first important choice for the design of the programming system for CECIs is that it will be in the form of a textual language and related tools. While it would be possible to invent a new programming language syntax, it is both unnecessary and a disservice to the end user. Using an existing and well known language as a template both produces a tested, cohesive syntax and ensures that at least some crafters will have had experience with a similar language.

Modern versions of programming systems designed around text-based languages typically come with a set of interacting tools called an integrated development environment (IDE) such as those described in [44, 81, 83]. These include an editor, compiler, version control, debugging tools, and a way to execute the program without leaving the tool set. These environments are popular with programmers, even when the individual tools are inferior to other stand-alone versions of the same tool. The integration provides sufficient added value that limitations in individual tools are accepted.

The integration of tools comes in two forms. Most common is to share information across tools based on particular capabilities—thus an editor, used to view and edit

source code, may have an option to start a compilation on the source code it contains. The other kind is task based integration, where a single tool has several capabilities which support a single task or group of related tasks. Examples of this are the better debugging tools, which integrate program execution, tracing and breakpoints, an editor, and a compiler to support debugging tasks.

Task based integration is most useful when it supports everyone's approach to that task, something that becomes more difficult as the tasks become less focused. When the task is not a common one, or the designer has not anticipated a user's approach to a common task, multiple tools supporting distinct capabilities is the more effective style of integration. Note that this applies to more than programming tools. A crafter who works with fabrics knows the usefulness of a sewing machine for most jobs, but also understands that a needle and thread are more flexible and can do jobs that a sewing machine cannot handle.

Programming tools for CECIs should provide integration; task integration where it can reasonably be expected to be useful, and capability-based tools that share information when it is not.

One of the general capabilities listed above for an IDE is the ability to execute programs without leaving the programming software. This is a problem for a CECI programming system, where the development platform is separate from the object that executes the program. As mentioned in the previous section, this problem is shared by embedded systems and small control computers. These systems address this problem in two general ways: making the transfer of programs to the execution platform as easy and fast as possible, and by providing a simulation of the execution platform.

The first approach is generally employed by programming systems for small control computers, while the second is commonly used by IDEs for microcontrollers. The split makes sense as the actions of a small control computer are more easily examined directly than those of a microcontroller. In addition, a microcontroller is usually

built into a larger device, making it even more difficult to examine its behavior. Thus, even if the transfer of a programs to a microcontroller were a trivial process, it would still be difficult to observe its behavior when running the program. (This is discussed in more detail in Section 4.4).

CECIs are somewhat different than either small control computers or microcontrollers in this regard. Their actions are even easier to observe directly than those of a small control computer, yet like the microcontroller they are often built into larger structures. In a sense neither simplifying transfer of programs or providing a simulation are complete solutions for CECIs. A decision of which approach was better was avoided by attempting to provide both options. The communication was made as easy as possible given the constraints of the prototype CECI hardware. A variation of a simulator was also provided as part of the programming tools. See Sections 4.3 and 4.4 for more specific information.

In summary, the high-level choices made for the design of the CECI programming system were to use a text-based language, use an integrated development environment with task-based integration where appropriate, and to ease access to program execution by both streamlined communication and a simulator.

3.5 Programming Language

Even given the high-level choice of a textual programming language, there is a great deal of freedom in the actual choice of syntax, control structures, data types, and the way primitives are used. When possible, these choices are justified by comparison with obvious alternatives. In some cases that is impossible. At times the choice was essentially a random one between multiple, equally attractive (or painful) options. The result does fit the overall design goals quite closely, and user tests show it to be reasonably usable (see section 6.1), so it must be considered at least a qualified success. While considerably more practical experience with the language will be necessary to prop-

erly evaluate its effectiveness, section 3.5.5 applies some standard objective criteria to some of the more noteworthy language features.

This section covers the important elements of the core CECI programming language. The overall structure of the basic language is described first, and then extended by discussing details of data handling, operations, control structures, and special function support. The section concludes with an evaluation of the programming system based on objective criteria. A more detailed description of the language can be found in Appendix A and details of its construction can be found in Section 4.5.

3.5.1 Basic Structure

Section 3.4.3 discussed the decision to use a text-based language based on an existing programming language. While no existing programming language exactly fits the needs of CECIs—particularly the support for very specific primitives tied to the operations on the behaviors—several suggest themselves as a sort of syntactic template, a basis for the overall style and structure of the language. In particular the C language, BASIC, and Logo have all been adapted for use with small control computers, and Forth was designed specifically for control applications. Of these, BASIC seems like a marginally better choice. The major advantage of BASIC reflects another design choice—the elimination of constructs to support modularity, including subroutines, functions, classes, and the related call mechanisms. Since CECI programs are small, there is no real need for even the minimal modularity supplied by subroutines. This choice seems fairly radical—going against not just the march of mainstream programming language practice, but also teaching languages such as Logo, which use modularity as a thinking tool. As a hedge, subroutines were implemented in the language and then disabled. Thus far, writing programs for CECIs suggests that subroutines are

unnecessary.[8] BASIC is better in this case because it is one of the few languages still in use that works well with no subroutines, functions, or call libraries.

The choice of BASIC may require some additional justification. It has a rather poor reputation in the Computer Science community, and not without reason. Papert's summary in *Mindstorms* [91, page 35] is classic:

> An example of BASIC ideology is the argument that BASIC is easy to learn because it has a very small vocabulary.... Its small vocabulary can be learned quickly enough. But using it is a different matter. Programs in BASIC acquire so labyrinthine a structure that in fact only the most motivated and brilliant ("Mathematical") children do learn to use it for more than trivial ends.

Papert is exactly right. BASIC, particularly the dialects available in the late seventies and early eighties, is not suited for writing large, complex programs. However, CECI programs are neither large nor complex, and avoid being trivial only because of their connection to the physical world and their incorporation into larger craft projects. Thus Papert's objections do not apply in this case, while the advantage of a small vocabulary remains.

But even BASIC, whose different varieties support a range of syntactic styles and features, must be modified to work with CECIs. One part of this involves the elimination of keywords and functions to support terminal and file interaction, while another comes from the need to directly support primitives for manipulating behaviors. Not only are these primitives unlike those supported in existing dialects of BASIC, but they vary from CECI to CECI. For example, compare this program for a Rototack:

```
CLOCKWISE
LOOP
    TURN 10 CLICKS
    REVERSE
    WAIT 1 MINUTE
```

[8] A number of features were disabled this way. It seems interesting that it was not necessary to enable any of them. I believe this reflects a common timidness in removing features from programming languages.

```
    END
```

to this program for a smart tile:

```
COLOR OFF
LOOP
    COLOR FLIP
    WAIT 1 MINUTE
    END
```

While there are similarities, they use different primitives to manipulate their respective behaviors. In addition, the behaviors used for one will not even be available on the other. Certainly a tile should not be able to “turn”.

The languages for the Rototack and smart tile are different. The difference comes from the support of the individual behaviors, which require different operations. But there is no reason for the rest of the language to differ between items. Indeed, there are several reasons why it should not, most notably to reduce the need to learn a completely new language for each type of CECI. So the CECI programming language is really a family of similar languages, with a core that handles common operations and control structures, and additional specific primitives to handle the behaviors of each type of CECI. The mechanics of dealing with these different languages are discussed in Section 4.1.

The Rototack language quick reference sheet is given in Figure 3.5, and shows the separation between Rototack-specific and core language elements. A more complete description of the Rototack language can be found in Appendix A.

3.5.2 Data and Operations

CECIs do not need to work with data to any great extent. They are designed to *do* things rather than to *know* things. Data is limited to flags, counting, and other ways of tracking the item’s overall behavior. These activities can be supported using a very limited set of data types and operations.

Function	**Command/Syntax**	**(*Comment*)**
Rotation	`TURN n CLICKS`	`(1 click = 9 degrees)`
	`CLOCKWISE`	
	`CCWISE`	
	`ISCLOCKWISE`	`(predicate)`
Two Color Light (LED)	`LED OFF`	
	`LED RED`	
	`LED GREEN`	
	`LED AMBER`	
In-Board Sensor	`ISIN`	`(predicate)`
Communication	`SEND msg TO object WITH argument`	
	`ISMSGIN`	`(predicate)`
	`MESSAGE`	`(value is msg ID)`
	`ARGUMENT`	`(value is msg argument)`
	`ME`	`(this objects ID number)`
Data	`A,B,C,D`	`(variable names)`
	`:=`	`(assignment)`
	`PARAMETER name = value`	
	`CONSTANT name = value`	
Operators	`=, <>, >, <, <=, >=`	`(comparison)`
	`+, -`	`(binary or unary)`
	`AND, OR, NOT`	`(bitwise and boolean)`
Conditionals	`IF cond THEN statements END`	
	`IF cond THEN statements ELSE statements END`	
Iterators	`LOOP statements END`	
	`REPEAT n DO statements END`	
	`WHILE cond DO statements END`	
	`WAITUNTIL cond`	

Figure 3.5: The quick reference sheet for the Rototack programming language. The sections above the double line are specific to the Rototack.

Booleans, and boolean expressions, are required to define conditions under which actions repeat or change. Integers are needed for counting and are also an adequate way to represent simple state information. For a CECI whose behaviors are simple interactions with the physical world—as is the case with all the prototype CECIs—these two types are sufficient.

A further simplification can be made by eliminating booleans as an *explicit* type. This can be done syntactically by limiting booleans to the result of comparison operations, or by expressing booleans as integers as is done in C, C++, Forth, and a number of other languages. Despite a number of minor problems with the latter approach,[9] the flexibility and simplicity of representing booleans as integer values make it a good choice. While this negatively impacts type safety, such issues are not terribly important in very small programs. See section 3.5.5 for a more complete discussion of this.

Certain types of behaviors, such as those that would be produced by a CECI with an LCD or other device capable of displaying symbolic information, might need additional data types. To be more specific, behaviors whose state or state transition information cannot be adequately represented as integers or booleans need the support of additional data types. These can be introduced and managed by the software library that provides compiler and animation support in the programming environment (see Sections 4.1, 4.4, and 4.6). By these methods the core language can be limited to an integer data type, which for the prototype CECIs is stored as a single byte. The operations supported on this single type are a full set of comparison and boolean operations (which double as bit manipulations) as well as addition and subtraction. More information about these operations can be found in Appendix A and Figure 3.5.

Variables in this programming language require no declaration and are named with single character identifiers. The reason for this is to eliminate the need for the user to create names, and to make it obvious which identifiers in a program are variables.

[9] Among them my own personal distaste for the practice.

The loss of documentation support in this approach is difficult to judge. There are a small number of individuals sharing programs for the prototype CECIs, and in subject tests users worked with programs containing at most one variable. This decision should be reevaluated when a larger pool of shared programs whose readability can be tested is available.

While variable names are limited to single characters, named constants can be any identifier not used as a keyword. Also, unlike variables, constants require a formal declaration. The following program illustrates the use of variables and constants in the CECI language:

```
;; Program uses one variable A, and two named
;; constants START and STOP
constant START = 5 ;; Smallest arc 5*9=45 deg
constant STOP = 30 ;; Largest arc 30*9=270 deg
A = START ;; Initialize variable A
loop
    turn A clicks
    reverse
    A = A + 1
    if A > STOP then A = START end
    end
```

A parameter is a variation of a constant that can be adjusted without using the full programming environment. Parameters are declared like constants, but use the `PARAMETER` keyword. The program above could be modified to contain parameters rather than constants by changing the declarations:

```
parameter START = 5 ;; Smallest arc 5*9=45 deg
parameter STOP = 30 ;; Largest arc 30*9=270 deg
```

This allows the start and stop values to be changed using a small device that can alter their values. This devices has been prototyped as a physical mockup and protoboard-level circuit, but a full working prototype has not been constructed. Parameters provide a way to “fine tune” or “tweak” a program independently of the

full development environment. This is particularly useful “in the field”—to adjust the behavior of a CECI that has already been built into a larger object[10] .

3.5.3 Control Structures

The language includes a minimal set of basic conditional and iteration constructs, and a set of less common constructs of the type found in languages for small control computers. All of these structures—with the exception of `WAITUNTIL`—assume a code block terminated with an explicit `END`. This minimizes the different ways to write code for a block and eliminates the dangling else problem, though it does require an `END` even when the block is a single statement.

Conditional operations use an `IF-THEN` or `IF-THEN-ELSE` construct:

```
;; Example 1 IF msgIn AND message = OpenMsg THEN
    OPEN COMPLETELY
    WAIT 2 SECONDS
END

;; Example 2 IF A > MaxValue THEN
    A = 0
ELSE
    A = A + 1
END
```

Note that in the if-then-else construct, the `ELSE` serves to terminate the first block. This is the only place an explicit `END` is not needed to terminate a block of code. Since the language is not line sensitive, the second example could be written as:

```
IF A > MaxValue THEN A = 0 ELSE A = A + 1 END
```

Basic iteration is provided by a `WHILE` loop:

```
A = 1
```

[10] My favorite example of this is adjusting the program of a tile in a mosaic mounted on the wall of the third floor of the *outside* of the building.

```
While A <= 10 Do
    Turn A clicks
    Wait A+A+5 tenths
    End
```

Two variations of the basic WHILE loop are provided: an infinite loop and a wait for condition loop. The infinite loop is exactly that, a loop with no termination, and is identified with the LOOP keyword. The wait until condition loop waits for some externally caused change, such as a message from another CECI or a change to an input behavior, and is identified with the WAITUNTIL keyword. The following programs illustrate both an infinite and wait for condition loop:

```
loop
    waituntil inboard
    while inboard do turn 1 end
    end

constant Open = 10
constant Close = 11
loop
    waituntil msgin
    if message = Open
        then open completely end
    if message = Close
        then close completely end
    end
```

The remaining control structure is a REPEAT loop, which will execute a block of code a specified number of times. It does not store an index in a variable, so it does not act like the more common FOR loop. The repeat value can be an expression or a constant.

```
REPEAT 10 DO FLASH END

REPEAT A+1 DO
    WAIT 8 TENTHS
    TURN A CLICKS
    IF A = 5 THEN CCWISE END
    END
```

Behaviors are given sufficient access to the compiler and the computational machinery they work with—both on the CECI itself and its simulated counterpart in the programming environment—to implement new control structures. This might prove useful for devices with special control needs, such as the faster response time provided by interrupt handlers or the ability to iterate over large amounts of data.

3.5.4 Other Support

In addition to the basics of handling data and program control flow, the core language supports primitives for pausing program execution, generating random numbers, and communicating with other CECIs. The communication is based on message passing between CECIs. The prototype CECIs use a three-wire bus for message traffic, though the protocol is flexible enough to be used with wireless media (see Section 4.3). In the programming language message passing is supported by a single `SEND` command and several functions to access incoming messages. For example, the following code would send two messages:

```
SEND OpenMsg TO MyHinge
SEND TickMsg TO Tile53 WITH A
```

Where `OpenMsg`, `MyHinge`, `TickMsg`, and `Tile53` are all named constants. `OpenMsg` and `TickMsg` are message identifiers, while `MyHinge` and `Tile53` are identifiers for specific objects. The value after the optional `WITH` keyword is the message argument.

Each CECI has a unique identification number that must be used when sending a message to that object. A message sent with a zero as an object identification will be broadcast; every CECI that sees the message will receive it, even though some programs may ignore it.

Receiving messages is somewhat more involved. When a message is received the CECI firmware puts the message identifier and argument in a buffer (overwrit-

ing information from the previous message) and sets a flag. The functions MSGIN, MESSAGE, and ARGUMENT access the flag, message identifier, and argument value respectively. The following Rototack program is designed so that the rotation can be controlled from another CECI.

```
CONSTANT TurnMsg = 21
CONSTANT DirMsg = 22
LOOP
    WAITUNTIL MSGIN
    IF MESSAGE = TurnMsg THEN
        TURN ARGUMENT CLICKS
        END
    IF MESSAGE = DirMsg THEN
        IF ARGUMENT = 0 THEN CLOCKWISE END
        IF ARGUMENT = 1 THEN CCWISE END
        END
    END
```

A common operation in a control program is to wait for some period of time. This operation is supported by WAIT, and comes in varieties for tenths of a second, seconds, and minutes. To wait for 59 minutes, 59.9 seconds requires the use of all three varieties:

```
wait 59 minutes
wait 59 seconds
wait 9 tenths
```

Pseudo random numbers are generated with the RANDOM keyword, which returns a random integer between zero and one less than the argument value. The following program uses this operation to turn a Rototack randomly:

```
LOOP
    WAIT RANDOM 5 SECONDS
    TURN RANDOM 40 CLICKS
    IF (RANDOM 2) = 1 THEN
        CLOCKWISE
    ELSE
        CCWISE
```

```
        END
    END
```

3.5.5 Programming Language Evaluation

In this section the CECI programming language is discussed in terms of objective criteria. These criteria were originally described by Hoare [51] (see also [108]). They are:

Simplicity. Overall simplicity of the language from a users point of view.

Security. Reducing the chance of undetected errors in the program.

Fast Translation. The speed of the compiler that implements the language.

Efficient Object Code. Time and space efficiency of the resulting code.

Readability. Ease with which humans can understand the source program.

Simplicity was an important goal in the design of the CECI programming language, so it is not a surprise that many aspects of the language contribute to its simplicity. In particular the decision to use a flat programming structure, a small number of commands, and the use of predefined variables names helped simplify the language, both from the users' point of view and for implementation purposes. The only aspect of the language that meaningfully detracted from its simplicity was the support for adjustable parameters. The use of separate languages for each type of CECI was ambiguous in relation to simplicity; while each language is simpler because of this choice, the aggregate set of all languages a crafter might have to use may be more complex than necessary.

Security was important in the language, but less important than it would have been if CECI programs were larger. Many of the problems detected by compilers for modern languages have to do with the matching and cross referencing of names and

type checking. Since CECI programs are very small, and because there are no subroutines to provide syntactically remote elements to match, there are fewer opportunities for these kinds of errors. The choice of fixed variable names also is a safer choice than automatically declaring new names as is done in many varieties of BASIC, though required declarations with strong typing would be safer. Indeed, the decision to collapse booleans and integers into a single data type had the most serious negative impact on the language's safety, since it allows users to mix the two kinds of values and to create highly implementation dependent code. For example:

```
A = A + (A <= 10)
```

Has no effect if `A` is larger than ten, but has the odd effect of decrementing `A` if it is ten or less.

The speed of translation criteria may seem outdated considering the increase in the processing speed of computers in the nearly thirty years since Hoare published these criteria. However, during that same time the size of a "large" program has also increased dramatically. Anyone who has worked on a truly large program (say a million lines of code or more) is familiar with the problems caused by long compile times and infrequent project code integration. Fortunately, CECIs programs are very short, so while the speed of the compilation process is a concern, it is not a problem. Writing a compiler that executes quickly for small programs is a trivial exercise.

CECI systems do not need to run very fast. The expression of their behaviors operates on a scale close to that of a human, and have dozens of milliseconds in which to affect changes in their operations. However, the computational power available to perform these operations is also very limited.[11] In addition, they have a very limited memory in which to store programs, so space efficiency is important. The use of separate languages for each type of CECI, and the limited set of variable types

[11] The Rototack can execute a maximum of 150 user program instructions per second.

helped keep the code compact and improved execution speed. The only aspect of the language that can seriously slow execution is the interrupt-driven communication system, which can saturate instruction cycles during heavy message traffic (see Section 4.2.3).

One of the affordances listed in Section 3.3 is that CECI programs will be used as a medium of communication between crafters. Readability is critically important when source programs are used this way. The use of parameters, the limited set of commands in the core language, and the use of command and function names drawn from the domain are all factors that improve readability. Conversely, the lack of variable declarations and the fixed variable names reduce readability. The small size of the programs improves overall readability, even as it reduces the cues that might allow the purpose of a variable to be deduced.

In summary the CECI programming language is simple and fairly readable, but suffers somewhat in security. The small size of the programs and the focus provided by separate versions of the language for each type of CECI helps mitigate these security problems and minimize the efficiency and performance issues.

3.6 Programming Tools

One of the basic constraints in programming CECIs is that the objects themselves are not directly programmable because they lack computational power and effective ways to interact with the user. Thus a separate programming platform is required. While this separation causes a number of problems, it is also provides a certain level of freedom in the design of software tools for developing programs. The platform chosen for prototype CECI programming was a commonly available personal computer.[12] This choice provided high-resolution graphics, a keyboard, and pointing device (a mouse) in a relatively inexpensive package.

[12] A slightly outdated Apple Macintosh.

The tools presented in this section support the high-level tasks that the user needs to do in order to find, create, understand, maintain and debug programs. This section describes each of the four tools in the environment and how they are used.

3.6.1 Editor-Animator

The editor-animator is the focus of all tasks involving the detailed manipulation or inspection of a single program. It is implemented as a single window divided into four sections, each serving a different purpose (see Figure 3.3). The sections provide access to the program source code, an animation, the program's meta information, and controls to operate on the program as a whole.

An editing area (center left in Figure 3.3) is where program source code is written, read, and modified. For the most part this area acts like other text-editing tools: it allows direct entry of text at an insertion point, selections of blocks of text, and standard editing operations. The only operation that was important enough to implement in a novel fashion is the way errors are reported. When a program's compilation is triggered, the compiler reports any errors in the editing area by inserting the error message where the error occurs and highlighting the message. In this way the user can see exactly where and what went wrong. The error message can be cleared by hitting the backspace key, which positions the editing cursor as close to the error as the compiler can determine. This approach has been successfully used in Smalltalk programming systems [44, 66]. The only drawback to this technique is the inability to report more than one error at a time. This is of limited importance when working with small pieces of code like a Smalltalk method or a CECI program.

The animator (center right in Figure 3.3) is a lightweight simulation and visualization system, called an "animation", used to display the overall behavior of a program as it executes (see Section 4.4 for more details on the animation of CECI programs). The animation displays the behaviors of the CECI in an abstract format that ig-

nores physical details of size and placement. This is sufficient for the user to get a rough idea of how a program will behave. The animator also provides time-compression and single stepping to assist in understanding programs that run too fast or slow for the user to easily understand.

The meta-information area (top left of Figure 3.3) allows the user to manipulate information about the program as a whole. This is currently limited to selecting the type of CECI and renaming the program.

The final area is a strip along the bottom of the window that contains buttons for actions that are applied to an entire program. These include downloading the program to the connected CECI, saving the program to a file, starting the animation, and doing an explicit compile. There is also a feedback area (similar to those found at the bottom of Web browsers) which displays the status of actions initiated with the buttons.

The editor-animator is used to compile programs, but the act of compilation is largely hidden from the user. While the button labeled “check” can be used to explicitly compile a program, compilation is also done when downloading a program or starting an animation. Implicit compilation, the “check” button’s label, and the way errors are reported are intended to minimize and simplify the interaction between the compiler and the user. The compiler is discussed in more detail in section 4.6.

3.6.2 Help Window

The help window provides the user with information about the language for the CECI currently being used. It is designed to assist users who are unfamiliar with the programming language for the CECI they are working with, or who want help remembering how a feature of the language is used. Thus the tool is designed for both reminder tasks where the user’s memory needs a little help, and for assisting with the understanding of existing code. This tool is not designed to be the sole reference for users attempting to write a CECI program for the first time. These users are expected

to use example programs as a starting point[13] .

The help window consists of two sections. The top portion of the window is a grid containing all the keywords in the programming language. The bottom portion is a read-only text area which displays help text for the selected keyword. An example of this window can be found in figure 3.6.

Both personal experience and user tests suggest that this tool's user interface works well for its intended task. It is uncluttered and displays all of the language keywords at one time, so the user does not have to hunt through a long list or use search tools to find what she is looking for. However, a help tool—no matter how sophisticated or well designed—is useless without appropriate content.

The keyword entries include a short description of each keyword and what it does, the syntactic structure used with the keyword, and at least one example. While only half of the commands shown in the upper portion of figure 3.6 are Rototack specific, the help text and examples for all the keywords refer to the Rototack. This means that the entire help content must be created for each type of CECI, but it also means that the entries will be more useful. While the help text window is read only, examples can be copied out of it and pasted into the editor-animator window, increasing the direct usefulness of the examples.[14]

3.6.3 Behavior Browser

The behavior browser is a tool for finding programs. It is intended for tasks that involve finding existing program code. Unlike most such search systems—which use icons, names, or short descriptions—the browser displays dynamic views of programs. In other words, it shows the behaviors the programs will exhibit rather than static or

[13] The current help system does contain a full example program. So far none of the test users have even looked at it, although they have all used the library of existing programs.

[14] Though I have noticed in teaching various programming languages that students learn the language better if they type in examples. Being able to copy and paste might hinder the learning process.

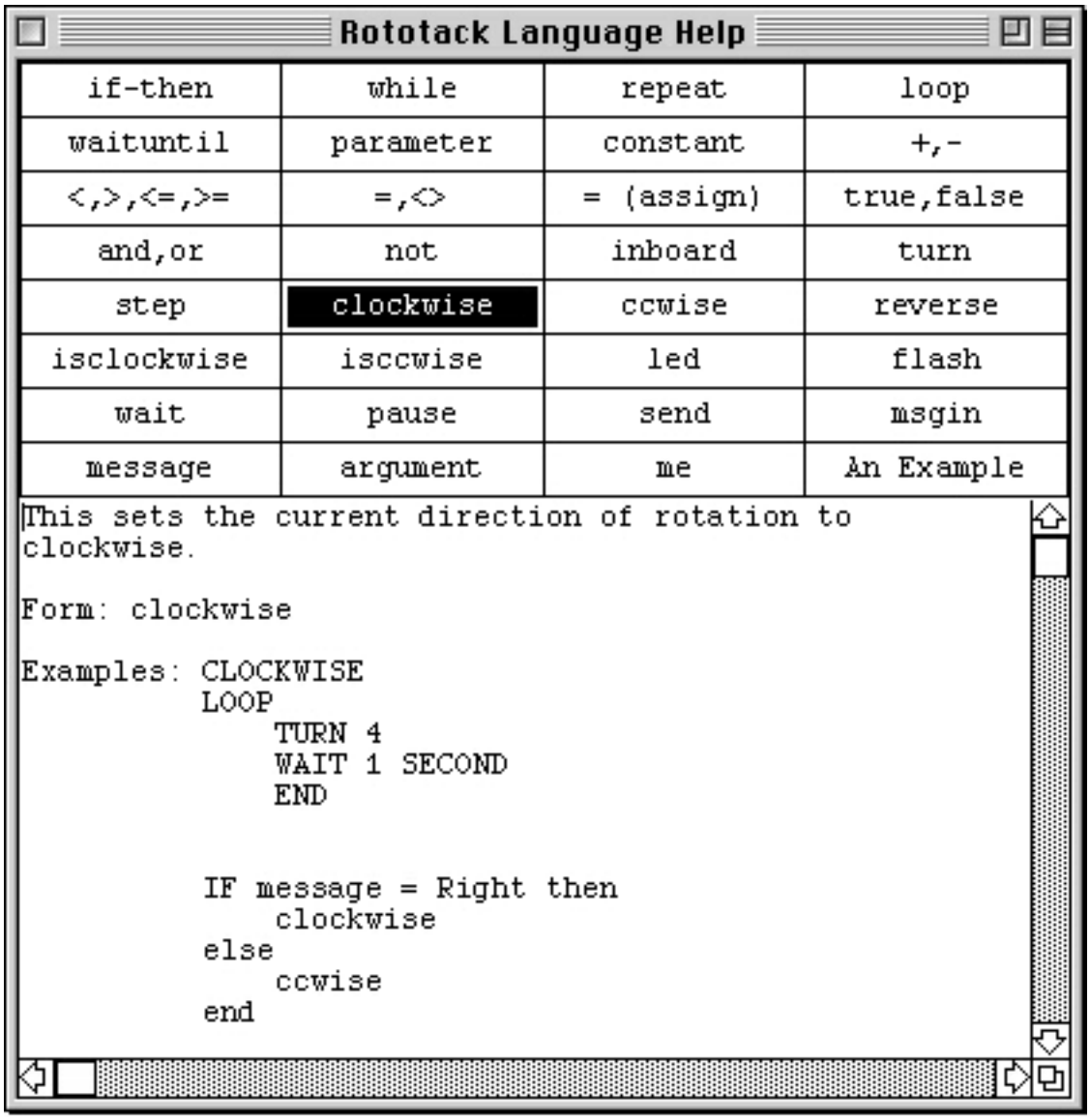


Figure 3.6: The help window for the Rototack programming language. The grid on top shows all of the keywords in the language. Selecting a keyword brings up help text for that keyword in the bottom part of the window.

symbolic information about the programs. Thus it can be used to find a program based on a pattern of actions.

In a sense, the behavior browser takes advantage of the animation's ability to show the user what a program does to avoid more difficult and complex searches via specifications or explanation of examples. The drawbacks to the behavior browser

over these more general approaches are the narrow window it provides into the search space—only five programs can be viewed at one time—and its lack of search capabilities. Systems like EXPLAINER [38, 98, 99] have tackled the more general problem of finding and explaining examples within a domain. Combining a system like EXPLAINER with the CECI programming environment would give crafters a much more powerful way to find and understand example programs.

The browser window is composed of a control strip at the bottom and five small animation spaces called “animation tiles”. The control area is used to scroll through the programs in the library and includes information about the browser’s current position in the full list of programs. Each animation tile shows the actions of a different program from the library.

The small animation tiles are very similar to the larger animation space in the editor-animator window: interactive behaviors can be manipulated and the timer can be clicked to force the countdown to zero. However, the tiles lack the start, stop, and step controls of the larger animator. The name of each program (located at the bottom of the tile) can be clicked, which opens an editor-animator window for that particular program. Figure 3.2 shows an example of the browser in use.

In user tests the browser was heavily used to find programs with specific behaviors. Test subjects used the program names as a first pass filter. Those with promising names were then examined for several seconds before they were either selected or the subject continued to search.

3.6.4 Download Window

The download window controls the communication interface to the CECI. It provides direct access to control commands (see table 4.3) and shows the current state of the communication manager to assist the user with debugging communications problems. There are also controls to adjust the communication parameters to match varia-

tions between CECIs.[15]

The top of the download window displays the state of the communications interface as a line of text. Directly under that are buttons to start and stop the execution of the program. At the bottom of the window are a set of controls to adjust the communications speed to match variations in individual CECIs. The layout of the download window is shown in figure 3.7.

Figure 3.7: The download window. The window includes a status area at the top and buttons to start and stop the program currently on the CECI. The bottom half of the window includes controls to manually manage the communication process, which is insufficiently robust to handle the adjustments automatically.

The communication interface is more complex than it needs to be. If the communication were sufficiently robust—as has been achieved by commercial systems such as modems, PDAs, and the Lego Mindstorms programming system—the communication manager interface would be reduced to the buttons that start and stop program execution on the attached CECI. The only additional function it might implement is

[15] This was intended to be an automatic process, but the automated version was insufficiently robust for the prototype CECIs.

uploading the program that resides on the CECI to the programming system, a capability missing in this prototype.

3.7 Summary

All programming systems are a means for a programmer to communicate intentions and needs to a computer. Because *all* programming systems share this basic structure, it can be used to compare and evaluate different types of programming system and to provide an understanding of programming systems in general. Using this structure to examine programming systems uncovers characteristics grouped into three general areas: those concerning the programmer, the form and media in which the program is expressed, and the characteristics of the target computer.

The characteristics of the programming system developed for CECI are presented informally as a use scenario. It is intended to give a general idea of the programming system developed for CECIs and the environment in which CECI programming may happen. The programming process presented in the scenario and supported by the programming system is one of problem conceptualization, searching through existing solutions, iteratively modifying a solution until it is close to correct, and then refining it to fit the larger physical system of which the CECI is part.

This scenario gives a feel for the environment or situation in which CECI programming takes place. Elements of this environment are the CECI, crafter, craft project the CECI will be part of, and a tool or set of tools for expressing the program and communicating it to the CECI. This environment contains a unique set of affordances that factor heavily into the design of the programming language and tools. These affordances include CECIs' relations to traditional craft items, their small programs, operations on scales comfortable to users, few and concrete primitives, programmers with a wide range of programming experience, programs as a form of communication, and the separation of the programming and development platforms.

The high-level design of the programming system was guided by goals based on these affordances, a preference for simplicity over complexity, and a need to support as much of the crafting community as possible. Other guidance came from examining embedded systems and small control computers, two types of systems that are similar to CECIs in a number of significant ways. Early, important choices were the use of a text-based language, integrating the development tools by task when appropriate, and providing access to program execution via easy program download and simulation of the target CECI.

The language developed with these goals in mind is really a family of related languages, each supporting a different kind of CECI. All of these languages share a common core set of functionality that includes data handling, numeric operations, control structures, and primitives to handle operations common to all CECIs. Because the language is very simple, BASIC—normally considered a poor language for writing meaningful programs—works well as the language's syntactic foundation and has the advantages of both a small vocabulary and a moderately large set of crafters who have been exposed to the language.

The individual tools in the development environment are integrated by sharing information with each other. In addition, the tool the user will spend the most time with—the editor-animator—comprises an editor, compiler, and animator to support writing, understanding, and debugging tasks for a single program. The help system is purposefully kept separate from the other tools and can be ignored by more experienced users. Remaining tools are the behavior browser, which is used to locate existing programs based on their contents, and the download window, which gives the user control over the communications between the programming platform and the attached CECI.

Chapter 4

Technical Solutions

Chapters 2 and 3 describe the two major themes of this dissertation: what computationally enhanced craft items (CECIs) are and how they can be programmed. This chapter focuses on the technical solutions created in the process of exploring these themes. The descriptions of the technical solutions in this chapter include some information on applicable existing work, but a more detailed description can be found in Chapter 5.

This chapter describes a set of technical solutions developed both to create CECIs and to enable them to be programmed. These technical solutions are interdependent, supporting each other and the construction and programming of CECIs as a group rather than as a set of independent tools. They are also minimal, in the sense that removing any would make the task of programming so difficult for a crafter that only the most technically adept could manage it.

The following technical solutions are described in this chapter:

- Type Configuration
- Virtual Machine
- Communication Protocol
- Animation

- Programming Language
- Compiler

The use and form of each of these items is discussed in Chapters 2 and 3. Here they are described as pieces of technology used to solve a problem, including a justification of their use and a description of their design and implementation.

4.1 Type Configuration

Different types of CECIs are... different. While their computation shares a set of common operations, they differ in behaviors, operations, and how the programming language presents those operations to the user. The designer of a new type of CECI must choose and describe these differences in the hardware, software, and programming language. This information is formalized and gathered into a description referred to as a configuration. A listing of the configuration for a Rototack can be found in Appendix C.

The remainder of this section addresses the choice of a configuration to represent this information, defines the nature and form of the information in the configuration, and discusses where the configuration information should be stored. Information on how the configuration is used can be found in the description of the compiler (Section 4.6) and animator (Section 4.4).

4.1.1 Justification

Computationally, all types of CECI have a great deal in common, including arithmetic operations, assignment, flow of control, and timing support. Any variations in the hardware that might obscure these operations are hidden by a virtual machine (see Section 4.2), which implements them as a set of common instructions[1] . Each type of

[1] More specifically as a set of bytecodes that are available on all CECIs.

CECI must also support instructions to control its behaviors. In addition, some types of CECIs will need enhanced versions of common services such as more accurate timing or larger numbers of global variables. Therefore each type of CECI must support two sets of instructions, one that is shared by all types, and the other unique to the type.

The programming system must also support these two types of instructions. The shared instructions are constant, and supporting them is straightforward. The unique instructions must be supported either by creating a separate programming system for each type of CECI, or by having a single system that adapts to new types. While the first approach is appealing, particularly when there are few types of CECI to consider, there are problems when hundreds, or even thousands, of different kinds of CECI are contemplated. One problem is the sheer magnitude of effort[2] required to implement that many different programming environments. Another problem is that at least some crafters are likely to work with several different types of CECI. Separate software for each would be awkward to use and more difficult to learn than a single system. The alternative—implementing a programming system that adapts to the unique instructions needed for a particular CECI—is not a trivial problem, but its drawbacks can be overcome using the formalized description provided by the configuration and a one-time effort in designing a programming system that takes those configurations into account.

4.1.2 Contents of a Configuration

Besides defining the instructions that are unique to a particular type of CECI, the configuration must situate those instructions in terms of the behaviors they manipulate and the user program commands that use those instructions. Figure 4.1 shows an example of the relationships between these pieces of information. The configuration

[2] For example, if each new programming system takes only one programmer-month to create, and there are one thousand different kinds of CECIs, that is better than 120 programmer *years*.

also contains additional information, such as short descriptions of user commands and hints that help the programming environment make the best use of the definitions it contains.

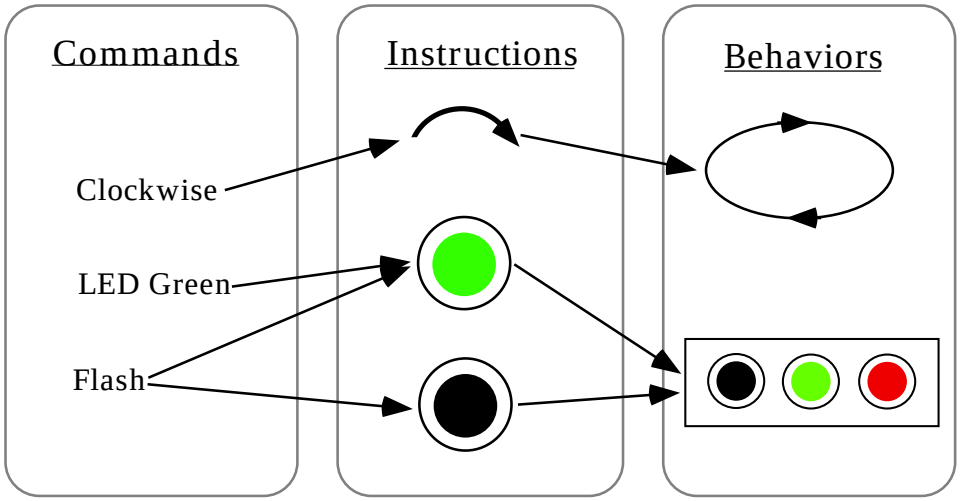


Figure 4.1: Example of the relationships between the commands, instructions, and behaviors in a configuration. The arrows show how commands are defined in terms of instructions, and instructions in terms of behaviors. Not shown is how instructions may be defined in terms of the operations that are common to all CECI.

Each instruction definition in the configuration specifies the bytecode used to represent it in the virtual machine, and a symbolic description of the action it will perform. This symbolic description is defined in terms of the operations on the CECI's behaviors. Additionally, some instructions use operations on the core virtual machine, such as stack manipulation or timing operations.

The following is a definition for an instruction to change the direction of rotation in a Rototack:

```
(instruction right 36 (stepper right))
```

The first `right` is the name of the instruction, `36` is the bytecode, and `(stepper right)` is the symbolic description of the the action. This last sequence can be read as "tell the stepper behavior to do its `right` operation".

The behaviors are defined as references to a type of behavior, which must be implemented in the library of behavior types supported by the programming system, and a set of parameters to match the specific implementation. The following is the definition for the rotation behavior of a Rototack:

```
(behavior stepper rotation
    (stepsize 9)
    (delay 20)
    (startangle 0)
    (startdirection cw)
    (device stepper40))
```

Here the name of the behavior is `stepper`, the values for `stepsize`, `delay`, `startangle`, and `startdirection` are parameters that specialize the rotation behavior for this particular implementation. The `device` parameter is a hint about how the behavior is actually implemented. This is optional information that the animator can use to create better visualizations.

Finally, the configuration defines the pieces of the programming language that use the special purpose instructions. These are defined as command definitions, which may be actual commands like `TURN`, or may be functions that return state information like `ISCLOCKWISE`. The command definitions are used by the compiler to turn user program statements into code for a particular type of CECI. Executable code for a command is defined as a sequence of bytecodes, which can either be those defined in the configuration or bytecodes common to all CECI. The following is the definition of the `CLOCKWISE` command, which sets the direction of rotation in a Rototack:

```
(command clockwise
    (type c0)
    (keyword clockwise)
    (form k)
    (code 36)
    (shortdesc "set direction to clockwise"))
```

Here both the name the command and the language keyword that invokes it

are labeled `clockwise`. The `code` attribute defines the bytecodes generated by the command. The `type`, `keyword`, and `form` attributes define the syntactic structure of the command.

A configuration is simply a sequence of these instruction, behavior, and command definitions as well as a small number of optional hints which help the programming environment make good decisions about details such as animation layout and code optimization.

4.1.3 Storing the Configuration

Since the programming environment—and in particular the compiler—use configuration information to generate programs, a CECI cannot be programmed without the appropriate configuration. Currently, configurations are stored as text files on the programming platform. While this is convenient now, the cost and inconvenience of ensuring that a copy of the file is kept with the CECI makes it inappropriate for large numbers of CECIs being used by many crafters.

In the future these files could be distributed using a standard distribution channel: over the internet, shipped on diskette with the object, or collected onto a CD. Another choice would be to store the configuration on the CECI itself, so it can be downloaded by the programming environment as needed. While the prototype CECIs described in Section 2.2 do not have the capacity to store the configuration, the cost and size of such an enhancement[3] is quite modest.

This last approach eliminates the cost and aggravation of shipping software with each CECI or keeping configurations available for download. It also eliminates the need to move the configuration information between platforms when a crafter starts using a different programming platform. The only apparent drawback, other than a small increase in cost of the CECI, is the inability to update the information if there is

[3] This could be done by adding an inexpensive serial memory chip about the size of a capital ‘O’.

an error.

4.2 Virtual Machine

A virtual machine is an artificial computer architecture implemented in software. Virtual machines can be used to insulate programs from the specifics of the underlying hardware, providing protection from variations in instruction sets, memory architectures, and input/output mechanisms. A program written for one of these virtual machines can execute on any platform on which the virtual machine runs. Thus porting one program—the virtual machine implementation—to a new hardware platform allows any program written for that virtual machine to run on the new platform. This approach is one of the technologies behind Java's well publicized portability [71], although the technique was used long before Java in systems like the UCSD P-machine [20] and others [45, 55].

A virtual machine's ability to protect programs from the details of a hardware platform implies an inability to access special features implemented in those details. Many virtual machines, like the UCSD P-system, simply accepted this limitation as a reasonable tradeoff for complete portability. In these cases the only way to access special features of the hardware is to write a new version of the virtual machine[4] . In contrast, Java has built-in mechanisms to call platform specific code, and it depends on this mechanism to support graphical user interfaces and other platform-specific capabilities. The CECI virtual machines take an approach between these two extremes, implementing a new virtual machine to include special purpose instructions to support different behaviors, while remaining independent of the particular hardware implementations of those behaviors.

This section describes the overall structure of the CECI virtual machine and the

[4] For example, this was done with the UCSD P-system to access the bitmapped graphics of the TERAK computer—a popular platform for teaching Pascal in the early 1980's.

operation of its kernel. The Rototack is used as an example throughout, as it is the prototype CECI with the most complete virtual machine implementation.

4.2.1 Justification

Different CECIs need different levels of processing power, memory, and input/output capacity. For example, the prototype smart tile needs two I/O pins to control its behaviors, a Rototack requires seven, and the programmable hinge needs a relatively large amount of processing power to handle the complexities of the shape memory alloy actuators. Even for a single type of CECI, advances in technology will produce variations in the computation hardware over time[5] . A virtual machine is used to insulate the programming system from these variations in the hardware implementation.

Virtual machines are useful in other ways. Because virtual machine instructions are not limited by the need to be implemented in hardware or microcode, they can be more compact and produce more sophisticated operations than those in a microprocessor. Also, since the cost of developing a virtual machine is dramatically lower than a hardware implementation, they can be designed for specific domains. Thus instructions that support specific domains can be implemented at a low cost[6] . This is a useful way for CECIs to support the domain-specific operations on their behaviors.

4.2.2 Structure of the Virtual Machine

The virtual machine is structured as a kernel which manages the allocation of processor time to other elements, as well as handling communication, instruction interpretation, timing, and the drivers that implement the behaviors. An overview of this architecture can be found in Figure 4.2. New types of CECI are implemented by

[5] The four prototype CECIs with custom computation (the smart tile and the three variations of the Rototack) used three different versions of the same microprocessor family, each with differences in instructions, number of I/O pins, and memory.

[6] Domain-specific instructions are sometimes implemented in hardware despite the cost. An example is the inclusion of specific instructions for multimedia in some of the Intel i86 family of microprocessors [60].

adding drivers that handle the hardware specific to the new type and then implementing instructions that use those drivers. At best this structure is completely modular; existing drivers can be plugged in and "wired" to instructions. At worst—say when a new type of microprocessor is used to implement the computation—the entire virtual machine will need to be rewritten.

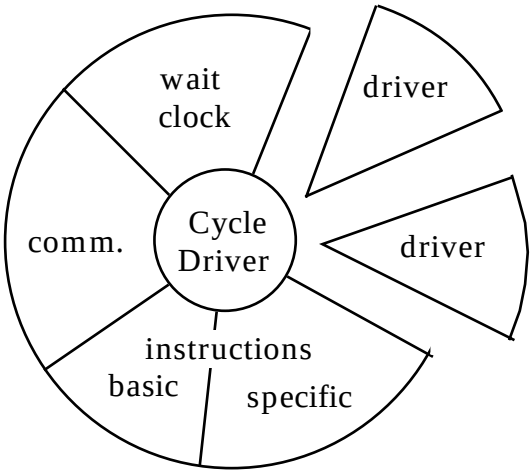


Figure 4.2: Diagram of the overall structure of the virtual machine. The kernel controls what happens and when it happens. The surrounding elements handle specific tasks such as executing basic (core) or CECI-specific instructions, handling communication requests, and timed waits. The drivers run the actuators and sensors that implement behaviors.

As described in Section 4.1, the instruction set is split into two parts. The core instructions are those available on all types of CECIs. These include instructions for flow of control, arithmetic, stack manipulation, communication, data handling, and timing. The other part includes instructions to manipulate the behaviors of that type of CECI. Table 4.1 includes representative instructions from the core and the specific instructions for the Rototack.

4.2.3 Cycles and Timing Support

The kernel of the virtual machine controls the overall behavior by allocating execution time to various activities. The allocation is based on a set of rules that assign

Op Code	Stack Change	Description
ADD	(a b–a+b)	Add the top two elements of the stack.
BTRUE addr	(bool–)	Branch if bool is true.
GT	(a b–bool)	Compare $a > b$ and push boolean on stack.
PARAM1	(–a)	Put value of first parameter on the stack.
WAITS	(n–)	Pause program execution for n seconds.
LEDG	(–)	Change the LED state to GREEN.
TURN	(n–)	Turn the stepper motor n clicks.

Table 4.1: Sample instructions from the virtual machine for the Rototack. The first five instructions are basic instructions available in all CECI, the last two are specific to the Rototack.

priorities, ensuring that critical tasks are given processor resources first. For example, a communication request has a higher priority than does executing a program instruction and will be given control of the processor as soon as it is detected. Table 4.2 lists task priorities for the Rototack.

Task	Priority	Description
CommCheck	idle	Check for a communication event.
LEDFlip	idle	Reverse color of bicolor LED when ‘amber’.
ButtonEvent	idle	Check for a button down event.
CommEvent	1	Handle a communication event.
WaitTick	2	Handle the countdown for a wait instruction.
Turn	3	Manage a step of the stepper motor.
Instruction	99	Execute a program instruction.

Table 4.2: Task priorities for the Rototack virtual machine. Lower numbers denote a higher priority. A priority of “idle” means the task executes once every cycle. Note that for each cycle every idle task executes and, if the cycle has not been saturated, one task with a priority number may also execute.

The kernel breaks processor time into *cycles*. The length of a cycle is fixed for a particular virtual machine, and may vary between virtual machines depending on the processor speed and the time needed by each task. A single cycle should be just long enough to complete the longest driver call or instruction the CECI supports. The Rototack uses a cycle of 128 ticks of the instruction timer, which is about 5 milliseconds. The smart tile uses a 256 tick cycle but, due to a faster processor clock, it finishes the

cycle in about 2 milliseconds.

The fixed execution time of a cycle simplifies tracking the many timing tasks required by computation systems that control physical devices. A single hardware timer is sufficient to synchronize the kernel, and other timing needs can be handled by using idle tasks that count cycles. This assumes a fairly low resolution in timing, a reasonable assumption given that CECIs operate on a scale close to that a human being, as discussed in section 3.3.

A single cycle is assumed to have a certain amount of overhead—sufficient time to handle the looping instructions and execute idle tasks—with the remainder available for larger tasks such as executing a program instruction. Sometimes one of the idle tasks may need to use a larger amount of time than usual. When that happens, it marks the cycle as “saturated”, and no large tasks execute during that cycle. Saturation is also part of the task priority implementation. Tasks are ordered by priority and the first task that needs to take up the cycle marks it as saturated, suppressing the execution of lower priority tasks for that cycle.

4.3 Communication Protocols

CECIs need to communicate, both to allow new programs to be downloaded and to enable cooperation between CECI used in the same craft project. The necessity of cooperation can be easily seen in projects like tile mosaics, where hundreds or even thousands of craft items are used to produce a single finished product.

Standard communication techniques such as USB, IrDA, I2C, and even serial (RS-232 or RS-422) communication either operate too quickly or are too sensitive to variations in timing to be supported by the prototype CECIs system clock. This is likely to be a problem with CECIs in general. Most communication protocols—including those listed above—require an exact timing source and a minimum level of processing power. The cost of implementing these is high when compared to a simple, inexpen-

sive craft item.

This section starts with a detailed description of the communication needs and issues inherent in CECIs in general and the prototype CECIs in particular. It then describes a communication system that fits those needs in terms of data formats, timing, and electrical interface.

4.3.1 Justification

There is no question about the necessity of communication—CECIs need to communicate both with the programming platform and with each other. The question is whether the communication needs and resources of CECIs are sufficiently unique to warrant the creation of a special communication protocol. Looking at just the communication needs the answer is obviously no. Any number of existing low-level communication protocols would handle the modest requirements of CECI communication. RS-232, IrDA and other infrared systems, bus systems like USB or I2C, or wireless communication systems like Bluetooth, WAP, and others would all be more than sufficient in terms of bandwidth and range. [9, 54, 84, 123].

The picture changes when the CECI's limited power, computation, and size are taken into account. Table 4.3 shows some of the problems which arise in using existing communication techniques with CECIs.

System	Media	Problems
Serial (RS-422)	4-wire	CECIs limit on timing accuracy (see text)
Infrared	IR light	Line-of-sight only
I2C	2-wire	Proprietary (Phillips Semiconductors)
USB	2-wire	Too fast for CECI computation
Bluetooth	RF	New technology, still expensive and large
Microcell	RF	Requires considerable computational support

Figure 4.3: Some existing low-level (data transmission) communication systems. The "problem" column describes the most important problem in using that system with CECIs.

Several communication protocols require a minimum level of accuracy—or at least agreement—between the clocks on the communicating devices. RS-232 and RS-422 serial communications, otherwise a workable choice for CECI communication, does a synchronization every byte (which is ten or eleven bits with control information) and requires the two devices to stay reasonably synchronized through the transmission of the entire byte. A disagreement of five percent in the timing sources of the two devices may cause information to be sent incorrectly. The oscillators of the prototype CECIs may differ by as much as twenty percent, and while production CECI units are likely to improve on that, the cost of accurate timing sources is likely to keep the level of improvement below that needed for asynchronous serial communication.

4.3.2 Structure of Communication

While communication is based on the transmission and reception of bits, there must be some structure to the raw data in order for it to be useful. This structure must be rich enough to represent the kinds of interactions CECIs will have with other CECIs and with the programming platform.[7] Table 4.3 summarizes the interactions implemented in the communication system used by the prototype CECIs.

Some of the interactions listed in table 4.3 can be broken down into lower level interactions. In particular, the program memory access (uploading and downloading programs) can be implemented as a set of messages that provide lower-level interactions with program memory. This may prove useful if new ways of interacting with programs—for example, some kind of program patching technique—are developed, as they can be implemented in terms of the lower-level interactions.

Several existing high-level communication mechanisms—including remote procedure calls, event posting, and message passing—could have been used to implement

[7] A special purpose bridge device for connection of CECIs to local networks or the internet has also been considered.

Purpose	Form	Data Size (bytes)
Start / Stop / Reset	PE→CECI	none
Get / Set Parameter	PE↔CECI	1
Download Program	PE→CECI	10-100
Upload Program	CECI→Tool	10-100
Information Request	PE→CECI	0
Return Object ID	CECI→Any	1
Return Memory Size	CECI→PE	1
Signal Behavior	CECI→CECI	0
Request Behavior	CECI→CECI	0
Tick	CECI→CECI	0 to 1
Set Control Value	CECI→CECI	1
Set Data Value	CECI→CECI	1

Table 4.3: Set of high-level interactions that have been implemented in the prototype CECI-to-CECI and programming environment (PE)-to-CECI communication.

the interactions listed in Table 4.3. Of these, message-passing seems like the obvious choice, as its use in large systems of heterogenous processors is well-known (for example, the Actor Model [3, 4] is based on message passing). In practice however, many of the features commonly associated with message-passing, such as automatic process synchronization and general purpose parameter passing mechanisms, were unnecessary. The actual implementation chosen can be described as a remote event posting system.

Messages sent by a CECI consist of three pieces of information: the identification number of the object the message is being sent to, the message identifier, and an optional argument. The object identification number can also be zero, which acts as a broadcast message that is handled by all the CECIs that receive it. Each of the fields is a single integer value, 8 bits each for the prototype CECIs.

4.3.3 Timing and Electrical Interface

The low-level interface of the communication system encodes a message as a sequence of pulses that indicate a break, a one bit, or a zero bit. A break is used to gain

the attention of the receiver, and is very long compared to the the pulses for a one or zero—long enough that most CECIs will be able to catch a break signal by checking just once per cycle. Specifically, a break pulse is 5 milliseconds long while a one and zero are 0.25 and 0.15 milliseconds respectively. The spaces between pulses are between 0.25 and 0.50 milliseconds. A message is composed of a break signal plus the bits used to encode the message. Figure 4.4 shows the voltage and time encoding of an example message.

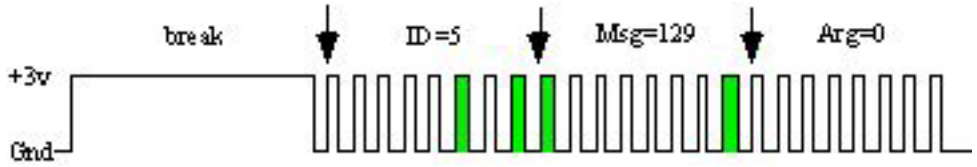


Figure 4.4: Example of the low-level timing of a message signal. This message is to start executing the program on a Rototack with an object ID of five. Arrows mark the beginning of each field. Pulses representing a binary one are shaded for visibility.

Communication is done with three lines: a reference ground, send, and receive.[8] When a CECI is about to send a message, it must first examine the receive line for 0.75 milliseconds to determine if any other devices are using the line before sending a break to announce its own intention to send. This procedure is not completely secure, there is a small chance (less than 1 in 1000) that a collision would not be detected. A production version of the system would need to modify the protocol slightly or allow for resending of messages that are corrupted by collision or other factors.

The voltage variation between a space and a pulse may be considerable, from one and a half to five volts. Currently there are no hard rules about what voltage levels are interpreted as a space or pulse. The prototype CECIs switch from recognizing a space to recognizing a pulse around 0.7 volts above the reference ground.

This wide range of usable voltages, choice of timing, and ability to synchronize

[8] This can—and probably should—be reduced to two lines, a reference ground and a combined send / receive line.

on each bit makes for a very forgiving communication system. The prototype CECIs are able to communicate with each other despite the instability in the clock circuits and different supply voltages. Measurements suggest up to ten CECI should be able to share the same set of wires, though the largest test was with three.

4.3.4 Conduit

One difficulty with a custom communication system is developing a way for the programming environment platform to communicate with CECIs. Either the communication system must be implemented on the programming platform, or an existing communication channel must be translated. The former approach is both platform dependent and essentially impossible on many platforms which do not allow sufficient access to the hardware to implement new low-level communication interfaces. Translation from an existing communication protocol is a more flexible and generally applicable solution. This may explain why most development systems for small control computers use such an approach. For the CECIs a serial (RS-422) interface was translated to CECI communication messages. Figure 4.5 is a photo of the hardware that does this translation.

4.4 Animation

One of the difficulties in a situation where programming is done on a different platform than that on which the program executes is the need to move between platforms during program development. This loss of immediacy makes tasks involving trial execution of the program more difficult. One solution to this problem is to increase the speed and ease of transferring programs using wireless transfer, simple user interfaces, or other techniques. This approach is used for many small control computers like the Cricket and BasicStamp [29, 75]. Another solution is to simulate the execution platform in the development environment, as is done by many chip pro-

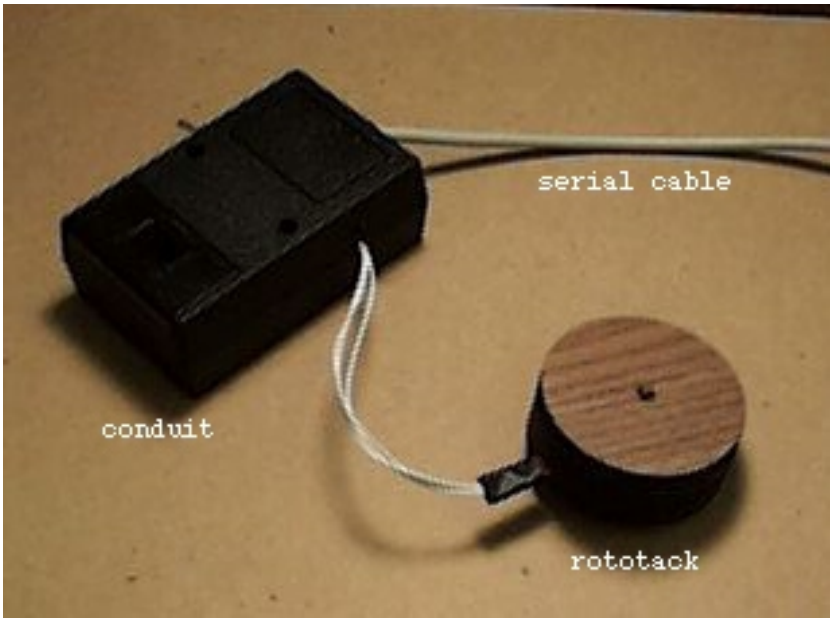


Figure 4.5: The hardware component of the conduit, shown with a Rototack attached. The hardware converts formatted serial data from the programming platform to CECI message protocol signals.

gramming systems [6, 82, 114]. Simulation has other advantages, such as fine control over execution and the opportunity to expose internal state information.

Unfortunately simulation is hard. Accurate simulation of a system that interacts with the outside world requires that the outside world be simulated—either by the user supplying the information about the outside world or by a larger simulation. The first can lead to situations where the user is trying to act as the simulator of a system that is difficult for them to understand, the second to an attempt to simulate the whole world. Fortunately, CECIs have some attributes that make a sort of lightweight simulation—dubbed “animation”—both feasible and useful.

4.4.1 Justification

The separation between programming environment and execution platform causes some problems for the programmer. One is the loss of immediacy in executing the code brought about by communication delays between the two systems and by the

need for the programmer to shift focus from one system to another. A second problem arises when the execution environment has no easy way to expose internal state, as is the case with small, inexpensive systems designed for control rather than interacting with humans[9] . Since the programmer is unable to inspect the program's internal state during execution, understanding, debugging, and testing tasks are substantially more difficult.

A simulation of the target platform running in the programming environment eliminates these problems, at least some of the time. The simulation provides an approximation to the behavior of the program on the actual device that is sufficient for ensuring the program works roughly as intended. However, it cannot completely replace tests on the actual target platform for fine tuning and discovering details of interaction with the overall system of which it is a part. What the simulation can do is display the (simulated) internal state of the program and provide useful functions such as single step execution and time compression for long-running programs.

4.4.2 Simulation is Hard

Unfortunately, simulation has its problems as well. Simulation is hard because it is difficult to define good edges between what should be simulated and what should not. In some cases an edge is obvious. For example, the more sophisticated development tools for microcontrollers (such as Emily52 [28] and MPLAB [82]) use a simulator that mimics everything inside the microcontroller chip, stopping at the chip's I/O pins. Thus a small microcontroller would have only a dozen or so points that need to interact with the outside world, a clear and simple dividing line. It also produces a simulation that is very hard to use.

In practice, microcontrollers are part of larger circuits which in turn are connect

[9] This comes from the relatively high cost of the symbolic display; it does not make much sense to add a factor of ten cost and size to a microcontroller to give it an LCD display.

to sensors and actuators that interact with the world. The developers who create these simulators cannot possibly know what kind of circuits and physical systems the microcontroller is connected to, since microcontrollers are general purpose devices and can be used in too many different ways. Instead the developers do what they can. They simulate everything inside the chip—which will be the same no matter what the application—and provide an interface that lets the users act as the rest of the world. The simulator handles the chip and the user handles everything outside of the chip. This makes some sense, as the user of the simulation should be in a good position to know their own application. In effect, the simulator and the user join forces to produce a simulation of the piece of the real world important to the particular application of the microcontroller.

There are a lot of advantages in having a human act as the "outside world" in such a joint simulation. Humans are flexible, intelligent, and know a lot about the world. However, these advantages do not extend to the ability to sense microsecond variations of voltages on a microcontroller's I/O pins. The user interface for the microcontroller simulation must try to represent a world that is outside human experience, where their knowledge of the world may not apply. While voltage levels can be mapped to something humans can sense directly—such as colors or symbolic representations—the incompatibility in time scales is another matter. Taking every possible shortcut, having a user deal with a microcontroller at this level requires slowing the simulated processor speed to a point where only the tiniest piece of code can be emulated before the user's endurance and patience fail.

The designers of the simulator must choose between dramatically slowing it to interact at a scale the user can handle, or expanding the simulation to cover more of the real world. In other words, they must interact with the user at the user's level or they must remove the user from the simulation loop. The problem with the first approach has already been mentioned; the problem with the second is that it eventually leads to

an attempt to simulate *everything.*

4.4.3 Simulation of CECIs

Microcontroller development systems often include simulators, but there are potentially many more types of CECIs then there are types of microcontrollers. A quick tour through any hobby or craft store will suggest hundreds or thousands of candidates for new kinds of CECI. The time and resources necessary to build individual simulations for hundreds of types of CECI is staggering.

One solution is to build simulations automatically. Sufficient information to automatically build a simulation can be obtained by combining information from the configuration, library of behaviors, and knowledge of the virtual machine's structure. Such a simulation lacks detailed physical information such as physical size, how behaviors are implemented, and physical interactions between behaviors. Without these physical details the simulation is restricted to displaying abstract schematic representations of an item's actions. The CECI development environment uses these simplified, automatically constructed simulations. Since they are abstract simulations combined with a visualization they have been dubbed *animations.*

The programming environment provides animations as a way to help the user understand what a program will do, and to search for particular kinds of behaviors.

4.4.4 Animation Architecture

As mentioned in the previous section, animations are generated automatically based on information about the configuration of a CECI and its behaviors. All of this information is available when the CECI is designed, and the animation could be generated at that time. An advantage to this approach is that the designer has an opportunity to improve the animation before it is distributed to crafters. The disadvantages are that the simulation and any updates or enhancements must be distributed to crafters

using that CECI, and the need to port the simulation to different programming environments. This is the same set of problems the configuration was created to handle (see Section 4.1), and it is the configuration that provides a solution.

The CECI development environment generates animations “on the fly” based on the information in a configuration. The animation combines its knowledge of the virtual machine with the instruction and behavior information in a configuration to create the simulation of the computational part of the CECI. There may also be hints embedded in the configuration to help specify layout and presentation.

The configuration information is necessary for the animation to work, but it is not sufficient. The programming environment must contain information about each of the behaviors in the configuration. These are stored as classes (in the object-oriented programming sense) which include information about the forms in which the behavior can be displayed, how to handle variations in the behavior, and the operations that can be performed on the behavior. This approach is similar to that taken by Aronson and Bose [8] who build simulations based on a model and a set of simulation components. The major points of difference are that Aronson and Bose’s method is more general, while animations produced by the CECI programming system include integrated visualization.

The visualization components are integrated with the simulation by providing two classes for each behavior, one to simulates its state and operations, and another to visualize that state. Behaviors are fairly general, and a small set—on the order of a hundred—should be sufficient to implement a broad range of CECIs, so the task of implementing the library should not be too onerous.

To illustrate, consider the rotation behavior of a Rototack. It is represented as an object which contains internal state to specify how the rotation works: step size, delay time between steps, and if the rotation is bidirectional. It also contains the information for an animation: the current position and direction. What it does not contain is infor-

mation on how it should present the simulated state to the user. This is handled by a different object. More detailed information can be found in Table 4.4 which lists state information for each of a Rototack's behaviors.

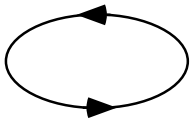
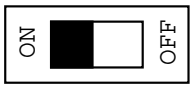
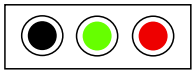
Behavior	State		Operations
	Configuration	Simulation	
Rotation 	stepSize stepSpeed isBidirectional initPosition initDirection	position direction	Turn Clockwise CCwise
State Switch 	initState	isOn	IsOnNow
Color Light 	colorList initColor	color	SetColor On Off

Table 4.4: State information for the behaviors in a Rototack animation. The state of each behavior includes information about the item's configuration, which is taken from the configuration file, and state to handle the animation. The operations are the animation-level actions that can be applied to that behavior.

Coordination of the behavior with the simulation of the CECI program execution is handled by a simulation controller. This object knows how to construct and integrate the set of behavior objects for a particular CECI and how to simulate the execution of a compiled CECI program for that object. A similar object takes care of managing the visualization of the behaviors. The execution architecture is similar to the Model-View-Controller (MVC) architecture of Smalltalk-80 [44] with two controllers, one handling the simulation and one handling the visualization. Figure 4.6 presents a somewhat simplified view of this structure for a Rototack.

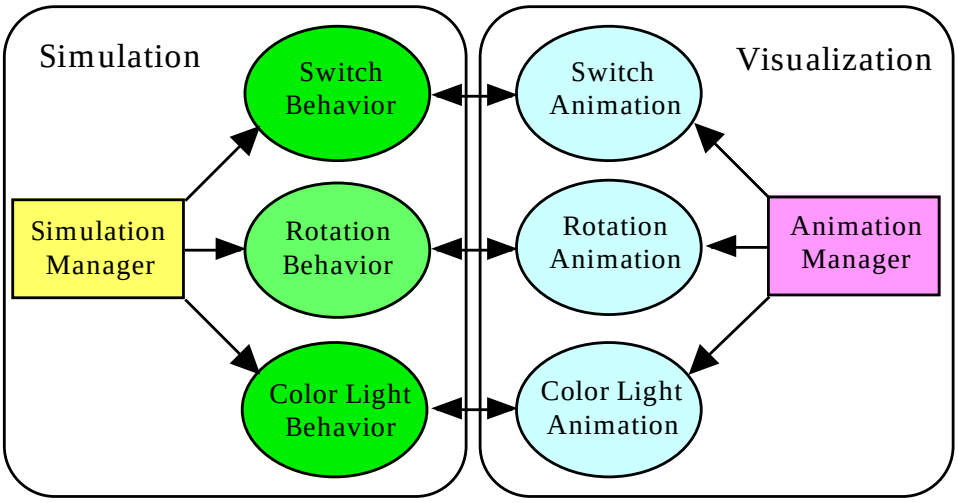


Figure 4.6: Object structure for the animation of a Rototack. The objects are separated by their function of simulation or visualization for the Rototack animation. Arrows show references between objects. Ovals are objects whose existence and structure depend on the type of CECI being animated.

4.5 Programming Language

It is not immediately obvious that a CECI needs a programming language. CECIs are concrete, simple, and require relatively small programs and so seem an excellent candidate for one of the handful of non-language programming systems that have been developed in the last thirty years. However, programming systems using languages have a number of advantages over their non-language counterparts, not the least of which is that a human-readable and editable program is created during programming. These advantages are sufficient to overcome any usability benefits of these alternative programming approaches.

The goals that drove the design of the CECI programming language are that it should be easy to use, allow the full capabilities of a CECI to be expressed, handle all the different types of CECI, and be easy to implement. This goals are somewhat contradictory. The design of the language is based on tradeoffs made in an attempt to satisfy these goals as completely as possible. It is by no means the only set of choices

that could be made, or even an optimal set, but it does strike an acceptable balance between the goals.

This section is not a detailed specification of the programming language, which can be found in Appendix A. Instead it focuses on what makes the CECI programming language unique.

4.5.1 Justification

Not all programming systems require the creation of a source program. The best known of these is programming by example [68, 70, 79], which uses one or more examples or demonstrations to produce an executable program. This and other programming systems, such as tactile programming or natural language processing [72, 113, 118], produce programs in a form that is difficult to reproduce or is dependent on context. CECI programming is an interesting domain for these programming systems, as its small programs, limited and concrete domain, and small number of primitives match well with their limitations and strengths. But are these systems appropriate for programming CECIs?

There are advantages to programming systems that require a source program. Source programs are important because they serve as a source of reflection, understanding, and as a medium for modification. These activities are at the root of program reuse, refinement, and debugging. They are also useful as objects of study in the process of learning to build new programs.

Source programs are intended to be a means for the programmer to communicate with a computer, but also serve as a way for programmers to communicate with each other. Without this means of communication, programmers would be forced to rely on descriptions of effects and informal explanations of the construction of the program. While a formal language could be developed to describe the construction of programs using non-language programming systems (see [87] for some examples of

formal languages in everyday activities), that would eliminate many of the advantages of using such a system[10] .

When considering their use for communication between programmers, text-based notations have some important advantages over graphical notations. While graphical languages are very effective for particular tasks (see [15, 116] and [87, pp 61–66]) their need for a larger display space and use of less familiar and potentially complex graphical elements make them more difficult to reproduce in media such as print, E-mail, and handwritten notes[11] .

4.5.2 Many Languages

Each type of CECI has a different set of behaviors, and even those which share some behaviors may choose to use different sets of operations to control them. Thus each type of CECI must support a somewhat different set of primitives to control those behaviors. Section 4.1 describes how the different instructions and user commands that use those instructions are defined in a configuration. These differences suggest that each type of CECI will require a somewhat different programming language. For example, the Rototack language contains commands like `TURN` and `CLOCKWISE` to handle its rotation behavior, while the smart tile language contains commands like `COLOR-ON` and `HASCHANGED` to handle its color change behavior.

While each type of CECI has a somewhat different language, many commands and constructs can be the same because they express general notions or support capabilities common to all types of CECI. For example, a general construct such as an infinite loop is always described with a `LOOP` and matching `END`, while a common operation like communicating to another CECI always uses the `SEND` command. The

[10] There may be some advantage to a separate formal language to talk about programs. Since it would be specifically developed for communication between humans it would be likely to fit human modes of communication better than a language developed for communication between a human and a computer.

[11] Although cocktail napkins are kind of a toss-up.

programming language for any given type of CECI is made up of this shared core language that is always the same, and a language extension to handle the behaviors particular to that type. It is more accurate to think of the CECI programming language as a family of closely related languages than a single language.

The core language includes control structures, directives, data handling primitives, commands, and functions while the extensions are limited to commands and functions. The commands are used to make things happen, while functions are used to examine the state of a behavior.

4.5.3 What the Language Is Not

CECIs are simple devices. They do not have file systems, graphical user interfaces, keyboards, mice, or even small character displays. The programming language does not need to support these devices and capabilities, and so can exclude the data structures, primitive operations, and libraries needed to deal with them. In addition the programs are short, rarely more than two dozen lines, and are built from a small number of concrete primitives (see Section 3.3). So language constructs intended to deal with large, complex programs or large vocabularies are unnecessary.

With no direct user interaction there is no need for character or string data in a CECI; indeed, there is only a limited need for data of any kind. CECIs are more about physical action than they are about information. Data is limited to tracking the state of a sequence of actions and counting. The language implements this minimal level of data support with a single integer data type. Specific behaviors can implement new data types when needed, as might be the case for behaviors intended for human-computer interaction, such as control of a small LCD display.

The small programs and minimal vocabulary of CECI languages led to the elimination of any type of modularity construct. In particular, subroutines are not included in the language. The idea is that a program of no more than twenty lines does not need

subroutines, packages, modules, classes, or any of the other constructs developed to break programs into pieces. This particular change was sufficiently controversial that it prompted a reexamination of ways that language features can be removed.

Disabled features—programming language features that are implemented and then disabled—were the result of this reexamination. This was a way to “hedge bets”, to exclude seemingly unnecessary features while allowing them to be quickly enabled if their removal caused problems. The advantage of this approach (over not including them at all) is that the features are included in the language design, ensuring that adding them does not require a complete redesign.

Interestingly, none of the disabled features was ever enabled. The nature of CECI programming, at least as practiced on the prototype CECIs, made them unnecessary. A complete list of disabled features can be found at the end of Appendix A.

4.5.4 Support for the Core Language

The core programming language—the part that does not change between different types of CECI—supports data handling, control structures, and capabilities shared by all CECIs. In the prototypes, this was limited to basic arithmetic, comparison operators, boolean operators, peer communications, and random number generation. In addition, all of the disabled features are part of the core language.

Most of the supported operations are similar to those in other languages. The data handling is somewhat different, in that it is limited to one type, and the lack of subroutines limits programs to a single global scope. The communication system is also somewhat different, as it lacks the stream abstractions common in other programming languages. Instead, communication between CECIs is based on message passing to specifically identified objects or as broadcasts to all connected objects. (More details on the communication system can be found in Section 4.3). More specifically, the language support for communications is limited to sending messages, testing if messages

have been received (polling), and examining the structure of the most recently received message. Appendix B lists some programs which use communication commands and functions.

The syntax of the core language is defined in Appendix A, but there is no formal semantic definition. Table 4.5 gives some indication of the semantic effects of each of several kinds of language structures. These effects are guaranteed for the core language, and normally apply to the extensions as well. The table includes information on the stack, control flow, global variables, state information in drivers or the communication system, and the production of physical actions.

Operation Type	Effects				
	stack	control	global	state	actions
Declaration	–	–	–	–	–
Control Structure	–	yes	–	–	–
Assignment	yes	–	yes	–	–
Expression	yes	–	–	may	–
Function	yes	–	–	may	–
Commands	–	–	–	yes	yes

Table 4.5: The effect of various types of programming language operations. A dash (–) means no effect. A “yes” indicates an area always affected, and a “may” indicates areas that may be affected.

4.5.5 Meeting Goals

At the beginning of this section, four goals for the CECI programming language were listed. These were that it should be easy to use, allow full capabilities to be expressed, handle all types of CECI, and be easy to implement. In the final implementation, these somewhat conflicting goals were not all given the same weight. Ease of use and the ability to handle all types of CECI were given priority over ease of implementation and full expression of capabilities. Ease of use is a complex issue, but the language’s small vocabulary and uncomplicated syntax are intended to reduce learning time and simplify use for the occasional programmer. The ability to handle all kinds of

CECIs was the reason behind splitting the language into a core and an extension which varies for the different types.

The acid test for showing that the CECI language can express the full range of capabilities of each underlying computational system is in its ability to express any program in the language that could be expressed in the underlying machine code, save that the program generated by the programming language can be larger. The language, as it is currently implemented, does not pass this test. It *is* able to pass, however, if two of the disabled language features—subroutines and static data arrays—are enabled.

Ease of implementation also suffers, though the language is not especially difficult to implement. The top-down structure and use of syntax to disambiguate operator precedence and the dangling `ELSE` problem lead to a straightforward parser implementation. Unfortunately, the need to extend the language for each type of CECI greatly complicates matters. More details can be found in Section 4.6 which discusses the compiler implementation.

4.6 Compiler

Compiler construction is well understood. Standard references and texts[12] describe it more completely than can be done here. Rather than restating standard techniques, this section focuses on the adaptation of the CECI compiler to different types using the configuration and the way the user interacts with the compiler.

4.6.1 Justification

Given a programming language and the virtual machine that is its target (see Sections 4.2 and 4.5), some kind of compiler is necessary to translate from the source language to the virtual machine's bytecodes. The important question is if the compiler needs to be different in some important way from compilers for similar languages. The

[12] I use the classic "dragon book" [5] for reference and Scott [107] as a text.

answer to this question is “yes” for two reasons. First, the compiler is dramatically simpler than most since it does not have to deal with multiple types, nested scopes, or large numbers of keywords. Second, its design is complicated by the need to adapt itself to new language commands, functions, and target instructions for the different types of CECI.

4.6.2 Specialization by Configuration

Each type of CECI uses the core language and its own set of commands and functions to manage its specific set of behaviors. Each extended command or function is defined in the configuration, and includes syntactic information, human readable command descriptions, and information on generating the compiled code. The following is the information from the Rototack configuration for the `STEP` command, which turns the Rototack one click (nine degrees) in the current direction.

```
(command step
    (type c0)
    (keyword step)
    (form k)
    (code 129 39)
    (shortdesc "turn a single click"))
```

Here, the `step` is both the name of the command and the keyword used to recognize it. Its type is `c0` which is a command with zero arguments. The syntactic form is `k`, which means it consists of a single keyword. The bytecodes generated by this command are given as a sequence of bytes, but could include instruction names. More examples can be found in Appendix C.

For each new language, a compiler is constructed by combining the basic compiler—which understands the core language—with a set of command compilers, each of which understands how to compile a single function or command from the extended set in the configuration. The basic compiler then passes unrecognized commands to

these command compilers.

This structure is very modular but too simple to handle the more advanced enhancements needed to support the operations on some behaviors. For example, a behavior for a character-based LCD display may want to implement a new type of variable to store strings. This requires the command compilers to interact more intimately with the basic compiler, in this case to allow access to the symbol table and variable resolution during code generation. This level of interaction is possible, but must be implemented by the behaviors stored as classes in the programming environment and used by configurations that include those behaviors.

4.6.3 User's Interaction with Compilers

In general, compilers interact with users in a very simple way. They are given a source program and a place to put the compiled code they generate. Then they either produce compiled code or one or more error messages. At best, compilers are tools that efficiently translate source programs into a form usable by the execution platform. At worst, compilers are more like small children who hate doing chores, spending more time telling you why they cannot compile your program than actually compiling it. This "worst case" is the more common view for novice users, who are more likely to make errors, and less likely to understand why the compiler is complaining.

In the CECI programming environment (see Section 3.6), these error messages are the only compiler output users ever see—compiled programs are stored internally for animation and download, but are not written to a file. Since the user's interaction with the compiler is so simple, and not of much benefit to the user, it is kept to a minimum. Figure 4.7 shows the users interface to the compiler.

The compiler can be explicitly activated using a button labeled "check". It can also be implicitly activated by an action that requires compiled code, such as downloading a program or starting an animation. Error messages are reported directly in the

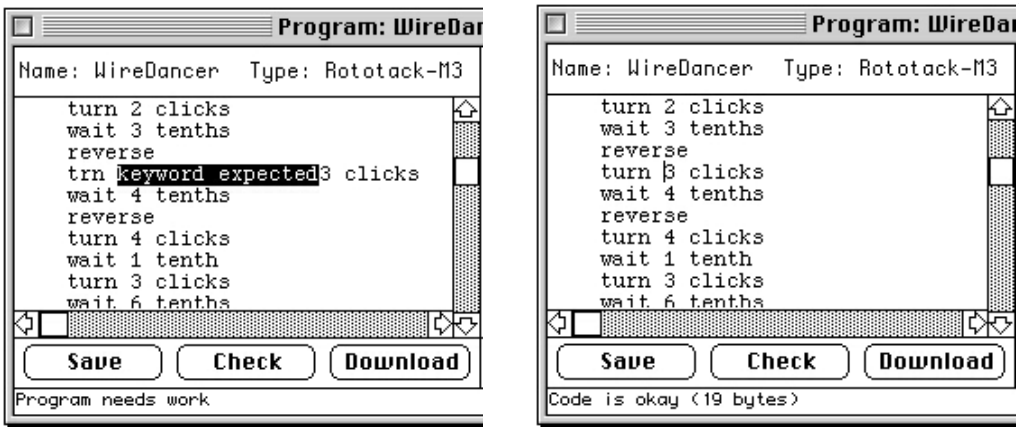


Figure 4.7: Two views of a window fragment showing the user interface to the compiler. The left hand view shows an error being reported as highlighted text in the editor. The right view shows a compilation that succeeded. Note the information in the small feedback area at the bottom of each figure. The check button explicitly activates the compiler.

editor window using a technique developed for Smalltalk [44] where a highlighted error message is inserted in the editor window at the point the error occurs. The user can easily see the error and can get rid of the message by hitting backspace. The drawback of this technique is that only one error can be reported at a time. While single-error reporting is unworkable for large pieces of code, it works well for the small methods in Smalltalk, and for the small programs written for CECIs. The implicit triggering of the compiler, the way errors are reported, and even the label for the button that triggers compilation are all intended to make the compiler as invisible as possible.

One area that was discussed but not implemented during this study was having the compiler provide feedback about details of CECI hardware. For example if CECIs were common enough that many variations of the Rototack existed, it is quite possible a user would try to write code that would work well on one variety of Rototack, but not on the variety the crafter was using. Informative messages about alternatives would be very helpful, particularly for novice crafters who may not have a deep understanding of their materials.

4.7 Summary

This chapter describes a set of technical solutions developed to create CECIs and to enable them to be programmed. These technical solutions are interdependent, supporting each other and the construction and programming of CECIs as a group rather than a set of independent tools. This set can be considered minimal, in that removing any would make the task of programming a CECI so difficult for a crafter that only the most technically adept could manage.

CECIs can be very different from each other. Each type is different in its physical form, the behaviors it supports, and the devices that implement those behaviors. These differences must be reflected in the programming systems as variations in the programming language, how executable programs are generated, available help text, and CECI animations. A configuration is a piece of information that specifies how a particular type of CECI is different. This information is then used by the programming system to adapt itself to the CECI being programmed.

Differences in CECIs also affect the type of computational support they need. Some will need little processing power, some will need more, and still others have size or power requirements that outweigh issues of cost or processing capability. These factors will influence the choice of the hardware that implements the computation, causing variations in instruction sets, memory, and I/O capabilities. A virtual machine is used to insulate the programming system from these variations and to provide an instruction set closer to the domain-level operations on the item.

The needs and capabilities of CECIs are not a good fit for existing communication protocols. In particular the relatively slow processing speed and low timing precision of CECIs preclude most existing communication systems. Since CECIs must communicate—both with the programming environment and with each other—a new protocol that fits their limitations and capabilities was developed. The result is a simple

message-based communication system whose low-level timing and electrical specifications are loose enough to match the varying timing and power resources of CECIs.

CECIs are incapable of acting as their own programming platforms. A difficulty with any system that has this separation is the loss of immediacy caused by switching between the programming and execution platforms. One way to reduce the impact of separation is to simulate the execution platform on the programming platform, a solution often used by microcontroller development systems. But simulation is hard, and tends to lead to either burdening the user with an unreasonable amount of the simulation work or to an attempt to simulate the whole world. CECIs are able to sidestep many of the problems with simulation by taking advantage of their human-like scale of operation and accepting a simplified simulator. This lightweight simulation combined with visualization capabilities has been dubbed an “animation”. Its primary purpose is to display an abstracted version of the actions a program would manifest when run on a CECI.

CECI programs are described using a programming language. This language is simple in some ways because CECIs are able to do without features that are considered necessary in other languages. Among these are support for code modularity and complex data types. In other ways the language is complicated by its need to support different sets of primitive operations for each type of CECI. The resulting language has a simple set of core commands and structures, which are then enhanced with an additional set specific to each type of CECI. Because these extensions integrate seamlessly with the core, the CECI programming language is actually a family of closely related languages, one for each type of CECI.

The compiler that implements the language differs from other small-language compilers in its need to handle a somewhat different version of the language for different types of CECIs, and in its attempt to minimize its impact on users. The first is done by splitting the parser into a part that handles the core language and a part that han-

dles the commands specified by a configuration. The latter is done by reporting errors directly in the editor window and automatically invoking the compiler as needed. The overall effect is to give the impression that the source code is the true program, with the system occasionally complaining about errors when an animation or download is invoked.

Chapter 5

Related Work

This thesis touches on many diverse areas. This chapter focuses on those areas that have a strong impact on this work: end user programming systems, computation as building material, and computation for craft. Other areas related to this work are ubiquitous computing [2, 115], wearable computing [67, 103], tangible user interfaces [62, 113], and microelectromechanical systems (MEMS) [11, 27, 37].

5.1 End User Programming

Considerable effort in the end-user programming field goes into alternative forms of programming languages. The idea is that programming in traditional text-based languages is hard [109], and so other forms of programming languages will be easier and more natural. Nardi provides an excellent, though somewhat skeptical overview in [87]. A more positive overview can be found in [65].

One form of programming where no explicit source program is generated is in programming by example (PBE), which infers executable programs from examples or demonstrations given by the user [24]. These range from fairly general-purpose programming systems like Garnet [79, 86] to special purpose “languages” embedded in applications such as Geometer’s Sketchpad [63] with systems such as Tinker, Metamouse and Edger between these extremes [22, 23, 70, 76, 77].

Visual languages are another form of programming for end users which do pro-

duce explicit program representations, but uses graphical rather than textual elements to do so [15, 47]. Proponents of visual programming languages claim they are more natural, though many limit that claim to certain problem domains while others are openly sceptical [15, 46, 47, 87, 116]. There are also hybrid languages that combine visual and text based elements in an attempt to take advantages of the strengths of each [97, 100, 110].

Another technique that can be used to implement end-user programming is the generation of programs based on natural-language descriptions or specifications. [14, 72]. While natural language user interface systems have been successful in limited domains such as database query, it has not had notable success as a basis for general purpose programming systems. An interesting approach that may have applications to programming CECI is combining natural language processing with more formal specifications or programming languages [95, 39, 40].

Of more interest to this study are investigations of end-user programming that focus on the domain and the user, rather than the language. Bruckman's work with users writing programs for MUDs [13] and Nardi's work on examining how users work with existing end-user programming systems [87, 88] are examples of studies that focus on the user and social setting of end-user programming. Eisenberg has looked at the advantages of adding end-user programmability to well focused applications [31] and he and others have looked at ways to help users learn to write programs within these programmable applications [26, 31].

5.2 Computation as Material

This thesis has focused on building materials that blend computation and physical objects. However, the current trend is to provide the computation separately. To add computation to physical structures like craft projects, the computation must be treated as a kind of special-purpose building material. Small control computers and

microcontrollers are the commonly available computational “materials”.

The microcontroller market is huge and dominated by professional engineers. However, several families of microcontrollers, such as Microchip’s PICs [58] are simple, inexpensive, and robust enough to have become popular in the hobbyist market. These chips are supported by a wide variety of products aimed at hobbyists and novice users [10, 56, 61]. A kind of easy-to-use “computational material” that has been available for some time is a variation of the Z8 [122]. While technically a microcontroller, it required very little additional hardware for programming and uses a BASIC variant as a programming language rather than assembly language.[1]

Modern small control computers have most of the advantages of the Z8 but are easier to connect to common sensors and actuators. The MIT Media lab has created a series of small control computers for education, including the Programmable Brick and the Cricket [74, 75, 101]. Martin, a principle designer of both the Programmable Brick and the Cricket, also developed the Handyboard [73], a control computer often used in educational robotics. Commercially, there are several small control computers designed for the hobbyist market, such as the BasicStamp and its derivatives [93]. There is also a commercial version of the Media Lab’s Programmable Brick: the Lego RCX. The RCX, Programmable Brick, and Cricket are all designed to be easy to use. This ease of use includes both an easy-to-use programming system and a set of standard sensors and actuators.

5.3 Computation as Craft Tool

Currently, the computation designed to support hobbyists is almost exclusively limited to software for designing craft projects. Examples are software to design model railroads layouts [96, 112] and tile mosaics [111]. An interesting step beyond simple

[1] This variation of the Z8 is still available and fairly popular. Its main advantage over higher performance microcontrollers and the easier-to-use small control computers is very low cost.

design is software like Hypergami [30, 34] which lets the user design geometric paper shapes and then print out the folding net for that shape. The net can then be cut out and folded to produce a physical representation of the shape. Hyperspider [19, 35] is a program for designing three-dimensional string art. Hyperspider does not come as close to producing the final object as Hypergami, but is able to print guides to assist users in threading the strings—very important for even moderately complex string sculptures. The Craft Technology Group at the University of Colorado has started looking into craft design software that makes use of other kinds of output devices such as laser cutters and three-dimensional "printing" systems [12]. These systems promise to shift software for craft from strictly design to a more integrated design and construction approach.

5.4 Summary

The work presented in this thesis draws from each of the three areas discussed in this chapter. Programming of computationally enhanced craft items (CECI) is domain-specific, end-user programming. The computation in the CECIs is a use of "computational materials", with the early prototypes taking advantage of small control computers while later prototypes use a microcontroller. The programming system is design software that assists crafters in the design and construction of programs that control the behavior of CECIs. This work combines these elements in a (hopefully) coherent and unique way to demonstrate that crafters can be provided with usable materials that blend traditional craft materials and computation.

Chapter 6

Conclusions and Future Work

The key findings in this research involve the design and construction of physical objects, computation, and tools that are needed to make it possible for crafters to program computationally enhanced craft items (CECIs). The minimal set of technical solutions needed to answer the research question is discussed in Chapter 4 and includes a configuration system, virtual machine, communication protocol, program animation, programming languages, and a compiler architecture. Other findings from this research work, including the results of user testing, the core contributions of this research, and future research directions, are discussed in this chapter.

6.1 User Tests

There are many questions about this work, and in particular about the usability and appropriateness of the programming system, that can only be answered by tests with crafters. A user test was designed to get preliminary answers to the following questions:

- Can the programming system be used by non-programmers?
- What parts of the programming system are well structured and useful?
- What parts of the programming system user interface can be improved in the next iteration?

6.1.1 Method

Subjects were recruited by posting signs in campus buildings not associated with science or engineering disciplines.

Tests began with a short interview. Subjects were asked a set of questions about their previous experience with programming, their range of experience with computer applications, and any crafts they might be involved in. This was followed by a short (five to six minute) verbal introduction to the Rototack and the programming software. This standardized talk included a description of each tool in the software environment and its purpose, but no information on how it was used. The terms program and programming were avoided in the talk. While the talk was not recorded, it did follow a specific outline, and some effort was made to cover the same material for each subject. After the talk, subjects were provided with a computer with the software already running, and a rototack already turned on and plugged into the programming interface. A written list of eight tasks were provided and users were asked to try to complete all the tasks without additional help. The following is an abbreviated version of the task list given to the subjects:

(1) Find a rototack program that rotates back and forth (clockwise and then counterclockwise).

(2) Look at the program and change it so that it waits for ten seconds at the end of each back and forth motion.

(3) Now modify the same program so each back and forth motion is exactly one half turn.

(4) Make sure the modified program is saved.

(5) Convince yourself that the program will work.

(6) Download the modified program to the attached rototack.

(7) Start running the program on the rototack.

(8) Find a rototack program named “unsure”. Use the software to figure out what it does and write a short description of its behavior.

After working on the tasks, subjects were asked if they had any questions or comments and given an opportunity to use the software further.

6.1.2 Results

Six subjects responded to the postings and participated in the test. Two reported having a small amount of previous programming experience and another reported extensive experience with spreadsheets. The other three subjects claimed some computer use experience but said they had not worked with spreadsheets, written macros, or had any experience they thought of as programming. All subjects reported having experience with some craft. The crafts they reported being involved in in a meaningful way were cross-stitch, knitting, pottery, quilting, painting, wood working, sand art, and stained glass. One subject also reported being involved in “kid’s crafts”, which they further explained as “school arts and crafts”.

Every subject completed tasks 1 through 5 and 8. Tasks 6 and 7 were impossible in some tests because of difficulties with the communications hardware and software. Two subjects completed tasks 6 and 7 with no difficulty and one complete tasks 6 and 7 despite some difficulty with the communications hardware.

Excluding tasks 6 and 7, subjects took from 11 to 45 minutes to complete the tasks, with an average of 20.5 minutes. The two subjects who had no difficulty with the communications hardware or software took two and four minutes to complete tasks 6 and 7. The subject who completed tasks 6 and 7 despite problems with the communications hardware completed tasks 6 and 7 in 19 minutes. Including all tasks completed

subjects took from 13 to 45 minutes to complete the tasks with an average of 24.5 minutes.

The subjects had available all of the tools described in Section 3.6: the behavior browser, editor-animator, help window, and download tool. When the subjects started, only the main menu was open, showing options to open an existing program, create a new program, open a behavior browser, and open the help window. All subjects used the behavior browser and editor-animator tools, and four attempted to use the download tool. Four subjects opened the help window, but only one selected a keyword and read the associated help text.

The use of the behavior browser was very consistent. All subjects used it for the first task and all but one subject went back to it for other tasks. All subjects showed the same usage pattern. First, they looked closely and scanned the bottom of the animation tiles where the names were written (in a rather small font). Then they either used the scrolling arrows to go to the next page (where the scanning behavior was repeated) or they spent several seconds looking at one or two of the animations. During post-experiment discussion, two subjects mentioned that looking at the names was easier. One of those two said that they liked the animation because it was “nice to see what it really did”.

Use of the editor-animator was not as uniform. All subjects edited source code and all started animations using the start button, though only four used the stop button to end the animation. There were no obvious usage patterns, though all but one subject tended to frequent switching between the editing and animating the program. Interestingly, there was only a single syntax error in all six tests and that subject noticed and corrected the error after looking at the screen for about twenty seconds.

The download window seemed to be confusing to the subjects. One subject did not use it at all after looking at it for some time. Another subject spent considerable time with it and was eventually able to get the program downloaded despite problems

with the communication hardware and software. That subject made several comments while using it suggesting that she was unhappy with the way it worked. Most of these comments were specific to particular buttons and feedback messages, though the subject said that the ”thing was hard to use” and ”I can’t figure this out, it’s confusing me”.

In the post-experiment session, one subject who had reported no previous programming experience wrote a new program for the rototack from scratch. That program is included in Appendix B.

6.1.3 Discussion

The programming system, while not perfect, was usable. All subjects were able to work with the software to examine and modify programs. The performance of the users significantly exceeded expectations, which were that about half the subjects would be able to successfully modify an existing program. Further tests should include more challenging tasks. In particular, it would have been useful to see the subjects’ performance on a task that required writing a new program without using an existing program as a template.

The behavior browser seemed to work well, though there seemed to be a greater reliance on names than expected. The animation of each program was useful but was not the primary means of search. The ability to click on the name of a program and bring up an animator-editor with that program was not obvious, though four subjects eventually discovered it. The browser should be redesigned to include a larger font for the program names and better visual cues signifying that the name is an active control.

The editor-animator was not used to its full potential. That may be due to the limited nature of the tasks and the short time the users had to work with the tool. One feature of the editor-animator that was a consistent problem was the “check” button, whose purpose was unclear to all but one of the users with previous programming ex-

perience. All but one of the users switched repeatedly between editing and animating a program.

There was no difficulty relating the motion of the animation to the physical rototack. This result may have been affected by introducing the users to the programming environment before they had a chance to do more than briefly inspect the rototack.

The download window was confusing and overly complex. Many of the controls on the window, such as the synchronization adjustments and reset button (see Figure 3.7), were needed only because the communications hardware was not sufficiently robust. These controls should be eliminated or hidden. Although there were no specific complaints, the program start and stop button seemed to take longer to understand and use than the controls on the other programming tools. This tool needs to be simplified and refined through additional user testing. The program start and stop controls may need to be moved to a new window.

While there were an insufficient number of subjects to draw meaningful quantitative results from the data, the fact that all six subjects were able to perform finding, understanding, and modification tasks using the programming software is very encouraging.

6.2 Future Work

There are three important goals that helped motivate this work and should continue to motivate future work in this area. The first is the understanding of what it means to combine computation and physical objects to produce computationally enhanced materials. The second is an understanding of how crafters might use CECIs in their craft. The third is how these ideas can be generalized to apply to areas beyond CECIs, crafters, and the craft culture.

In the short term, there are several areas of study suggested by the work reported here. One interesting area is the exploration of the design space of CECIs that

have approximately the same level of complexity as the prototypes described in Chapter 2. This area is particularly interesting for short, focused research efforts[1] which would consist of designing, prototyping, and testing new kinds of CECI. Another area of study concerns parameterizing programs and adjusting their behavior in the field. It may be best to explore this outside of the craft domain, in an area like children's toys, where many smart but non-programmable objects already exist. Finally, there are many opportunities to test alterative programming systems such as visual programming languages and natural language specification of programs on CECIs.

The prototype CECIs built as part of this work are medium to high end CECIs. There are many simpler forms that "smart" craft materials can take that may require different approaches to programming and use. Examples might be pieces of string that can lengthen or shorten themselves, tape that changes color over a (controlled) period of time, and fasteners that can lock and unlock themselves. These objects may be so simple that programming is not required at all—selection from a variety of different behaviors and the ability to adjust a few parameters may be sufficient.

In Chapter 2, the problems associated with using computation as a type of craft material were mentioned. In this study, those problems were avoided by combining the computation with existing craft materials in a CECI. Behaviors were the necessary bridge between the physical and computational elements of a CECI. It may be possible to create a new kind of craft material that focuses on behaviors rather than computation or physical material. An exploration of new kinds of craft materials based on behaviors may uncover new ways for crafters and others to combine physical objects with computation. One approach to this would be to focus on craft items that already embody a single behavior. Examples might be a motor or a push button. Another approach would be to choose a particular behavior and create a new kind of CECI to embody that behavior. An example of this might be creating a sticker whose sole

[1] In particular, this sort of exploration may be appropriate for undergraduate research projects.

(computationally controlled) behavior is changing its color.

The concept of libraries of behaviors and the devices that might implement those behaviors was touched on in this work, but not fully developed. Such a library could be a valuable resource for researchers and developers trying to combine elements of the physical and computational world. Formalizing the behaviors identified in this work combined with a search of the literature in sensors and actuators, MEMS, and embedded computing should be sufficient to develop the basic categorization schemes, vocabulary, and abstractions needed to create such a library.

Finally, and most importantly, this work requires feedback from real crafters using these materials. This can be accomplished either academically, by building larger numbers of more robust prototypes and running controlled studies. Or it could be done by producing CECIs as commercial products, which can then be given a large, though not terribly controlled, test in the real world.

6.3 Core Contributions

The question that guided this research was largely answered in Chapter 4, which defines and discusses the minimal set of technical solutions required to make the construction and programming of CECIs feasible. More accurately, it discusses the technical solutions that were *interesting*: either because they were unexpected or because they required innovation to fit to the problem. While these are significant, this work produced a number of other meaningful contributions.

The first of these is a model for the design and creation of programmable *materials*—objects intended to be used to build larger structures—that combine physical materials and computation. This model includes a general approach: adding behaviors and computation to existing materials (in Chapter 2), an architecture for the computation and control software (sections 2.3, 4.1, and 4.2), and a set of concepts and vocabulary with which to discuss it (introduced throughout the thesis, but mainly in

Chapter 2).

Behaviors, which can be thought of as abstractions of specific ways of interating with the world, were discussed in this work as part of the model described above, where they were used as a bridge between computation and the physical world. Discussion of the role of behaviors has here been limited to their use in craft, but behaviors are a more general idea that can be applied to the study of controlling actuators and sensors, ubiquitous computing, tactile user interfaces, and other areas of study that combine computation with the physical world.

The affordances of the environment in which crafters program CECIs are unique. The programming system takes advantage of these affordances by using a simplified programming language and a programming environment whose tools are designed for the kinds and sizes of tasks that crafters encounter when programming CECIs. Some of the features of the programming system design and the rationale behind those features apply to areas outside the realm of CECI programming. In particular, the affordances of short programs and small numbers of concrete primitives can apply to many situations where simple objects based on physical systems or metaphors are being programmed. Some specific areas where this might be applied are ubiquitous computing research, wearable computing, and simulations of physical systems.

In summary, the set of technical solutions discussed in Chapter 4 is the part of this work's contribution that was implied by the research question. Other contributions were models, concepts, and design elements useful in areas where programmable objects and materials that combine computation and the physical world are used.

6.4 A Final Thought

Sitting on my desk as I write this is a toy robotic cat. Variations of toy robotic pets were a major seller during the last holiday season, and smart toys of all kinds are becoming the norm. This is not limited to toys. Television sets, refrigerators, thermostats, and even the lowly toaster are all being made "smart" by the inclusion of small amounts of computation. To use Nicholas Negroponte's terminology [89], the world of "bits" is invading the world of "atoms" at an increasing rate.

At the beginning of this thesis I mentioned this idea, and describe how the ubiquitous computing community predicts that these bits of computation will become so common that they will fade into the background.

I disagree. While it is true that for the most part these objects *will* be unnoticed, there will be times when we will be forced to notice. This will happen when these bits of computation break, when—as is discussed here in great detail—they are used as building material, and when they simply do not work the way *we* need them to work. My question is, when we are forced to notice these "bits" because they are not acting the way we need them to, what will we be able to do about it?

Right now it looks as if the answer might be "not much". These devices are black boxes that we cannot see into to understand, much less control. Take the little robot cat for example. It is certainly cute and it is obvious the programmers went to a lot of trouble to make it act in very non-robotic ways. One of those ways is that, like a real cat, it lacks an off switch.

That is also why it is sitting on my desk. Rather than having an off switch, it shuts itself off when it is left alone. Unfortunately it is programmed to mimic annoyance and anger when it is left alone for more than a few minutes; its eyes flash and it growls loudly. Since these actions frighten my daughter, the toy sits on my desk. It is a smart toy, but not smart enough to let its behavior be changed when necessary.

Is this the future of ubiquitous computing? Will we have to hope that the way we want these objects to behave is sufficiently near the demographic norm that there will not be any problems? Or do we have to wait until they become smart enough to adapt to the behavior we want? That will be quite a trick; even my friends and family cannot always tell what I want.

Another answer is to give some kind of programmatic control to users. It does not have to be perfect. If someone needs to change the behavior of one of these devices and does not know how to program, they can always call in a professional (or maybe the ten year old who lives next door). The point is that without some level of end-user programmability, even that option is unavailable. The only choices are to replace the object, which may or may not help, or put up with suboptimal behavior.

In the end, it is a question of who controls the computation. Currently the manufacturers keep that control for themselves. Maybe it is time for at least some of that control to shift to the users.

Bibliography

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, et al. Amorphous computing. Technical Report AI Memo 1665, MIT Artificial Intelligence Laboratory, 1999.

[2] Gregory D. Abowd and Elizabeth D. Mynatt. Charting past, present, and future research in ubiquitous computing. ACM Transactions on Computer-Human Interaction, 7:29–58, MAR 2000.

[3] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts, 1986.

[4] Gul Agha. Concurrent object-oriented programming. Communications of the ACM, 33(9):125–141, Sep 1990.

[5] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. Compilers, Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[6] Kalle Anderson, Jason Buttron, Paul Clarke, and Matt Enwald. Wookie: A 68h11 emulator. Dr. Dobb's Journal, March 1999.

[7] Ernesto Arias, Hal Eden, and Gerhard Fisher. Enhancing communication, facilitating shared understanding, and creating better artifacts by integrating physical and computational media for design. In Proceedings of the Conference on Designing Interactive Systems : Processes, Practices, Methods, and Techniques, pages 1–12. ACM Press, 1997.

[8] Jesse Aronson and Prasanta Bose. A model-based approach to simulation composition. In Proceedings of the Fifth Symposium on Software Reusability, pages 73–82. ACM Press, 1999.

[9] Jan Axelson. Control and automation interfaces overview. The Home Automation Times, Jul 1999. Available electronically at http://www.homeautomationtimes.com/.

[10] David Benson. Easy PIC'n. Square One Electronics, Kelseyville, California, 1998. www.sq-1.com.

[11] Andrew A. Berlin and Kaigham J. Gabriel. Distributed mems: New challenges for computation. IEEE Computational Science and Engineering, 4(1):12–16, Jan 1997.

[12] Glenn Blauvelt and Michael Eisenberg. Machine shop: Steps toward exploring novel i/o devices for computational craftwork. In Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT2001), 2001. Forthcoming.

[13] Amy Bruckman. Programming for fun: Muds as a context for collaborative learning. In Proceedings of the National Education Computing Conference, 1994. Held in Boston, MA, June 1994.

[14] Amy Bruckman and Elizabeth Edwards. Should we leverage natural-language knowledge? an analysis of user errors in a natural-language-style programming language. In Proceeding of the CHI 99 Conference on Human Factors in Computing Systems : The CHI is the Limit, pages 207–214. ACM Press, 1999.

[15] Margaret M. Burnett. Visual programming. In John G. Webster, editor, Encyclopedia of Electrical and Electronics Engineering. John Wiley and Sons, Inc., New York, NY, 1999.

[16] William Buxton. Lexical and pragmatic considerations of input structures. Computer Graphics, 17(1):31–37, Jan 1983.

[17] William Buxton. A three-state model of graphical input. In Human-Computer Interaction - INTERACT'90, pages 449–456, Amsterdam, 1990. Elsevier Science Publishers B. V.

[18] Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. A morphological analysis of the design space of input devices. ACM Transactions on Information Systems, 9(2):99–122, Apr 1991.

[19] Theodore Y. Chen. Hyperspider: Integrating computation with the design and construction of educational crafts. Master's thesis, University of Colorado, 1999.

[20] Randy Clark and Stephen Koehler. The UCSD Pascal Handbook: A Reference Guidebook for Programmers. Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[21] Adam Cohen. Near-field photolithography. Westinghouse Competition paper, 1996–1997.

[22] Allen Cypher. Customizing application programs. In Proceedings of the Moscow International Workshop on Human-Computer Interaction, Aug 1991.

[23] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Watch What I Do: Programming by Demonstration, pages 205–217. MIT Press, Cambridge, Massachusetts, 1993.

[24] Allen Cypher. Introduction to What What I Do. MIT Press, Cambridge, Massachusetts, 1996.

[25] Michael Dertouzos. The user interface is the language. In Brad A. Myers, editor, Languages for Developing User Interfaces, pages 21–30. Jones and Bartlett Publishers, Inc., Boston, Massachusetts, 1992.

[26] Chris DiGiano and Michael Eisenberg. Self-disclosing design tools: A gentle introduction to end-user programming. In Proceedings of Designing Interactive Systems (DIS'95), pages 189–197, 1995.

[27] K. Eric Drexler. Nanosystems: Molecular Machinery, Manufacturing, and Computation. Wiley Interscience, New York, NY, 1992.

[28] Dunfield Development Systems, Ottawa, Ontario. Emily52 Simulator Manual, 2000. Emily52 is a 8051 and Dallas 80C320/520 simulator.

[29] Scott Edwards. Programming and Customizing the Basic Stamp Computer. McGraw-Hill, 1998.

[30] Ann N. Eisenberg. An Educational Program for Paper Sculpture. PhD thesis, University of Colorado, 1999.

[31] Michael Eisenberg. Programmable applications: Interpreter meets interface. SIGCHI Bulletin, 27(2):68–93, 1995.

[32] Michael Eisenberg and Ann N. Eisenberg. Middle tech: Blurring the division between high and low tech in education. In Allison Druin, editor, The Design of Children's Technology, pages 244–273. Morgan Kaufmann, San Francisco, California, 1999.

[33] Michael Eisenberg and Ann Nishioka Eisenberg. Shop class for the next millennium: Education through computer-enriched handicrafts. Journal of Interactive Media in Education (JIME), Oct 1997. JIME is available online at http://www-jime.open.ac.uk/.

[34] Michael Eisenberg and Ann Nishioka Eisenberg. Shop class for the next millennium: Education through computer-enriched handicrafts. Journal of Interactive Media in Education, Oct 1998.

[35] Michael Eisenberg, Andi Ruben, and Ted Chen. Computational and educational handicrafts: A strategy for integrative design. In Proceedings of International Conference on the Learning Sciences, 1999.

[36] CTI Electronics. Train brain software (version 6.0), 2001. www.cti-electronics.com.

[37] Gerhard Fasol. Nanowires: Small is beautiful. Science, 280:545–546, 1998. Issue of 14 April 1998.

[38] Gerhard Fischer, David Redmiles, Lloyd Williams, Gretchen I. Puhr, Atsushi Aoki, and Kumiyo Nakakoji. Beyond object-oriented technology: Where current approaches fall short. Human-Computer Interaction, 10:79–119, 1995.

[39] III Frank M. Shipman and Raymond McCall. Supporting knowledge-base evolution with incremental formalization. In Proceedings of the CHI '94 Conference Companion on Human Factors in Computing Systems, pages 220–206. ACM Press, 1994.

[40] III Frank M. Shipman and Raymond J. McCall. Incremental formalization with the hyper-object substrate. ACM Transactions on Information Systems (TOIS), 17(2):199–227, 1999.

[41] Phil Frei, Victor Su, Bakhtiar Mikhak, and Hiroshi Ishii. curlybot: Designing a new class of computational toys. CHI Letters, 2:129–136, 2000.

[42] Konrad Froitzheim, Heiner Wolf, and Holger Bonisch. The interactive model railroad, 2001. ernst.informatic.uni-ulm.de:81/rr/gui2/.

[43] Jack Ganssle. The Art of Programming Embedded Systems. Academic Press, New York, NY, 1992.

[44] Adele Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison Wesley, Reading, Massachusetts, 1984.

[45] Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison Wesley, Reading, Massachusetts, 1983.

[46] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In Human-Computer Interaction: Tasks and Organization, Proceedings of the 6th European Conference on Cognitive Ergonomics), 1992.

[47] Thomas R. G. Green. Noddy's guide to visual programming. Interfaces (Newsletter of the British Computer Society Human-Computer Interaction Group), 1995. Also available at http://www.ndirect.co.uk/thomas.green/workStuff/Papers/NoddyOnVPLs.ps.

[48] Mark Guzdial. Software-realized scaffolding to facilitate programming for science learning. Interactive Learning Environments, pages 1–44, 1995.

[49] Mark Guzdial, Peri Weingrad, Robert Boyle, and Elliot Soloway. Design support environments for end users. In Brad A. Myers, editor, Languages for Developing User Interfaces, pages 57–78. Jones and Bartlett Publishers, Inc., Boston, Massachusetts, 1992.

[50] Daniel C. Halbert. Smallstar: Programming by demonstration in the desktop metaphor. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, Massachusetts, 1993. Based on work done from 1980–1984.

[51] C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Department of Computer Science, 1973.

[52] Daniel E. Hodgson, Ming H. Wu, and Robert J. Breimann. Shape memory alloys. Metals Handbook, 2, 1999.

[53] Grace Murray Hopper. Keynote address. In Richard L. Wexelblat, editor, History of Programming Languages, pages 7–24, New York, NY, 1978. Academic Press.

[54] Paul Horowitz and Winfield Hill. The Art of Electronics. Cambridge University Press, Cambridge, United Kingdom, second edition, 1989.

[55] R. Ierusalimschy, L.H. de Figueiredo, and W Celes. Lua-an extensible extention language. Software: Practice & Experience, 26#6:635–652, 1996.

[56] Carl's Electronics Inc. Ck1700-intro pic programmer (kit), 2001. www.electronickits.com.

[57] Gemmy Industries Inc. Douglas fir, 2000. www.gemmy.com.

[58] Microchip Technology Inc. Pic16x84 microcontroller family, 2001. www.microchip.com.

[59] Ottinger Display Co. Inc. Victorian caroler: Boy in green, 2001. www.ottingerdisplay.com.

[60] Intel Corporation, Santa Clara, California. Intel Celeron Processor up to 800 MHz Datasheet, 2001. www.intel.com.

[61] John Iovine. PIC Microcontroller Project Book. McGraw-Hill Professional Publishing, 2000.

[62] Hiroshi Ishii and Brygg Ullmer. Tangible bits: Towards seamless interfaces between people, bits and atoms. In Proceedings of CHI'97, New York, NY, 1997. ACM Press.

[63] Nicholas Jackiw and William F. Finzer. The geometer's sketchpad: Programming by geometry. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration, pages 292–307. MIT Press, Cambridge, Massachusetts, 1993.

[64] Michael Patrick Johnson, Andrew Wilson, Bruce Blumberg, Christopher Kline, and Aaron Bobick. Sympathetic interfaces: Using a plush toy to direct synthetic characters. In Proceedings of the CHI'99 Conference, pages 152–158, New York, NY, MAY 1999. ACM Press.

[65] Ken Kahn. Drawings on napkins, video-game animation, and other ways to program computers. Communications of the ACM, 39(8):49–59, 1996.

[66] Gene Korienek and Tom Wrensch. A Quick Trip to ObjectLand: Object-Oriented Programming in Smalltalk/V. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[67] John Kymisis, Clyde Kendall, Joseph Paradiso, and Neil Gershenfeld. Parasitic power harvesting in shoes. In Second IEEE International Conference on Wearable Computing (ISWC). IEEE Computer Society, OCT 1998.

[68] Tessa A. Lau and Daniel S. Weld. Programming by demonstration: An inductive learning foundation. In Proceedings of the 1999 International Conference on Intelligent User Interfaces (IUI99), 1999. Held January 1999 in Redondo Beach, California.

[69] Lego-Dacta. Lego mindstorms, 2000. www.lego.com.

[70] Henry Lieberman. Tinker: A program by demonstration system for beginning programmers. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration, pages 49–63. MIT Press, Cambridge, Massachusetts, 1993. Based on work done from 1980–1984.

[71] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley, Reading, Massachusetts, 1997.

[72] Bill Manaris. Natural language processing: A human-computer interaction perspective. In Marvin V. Zelkowitz, editor, Advances in Computers, volume 47, pages 1–66. Academic Press, New York, NY, 1998.

[73] Fred Martin. Handyboard, 2000. www.handyboard.org.

[74] Fred Martin, Bakhtiar Mikhak, Mitchel Resnick, Brian Silverman, and Robbie Berg. To mindstorms and beyond: Evolution of a construction kit for magical machines. In Allison Druin and James Hendler, editors, Robots for Kids: Exploring New Technologies for Learning. Academic Press, San Diego, California, 2000.

[75] Fred Martin, Bakhtiar Mikhak, and Brian Silverman. Metacricket: A designer's kit for making computational devices. IBM System Journal, 39:3-4:795–815, 2000.

[76] David Maulsby and Ian H. Witten. Inducing programs in a direct-manipulation environment. In Proceedings of the SIGCHI Conference on Wings for the Mind, pages 57–62. ACM Press, Apr 1989.

[77] David Maulsby and Ian H. Witten. Metamouse: An instructible agent for programming by demonstration. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration, pages 155–182. MIT Press, Cambridge, Massachusetts, 1993.

[78] Malcolm McCullough. Abstracting Craft: The Practiced Digital Hand. MIT Press, Cambridge, Massahusetts, 1996.

[79] Richard G. McDaniel and Brad Myers. Getting more out of programming-by-demonstration. In Proceedings of CHI'99, pages 442–449. ACM Press, May 1999.

[80] Jim McRae. Apples for Teachers—101 programs for the classroom. Tab Books, Blue Ridge Summit, PA, 1985.

[81] Inc. Metroworks. Codewarrior, 2000. www.codewarrior.com.

[82] Microchip Technology Inc., Chandler, Arizona. MPLAB: IDE, Simulator, Editor User's Guide, 2000. www.microchip.com.

[83] Inc. Microsoft. Visual c++, 2000. www.microsoft.com.

[84] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In Proceedings of ASA / MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents, 1999.

[85] Brad A. Myers. Introduction. In Brad A. Myers, editor, Languages for Developing User Interfaces, pages 1–17. Jones and Bartlett Publishers, Inc., Boston, Massachusetts, 1992.

[86] Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Varner Zanden, et al. The garnet user interface development environment. In Conference Proceedings on Human Factors in Computing Systems, pages 293–300. ACM Press, 1994.

[87] Bonnie A. Nardi. A Small Matter of Programming. The MIT Press, 1995.

[88] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end user programming. Technical Report HPL-90-08, Hewlett Packard, 1990.

[89] Nicholas Negroponte. Being Digital. Alfred A. Knopf, Inc., New York, New York, 1995.

[90] Jon Palfreman and Doron Swade. The Dream Machine: Exploring the Computer Age. BBC Books, London, United Kingdom, 1991.

[91] Seymour Papert. Mindstorms: Children, Computers, and Powerful Ideas. Basic Books, New York, NY, 1980.

[92] Seymour Papert. The Children's Machine: Rethinking School in the Age of the Computer. Basic Books, 1993.

[93] Inc. Parallax. The basicstamp computer, 2000. www.parallax.com.

[94] Richard Potter. Triggers: Guiding automation with pixels to achieve data access. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, Massachusetts, 1993.

[95] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. Naturaljava: A natural language interface for programming in java. In Proceedings of the 2000 International Conference on Intelligent User Interfaces, pages 207–211. ACM Press, 2000.

[96] R and S Enterprises. Rr-track sectional track layout software, 2001.

[97] Cyndi Rader, Gina Cherry, Cathy Brand, Alexander Repenning, and Clayton Lewis. Designing mixed textual and iconic programming languages for novice users. In Proceedings of the 1988 IEEE Symposium on Visual Languages, pages 187–194, 1998. Held September 1-4, 1998 in Halifax, Nova Scotia, Canada.

[98] David F. Redmiles. From Programming Tasks to Solutions—Bridging the Gap through the Explanation of Examples. PhD thesis, University of Colorado, 1992.

[99] David F. Redmiles. Reducing the variability of programmers' performance through explained examples. In Conference Proceedings on Human Factors in Computing Systems, pages 67–73. ACM Press, Apr 1993.

[100] Alex Repenning. Agentsheets: an interactive simulation environment with end-user programmable agents. In Proceedings of Interaction 2000, 2000. Held in Tokyo, Japan Feb 29 thru Mar 1, 2000.

[101] M. Resnick, F. Martin, R. Sargent, and Silverman B. Programmable bricks: Toys to think with. IBM System Journal, 35:3:443–452, 1996.

[102] Mitchel Resnick, Fred Martin, Robert Berg, et al. Digital manipulatives. In Proceedings of the CHI '98 Conference, New York, NY, APR 1998. ACM Press.

[103] Bradley J. Rhodes, Nelson Minar, and Josh Weaver. Wearable computing meets ubiquitous computing: Reaping the best of both worlds. In The Proceedings of The Third International Symposium on Wearable Computers (ISWC '99), pages 141–149. IEEE Computer Society, OCT 1999.

[104] Wayne Roderick. Computer integration on the teton short line, 1999. www.ida.net/users/tetonsl/railroad/computer.htm.

[105] Start Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, NJ, 1995.

[106] L. McDonald Schetky. Shape memory alloys. Scientific America, 241:74–82, Nov 1979.

[107] Michael L. Scott. Programming Language Pragmatics. Morgan Kaufmann Publishers, 2000.

[108] Ravi Sethi. Programming Languages: Concepts and Constructs. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[109] Ben Shneiderman. Software Psychology: Human Factors in Computer and Information Systems. Winthrop Publishers, Cambridge, Massachusetts, 1980.

[110] David Canfield Smith, Allen Cypher, and J Spohrer. Kidsim: Programming agents without a programming language. Communications of the ACM, 37(7):54–67, Jul 1994.

[111] Crafted Software. Mosaic 2000, 2000. www.mosaicdesigner.com.

[112] Sandia Software. Cadrail railroad design software, 2000. sandiasoftware.com.

[113] Brygg Ullmer, Hiroshi Ishii, and Dylan Glas. mediablocks: Physical containers, transports, and controls for online media. In Computer Graphics Proceedings (SIGGRAPH'98). ACM Press, JUL 1998.

[114] Virtual Micro Design, Bidart, France. UMPS Reference Manual, 1999. Available electronically from ftp://ftp.vmdesign.com/pub/.

[115] Mark Weiser. Some computer science issues in ubiquitous computing. Communications of the ACM, 36:75–84, AUG 1993.

[116] K. N. Whitley. Visual programming languages and the empirical evidence for and against. Journal of Visual Languages and Computing, 8:109–142, 1997.

[117] Ian H. Witten and Dan Mo. Tels: Learning text editing tasks from examples. In Allen Cypher, editor, Watch What I Do: Programming by Demonstration, pages 183–204. MIT Press, Cambridge, Massachusetts, 1993.

[118] William Woods. Progress in natural language understanding: An application to lunar geology. In AFIPS Conference Proceedings, volume 42, pages 441–450, 1972.

[119] Thomas Wrensch, Glenn Blauvelt, and Michael Eisenberg. The rototack: Designing a computationally enhanced craft item. In W. E. Mackay, editor, Proceedings of DARE 2000, Designing Augmented Reality Environments, pages 93–102, 2000. Held in Elsinore, Denmark, April 2000.

[120] Thomas Wrensch and Michael Eisenberg. The programmable hinge: Toward computationally enhanced crafts. In Proceedings of UIST'98, pages 89–96, 1998. Held in San Francisco, CA, November 1998.

[121] Xilinx. Xilinx Data Book 2000: HP-30 Programmer, 2000.

[122] Inc. Zilog. Z8 family of microcontrollers, 2001. www.zilog.com.

[123] Thomas G. Zimmerman. Wireless network digital devices: A new paradigm for computing and communication. IBM Systems Journal, 38(4), 1999.

Appendix A

Rototack Programming Language Specification

This appendix includes a brief specification of the CECI programming language used on the final prototype of the rototack. The specification is divided into the core language elements and the extensions specific to rototacks.

A.1 Core Language Elements

This section includes the elements of the language that are the same across different types of CECIs.

A.1.1 Comments on Syntax

The CECI programming language does not use statement terminators or arbitrary blocks of code. Blocks are implicit in control structures. All blocks end with the `END` keyword. Expressions are not equivalent to statements.

Comments start with a semicolon (`;`) and go until the end of the line. Semicolons can be doubled to increase readability, but this is not required.

Keywords can not be used as constant or parameter names. The following are the keywords in the core language:

```
A           AND         ARGUMENT    B           C
CONSTANT    D           DO          ELSE        END
IF          LOOP        ME          MESSAGE     MSGIN
NOT         OR          PARAMETER   RANDOM      REPEAT
```

```
SEND        THEN        TO          WAIT        WHILE
WAITUNTIL   WHILE       WITH
```

The language also includes unit words. These are used to specify the meaning of a numeric value or to define one of a small set of values. Since a unit word is recognized as part of the statement that uses it, they do not have the same restrictions as keywords. In the core language, only the `WAIT` makes use of unit words. These are: `TENTHS`, `SECONDS`, and `MINUTES`.

A.1.2 Variables

The language has a single, global scope. There is one variable type: *integer*. The prototype CECIs use single byte integers. The global variables are predefined with the names `A`, `B`, `C`, and `D`. Variable names are case insensitive. Assignment is done using an assignment statement and assignments cannot be used as expressions. The `=` symbol is used to identify assignment. Note that this symbol is also used to identify the equality comparison operation (see section A.1.5).

Examples:

```
A = 1

B = B + A - 1
```

A.1.3 Control Structures

Control structures are limited to the basic conditional and iteration statements and a few special-purpose variations useful in programming control systems. Blocks are implied where needed and must be explicitly terminated with the `END` keyword except in the case of the Then block in an If-Then-Else statement.

If-Then. Conditional execution statement. Must be terminated by `END`.

If-Then-Else. Either-or execution statement. Code for the Then and Else cases are both blocks, but only the Else block requires a terminating `END`. The `ELSE` serves to terminate the Then block.

While. Iterate while a condition is true. The condition is tested before the loop is executed. Must be terminated with `END`.

Repeat. Iterate the number of times specified by an expression. Must be terminated with `END`.

Loop. Infinite loop. Must be terminated with `END`. Note that there is no way to break out of the loop without resetting the device.

Waituntil. Wait for some condition to become true. Does not include a code block, only a condition.

Examples:

```
IF A > 10 THEN
    A = 1
    TURN 1 CLICK
    END

IF MsgIn AND Message = 1 THEN
    REVERSE
    FLASH
ELSE
    TURN 1 CLICK
END

WHILE inboard DO
    WAIT 1 SECOND
    LED RED
    WAIT 5 TENTHS
    LED GREEN
    END

REPEAT A + 1 DO
    FLASH
    END
```

```
LOOP
     TURN 40
     REVERSE
     END

     WAITUNTIL inboard AND msgin AND message = 10
```

A.1.4 Parameters and Constants

While variables do not require declarations, constants need a declaration to associate a value with a name. Constant and Parameter declarations may appear anywhere in the program but their scope is always the entire program.

Constant. Named constants are created with the Constant declaration.

Parameter. Parameters are a form of constant that can be modified by an external device sending a system message. They are created with a Parameter declaration.

Examples:

```
CONSTANT InMessage = 42

CONSTANT Countdown = 10

PARAMETER DelayTime = 100

PARAMETER AmountToTurn = 5
```

A.1.5 Arithmetic and Comparison

The language includes a minimal set of arithmetic, logical, and bitwise manipulation functions. These are parsed with the usual precedence rules, whose ordering is arithmetic, comparison, binary logical, and unary. The language does not differentiate between boolean and integer expressions, any operator can be used for any expression.

Numeric Constants. Numeric constants are specified in base 10. Negative numbers are allowed (with a leading -), but are transformed into their unsigned equivalent.

Arithmetic. Addition and subtraction are provided with the binary operators + and -.

Comparison. The six standard numeric comparison operators are supported: =, <>, <, >, <=, and >=.

Boolean constants. The names `true` and `false` are predefined to be equivalent to the values -1 and 0 respectively.

Boolean operators. The three basic boolean operations are defined as the unary operator `NOT` and the binary operators `AND` and `OR`.

Bitwise operators. The boolean operators above serve as bitwise operators on integers.

Random. A function that returns a random integer between zero and one less than the argument value.

Examples:

```
A = A + 1 - B

IF (RANDOM 10) < 5 THEN
    REVERSE
    END

IF MSGIN AND (MESSAGE AND 3) = 0 THEN
    SEND (MESSAGE OR 1) TO Dewy WITH 0
    END

IF NOT INBOARD THEN
    REPEAT 10 DO FLASH END
    WAITUNTIL InBoard
    END
```

A.1.6 Duration Timing

Program wait times can be specified using the `WAITUNTIL` statement, or by using fixed wait times. Different wait periods use different unit words.

Wait-Tenths. Pause program execution for a specified number of tenths of a second.

Wait-Seconds. Pause program execution for a specified number of seconds.

Wait-Minutes. Pause program execution for a specified number of minutes.

Pause. Pause program execution for 1 second.

Examples:

```
WHILE NOT InBoard DO WAIT 1 TENTH END

REPEAT 5 DO LED RED PAUSE LED OFF PAUSE END

LOOP
    WAIT 59 MINUTES
    REPEAT 6 DO
        TURN 1 CLICK
        WAIT 10 SECONDS
        END
    END
```

A.1.7 Communication Between CECI

CECIs communicate with each other via messages. A message includes the ID of the receiver, a message ID, and an optional argument value. The receiver and message IDs are integer values. A receiver ID of 0 is a broadcast message.

Send. Send a message to a specified CECI with an optional argument value.

MsgIn. Function that returns the message-in flag. If the flag is true this function resets it before returning the value.

Message. Function that returns the message ID of the last message received.

Argument. Function that returns the argument value of the last message received.

Me. Function that returns the object ID of the CECI the program is running on.

Examples:

```
SEND StopMsg TO Tack4

IF MsgIn AND Message = StopMsg THEN
    D = false
    END

SEND wiggle TO 99 WITH 5

IF MsgIn AND Message = wiggle THEN
    REPEAT Argument DO
        CLOCKWISE TURN 2 CLICKS
        WAIT 1 TENTH
        CCWISE TURN 2 CLICKS
        END
    END

SEND ImHere TO 0 WITH ME
```

A.2 Rototack Specific Elements

This section specifies the rototack extensions to the core language, as developed for the Mark III prototype.

A.2.1 In Board Sensor

Access to the value of the sensor that detects when the rototack is pushed into a corkboard.

InBoard. A function that returns true if the in-board switch is on and false otherwise.

Example:

```
LOOP
    WAITUNTIL InBoard
    WHILE InBoard DO
        TURN 1 CLICK
        WAIT 4 TENTHS
        END
    END
```

A.2.2 Rotation

Commands to control and access the state of the rotation behavior.

Turn. Command to turn the rototack. Turn distance is specified in clicks, where one click equals nine degrees.

Step. Turn a single click.

Clockwise. Set the turn direction to clockwise.

CCWise Set the turn direction to counterclockwise.

Reverse. Reverse the turn direction.

IsClockwise. Function that returns true if the current direction is clockwise and false otherwise.

Examples:

```
CLOCKWISE LOOP STEP END ;; Turn clockwise forever

A = 0
REPEAT 20 DO
    A = A + 1
    TURN A CLICKS
    WAIT 5+A TENTHS
    REVERSE
    END

IF IsClockwise THEN CCWISE ELSE FLASH END
```

A.2.3 LED

The final (Mark III) prototype rototack had a bicolor LED implementing a colored light behavior. This section specifies the commands used to control it.

LED-Green. Change the LED color to green.

LED-Red. Change the LED color to red.

LED-Amber. Change the LED color to amber (both green and red together).

LED-Off Turn the LED off.

Flash Set the LED color to green for one tenth of a second followed by two tenths of a second with the LED off. Changes the LED state to off. Requires three tenths of a second to execute.

```
;; Sequence through colors
LOOP
    LED OFF
    PAUSE
    LED GREEN
    PAUSE
    LED RED
    PAUSE
    LED AMBER
    PAUSE
    END

WAITUNTIL InBoard
REPEAT 5 DO FLASH END
```

A.3 Disabled Language Features

A number of language features were implemented and then disabled. These were features that did not seem necessary, but were too important to take a chance on not having available if they were needed.

A.3.1 Extended Arithmetic Operations

The extended arithmetic operations included multiplication and division. These are added to the expression syntax with the * and / operators. The expression precedence order must be modified by adding these before the addition and subtraction operators in the precedence ordering.

Example:

```
;; Program to rotate and then wait.  The wait
;; time increases by a factorial function.
A = 1
B = 1
LOOP
    TURN 40 CLICKS
    WAIT A SECONDS
    B = B + 1
    A = A * B
    END
```

A.3.2 Read-Only Data Array

A single read-only data array is implemented using DATA statements. Reading data is done using a Read function and a Reset command.

Example:

```
;; Program that turns and waits based on
;; 10 data pairs
LOOP
    REPEAT 10
        TURN READ CLICKS
        WAIT READ TENTHS
        END
    RESET
    END
DATA 1,10, 9,5, 8,6, 2,9
DATA 9,1, 1,14, 5,9 5,6
DATA 1,1, 9,14
```

A.3.3 Subroutines and Call Mechanism

Subroutines and call mechanisms are implemented with the CALL, SUB, ENDSUB, and RETURN keywords. Subroutines can return either no value or one value. Note that this implementation does not use arguments.

Example:

```
;; Program that uses a subroutine to call a
;; group of behaviors that are used repeatedly.
clockwise
call dance
loop
    wait 10 seconds
    clockwise
    call dance
    ccwise
    call dance
    end
sub dance
    turn 5
    led red
    wait 2 tenth
    reverse
    led green
    turn 4
    wait 3 tenths
    turn 1
    led off
    endsub
```

A.3.4 Renaming

The fixed set of global variable names eliminates the user's need to define variable names, but also eliminates the ability to document variable usage by choosing appropriate names. An aliasing mechanism was implemented to overcome this by using a Rename declaration that replaces the Constant declaration.

Example:

```
;; Program showing use of RENAME
```

```
rename A to Counter
rename amber to yellow
rename 10 to Maximum
Counter = 1
Loop
    Counter = Counter + 1
    Repeat Counter
        LED yellow
        Wait 1 second
        LED off
        END
    IF Counter > Maximum Then
        Counter = 1
        End
    END
```

Appendix B

Sample Programs

The following program is a variation of the one used for all the smart tiles in a tile mosaic built from a combination of smart and normal tiles. The mosaic is a flower that grows and blooms during the day, only to wither and die at night (see Figure 2.1). The programs for the different tiles differ only in two parameter values.

This is a case where the artistry lies not in the programming, but in the design and construction of the physical craft project. All the tile programs have to do is switch between their natural color and their accent color at certain times. A single CECI sends out a timing 'tick' every hour to keep the different tiles synchronized.

```
;; Stem tile program for the blooming flower
;; mosaic. Turns on early and off late.
;;   on=tick1
;;   off = tick14
;; Note that the color change speed can be ignored
;; unless it takes more than ten minutes or so.
constant tick = 100
parameter ontick = 1
parameter offtick = 14
color base         ;; start with the natural color
Loop
    waituntil msgin
    if message = tick
        if argument = ontick
            color accent    ;; use highlight color
            end
        if argument = offtick
```

```
            color base      ;; back to natural color
            end
        end
    end
```

Assuming one of the smart tiles does double duty and generates the synchronization tick as well as changing color, then its program would be more complex, close to the limit for the prototype CECIs. This is the largest program created so far for a CECI.

```
;; Tile for the blooming flower mosaic. This tile also acts
;; as the timer for the entire mosaic. Since the duration
;; timers are not that accurate, it allows itself to be
;; synchronized by a touch early in the morning.
;; It assumes it is turned on in the zeroth hour, which is
;; the hour it will allow a touch to start the process.
constant tick = 1
parameter ontick = 3
parameter offtick = 8
color base
A = 0 ;; A is the hour counter.
Loop
    if A = 0 then
        B = 60
        while B > 0 do
            wait 1 minute
            if isTouched then
                B = 0
            else
                B = B - 1
            end
    else
        wait 60 minutes
    end
    A = A + 1
    if A = 24 then A = 0
    ;;
    ;; Now that the timing stuff is done, do my
    ;; own color changes if necessary.
    send tick to all with A
    if A = ontick then
        color accent
        end
```

```
    if A = offtick then
        color base
        end
    end
```

CECIs will be shipped with some kind of program already in place, even it it is just a program that does nothing at all. This default program could provide users with a simple set of behaviors which could be adjusted using the parameter mechanism. The following are possible default programs for rototacks, smart tiles, and programmable hinges.

```
;; Default program for a Rototack.
;; Produces back and forth motion with pauses
;; between each move.
parameter cwDist = 20  ;; half turn
parameter ccDist = 10  ;; quarter turn
parameter delay1 = 10  ;; one second
parameter delay2 = 10  ;; one second
loop
    clockwise
    turn cwDist clicks
    wait delay1 tenths
    ccwise
    turn ccDist clicks
    wait delay2 tenths
    end
```

```
;; Default program for a smart tile.
;; Turns on for a fixed time, can wait for
;; an external signal
parameter offtime = 5     ;; 5 minutes
parameter ontime = 20     ;; 20 seconds
parameter signal = 101    ;; external signal ID
parameter sigflag = false ;; wait for signal?
color off
Loop
    if sigflag then
        waituntil msgin and message = signal
    else
```

```
        wait offtime minutes
    end
    color on
    wait ontime seconds
    end
```

```
;; Default program for programmable hinge.
;; Opens and closes on an external signal.
parameter opendist = 120   ;; degrees to open
parameter opensignal = 91  ;; open signal ID
parameter closesignal = 92 ;; close signal ID
close fully
loop
    waituntil msgin
    if message = opensignal then
        opento opendist
        end
    if message = closesignal then
        close fully
        end
    end
```

During user testing (see Section 6.1), one subject who reported no previous programming experience wrote the following program for a rototack. They did not state a purpose for the program.

```
Clockwise
Wait 1 minute
Loop
ccwise
Turn 120 clicks
clockwise
Turn  120 clicks
Wait 1 minute
End
```

The following is a set of small programs that were used for small projects done using the rototack prototypes. They are all quite simple, suggesting that many useful tasks can be accomplished with very little code:

```
;; Turn a wheel to show a spiral optical illusion effect.
LOOP
    CLOCKWISE
    REPEAT 10 DO TURN 40 END
    WAIT 1 SECOND
    CCWISE
    REPEAT 10 DO TURN 40 END
    WAIT 1 SECOND
    END



;; WireDancer (fixed motion version)
CLOCKWISE
LOOP
    WAIT 3 TENTHS
    CLOCKWISE TURN 2 CLICKS
    CCWISE TURN 1 CLICK
    WAIT 1 TENTH
    CLOCKWISE TURN 2 CLICKS
    WAIT 1 TENTH
    CCWISE TURN 2 CLICKS
    CLOCKWISE TURN 1 CLICK
    WAIT 1 TENTH
    CCWISE TURN 2 CLICKS
    END



;; WireDancer (random motion)
LOOP
    WAIT RANDOM 2 TENTHS
    CLOCKWISE RANDOM 2 CLICKS
    CCWISE RANDOM 2 CLICKS
    END



;; Propeller turning attention getter
;; for corkboard
CCWISE
LOOP
    WAITUNTIL InBoard
    WHILE InBoard DO
        TURN 1 CLICK
        END
    END
```

```
;; Waving hand attention getter
;; for corkboard
CLOCKWISE
LOOP
    WAITUNTIL InBoard
    WHILE InBoard DO
        TURN 8 CLICKS
        WAIT 4 TENTHS
        REVERSE
        END
    END
```

Appendix C

Configuration Specification for a Rototack

This appendix shows the configuration file for the third iteration of the Rototack. This information is used by the programming environment to define the part of the programming language that is different for rototacks and to supply information necessary to run animations of the rototack.

There are three important kinds of entities defined in these files; behaviors, instructions, and commands. Behaviors describe the high-level behaviors for each CECI. The instructions describe the low-level operations on those behaviors as virtual machine instructions (single operations are mapped into a single instruction). Commands are the extensions to the programming language for a particular type of CECI.

```
;; Configuration file for the "Mark III" Rototack.
(name "Rototack-M3")

(behavior stepper rotation
    (stepsize 9)
    (delay 20)
    (startangle 0)
    (startdirection cw)
    (device stepper40))

(behavior rgled led
    (colors green red yellow)
    (start 0)
    (device BCLED-RG))

(behavior inswitch switch
```

```
    (start false)
    (onname "in")
    (offname "out")
    (device momentaryswitch))

(behavior timer waitclock)

(layout (stepper inswitch rgled timer))

(instruction left 35 (stepper left))
(instruction right 36 (stepper right))
(instruction rd 37 (stepper reverse))
(instruction isleft 38 (stepper isleft))
(instruction turn 39 (stepper turn))

(instruction ledoff 40 (rgled off))
(instruction ledg 41 (rgled color1))
(instruction ledr 42 (rgled color2))
(instruction leda 43 (rgled color3))
(instruction flash 44
    (computer sequence
        (rgled color1)
        (computer delay
            (timer tenths 1)
            (rgled off)
            (timer tenths 1) )))

(instruction isin 45 (inswitch ison))

(instruction waitt 32 (timer tenths))
(instruction waits 33 (timer seconds))
(instruction waitm 34 (timer minutes))

(command tackinboard
    (type f0)
    (keyword inboard)
    (form k)
    (code 45)
    (shortdesc "is tack pushed into a board"))

(command turnclicks
    (type c1)
    (keyword turn)
    (unitword clicks)
    (form kvu)
    (code e 39)
    (shortdesc "turn the tack"))
```

```
(command turnclick
    (type c1)
    (keyword turn)
    (unitword click)
    (form kvu)
    (code e 39)
    (shortdesc "turn the tack"))

(command turn
    (type c1)
    (keyword turn)
    (form kv)
    (code e 39)
    (shortdesc "turn the tack"))

(command step
    (type c0)
    (keyword step)
    (form k)
    (code 129 39)
    (shortdesc "turn a single click"))

(command clockwise
    (type c0)
    (keyword clockwise)
    (form k)
    (code 36)
    (shortdesc "set direction to clockwise"))

(command ccwise
    (type c0)
    (keyword ccwise)
    (form k)
    (code 35)
    (shortdesc "set direction to counter-clockwise"))

(command reverse
    (type c0)
    (keyword reverse)
    (form k)
    (code 37)
    (shortdesc "reverse direction"))

(command isright
    (type f0)
    (keyword isclockwise)
```

```
    (form k)
    (code 38 16)
    (shortdesc "true when direction is clockwise"))

(command isleft
    (type f0)
    (keyword isccwise)
    (form k)
    (code 38)
    (shortdesc "true when direction is counter-clockwise"))

(command led0
    (type c0)
    (keyword led)
    (unitword off)
    (form ku)
    (code 40)
    (shortdesc "turn LED off"))

(command led1
    (type c0)
    (keyword led)
    (unitword green)
    (form ku)
    (code 41)
    (shortdesc "turn LED green"))

(command led2
    (type c0)
    (keyword led)
    (unitword red)
    (form ku)
    (code 42)
    (shortdesc "turn LED red"))

(command led3
    (type c0)
    (keyword led)
    (unitword amber)
    (form ku)
    (code 43)
    (shortdesc "turn LED amber"))

(command flash
    (type c0)
    (keyword flash)
    (form k)
```

```
    (code 44)
    (shortdesc "brief green flash on LED"))

(command waittenths
    (type c1)
    (keyword wait)
    (unitword tenths)
    (form kvu)
    (code e 32)
    (shortdesc "wait for tenths of a second"))

(command waittenth
    (type c1)
    (keyword wait)
    (unitword tenth)
    (form kvu)
    (code e 32)
    (shortdesc "wait for tenths of a second"))

(command waitseconds
    (type c1)
    (keyword wait)
    (unitword seconds)
    (form kvu)
    (code e 33)
    (shortdesc "wait for seconds"))

(command waitsecond
    (type c1)
    (keyword wait)
    (unitword second)
    (form kvu)
    (code e 33)
    (shortdesc "wait for seconds"))

(command waitminutes
    (type c1)
    (keyword wait)
    (unitword minutes)
    (form kvu)
    (code e 34)
    (shortdesc "wait for minutes"))

(command waitminute
    (type c1)
    (keyword wait)
    (unitword minute)
```

```
    (form kvu)
    (code e 34)
    (shortdesc "wait for minutes"))

(command pause
    (type c0)
    (keyword pause)
    (form k)
    (code 129 33)
    (shortdesc "wait for one second"))
```