Pushpin Computing: a Platform for Distributed Sensor Networks

by

Joshua Harlan Lifton

B.A. Physics and Mathematics
Swarthmore College, 1999

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

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

Author____
Program in Media Arts and Sciences
August 9, 2002

Certified by____
Joseph A. Paradiso
Principal Research Scientist
M.I.T. Media Laboratory
Thesis Supervisor

Accepted by____
Andrew B. Lippman
Chairperson, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Pushpin Computing: a Platform for Distributed Sensor Networks

by

Joshua Harlan Lifton

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 9, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

A hardware and software platform has been designed and implemented for modeling, testing, and deploying distributed peer-to-peer sensor networks comprised of many identical nodes. Each node possesses the tangible affordances of a commonplace pushpin to meet ease-of-use and power considerations. The sensing, computational, and communication abilities of a ”Pushpin”, as well as a ”Pushpin” operating system supporting mobile computational processes are treated in detail. Example applications and future work are discussed.

Thesis Supervisor: Joseph A. Paradiso
Title: Principal Research Scientist, M.I.T. Media Laboratory

Pushpin Computing: a Platform for Distributed Sensor Networks

by

Joshua Harlan Lifton

The following people served as readers for this thesis:

Thesis Reader____

V. Michael Bove, Jr.
Principal Research Scientist, Object-Based Media Group
M.I.T. Media Laboratory

Thesis Reader____

Gerald Sussman
Matsushita Professor of Electrical Engineering
M.I.T. Department of Electrical Engineering & Computer Science

Acknowledgments

To **Joe Paradiso** for being a wonderful advisor and friend. I certainly would not have come this far without him. His moral support and technical expertise are unsurpassed.

To **V. Michael Bove, Jr.** and **Gerald Sussman**, my gracious readers, for making fertile the field of research on which I cultivated my small plot.

To all the staff at the Media Lab, especially **Linda Peterson**, **Lisa Lieberson**, and **NeCSys**, for making everything and anything possible.

To **Bill Butera** for his unwavering enthusiasm, encouragement, patience, openness, and helping hand through it all. Any commendable portions of this work are as much his as they are anyone's.

To **Michael Broxton**, **Cynthia Johanson**, and **Kirk Samaroo** for their dedication, curiosity, and hard work to make this project a reality. I expect to hear (and already myself proclaim) great things about them all.

To **Devasenapathi P. Seetharamakrishnan** for his intelligent discussions, probing questions, and impeccable sense of humor.

To the **Responsive Environments Group**, comrades of the finest caliber.

To my **grandparents** for reminding me to stop and smell the roses. To my **uncle** and **aunt** for providing a home away from home over the last many years. To my **brother**, whose kung-fu is better than mine, for being righteous and true. To my **parents** for shaping me into the person I am today and for their understanding of a sometimes bewildering son.

Finally, to **Tam** for making it all worthwhile.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"A cockroach has 30,000 hairs, each of which is a sensor. The most complex robot we've built has 150 sensors and it's just about killed us. We can't expect to do as well as animals in the world until we get past that sensing barrier."*
>
> *Rodney Brooks in* Fast, Cheap & Out of Control *[1]*

This thesis sprouts up at the convergence of two fundamental questions, where the answer to each lies in the posing of the other.

The first question naturally arises from the progress in sensing systems exhibited over the last twenty years. Sensors to transduce physical quantities from the real world into a machine-readable digital representation are advancing to the point where size, quality of measurement, manufacturability, and cost are no longer the major stumbling blocks to creating machines equipped with as much sensory bandwidth as some animals, if not people. We now begin to face a problem of our own devising – how do we communicate, coordinate, process, and react to the copious amount of sensory data just now becoming available to the machines we build? Although some success in harvesting and responding to multiple data streams originating from a quantity of sensors has been demonstrated (e.g. [2]), asymptotic refinements of current engineering practices will not suffice to solve this problem over the long term; using traditional sensing methods, even adding one more sensor to an array of a couple dozen sensors can often present a formidable challenge on both the hardware and software fronts. Clearly, as the number of sensors increases to thousands, millions and beyond, we must employ fundamentally different approaches to building, maintaining, and understanding sensor networks.

The second question is rooted in the problem of extracting elegance from complexity, an interdisciplinary study variously referred to as self-organization, emergent behavior, self-assembly, and complex adaptive systems, among other names. For lack of an agreed upon name, we will provisionally use the term 'emergent systems', as in a system in which elegance emerges from complexity. The field of emergent systems is itself an emerging field. It currently encompasses only an ill-defined perspective from which to view other fields and a small canon of case studies, such as anthills or traffic jams. Case studies of this sort are endemic to immature areas of study, indicating a lack of a unifying framework or context of applicability. The field of emergent systems provides neither an underlying theoretical framework that can be applied and empirically tested nor an overarching context within which to apply and test such a framework. Note both the framework and the context are crucial – theory stagnates without a compelling reason to apply it. Is there a compelling context for exploring emergent systems and, if so, what will be the theoretical framework defining our mode of exploration?

Both these questions complement each other, each suggesting the answer to the other. Namely, any tractable solution to communicating, coordinating, processing, and reacting to myriad streams of sensor data will have to rely on principles of emergent systems and self-organization at the level of the sensors themselves in order to guarantee proper scaling properties. Conversely, the most compelling applications of emergent systems are found in the realm of sensing and control/activation. In light of this convergence, it behooves us to begin treating sensor systems as distributed networks wherein each node is a self-sufficient sensing unit and coordination among nodes takes place locally, automatically, and without centralized supervision.

Such distributed sensor networks are immediately applicable to many real world applications spanning a large range of physical scales; robot skins, smart floors, battlefield reconnaissance, environmental monitoring, HVAC (heating, ventilation, air-conditioning) control, high-energy particle detectors, and space exploration are among the many applications that could benefit from distributed sensor networks. Perhaps the greatest use of distributed sensor networks, however, lies not in the preexisting applications they augment, but rather in the future applications they enable. Obviously, it is impossible to fully enumerate these future applications, but it is not hard to speculate that advances in a variety of fields will only make that list longer.

This thesis presents Pushpin Computing, a hardware and software test bed for quickly prototyping and exploring real-world distributed sensor networks. Central to both the hardware and software aspects of the Pushpin platform are notions taken from emergent systems, as mentioned above and examined in detail throughout the remainder of this document.

1.1 Synopsis

This document is divided into six chapters: *Introduction*, *Hardware*, *Software*, *Applications*, *Discussion & Evaluation*, and *Conclusions*. The remainder of this chapter (*Introduction*) offers a review of related work and an overview of the Pushpin project as a whole. The *Hardware* chapter details the design and implementation of each of the four types of modules comprising a Pushpin and the substrate through which the Pushpins receive power. The *Software* chapter details the design and implementation of the process fragments, the Pushpin operating system, and the user integrated development environment (IDE). The *Applications* chapter goes into possible/actual applications that can be/have been developed on the Pushpin platform and their results. The *Discussion* chapter ties up loose ends, attempts to step back to view the Pushpin project as a coherent whole, and conveys some of the lessons learned. The *Conclusions* chapter provides a final summary, possible future directions, and possible future applications that may stem from the Pushpin project. A list of references and appendices containing full technical details follow.

1.2 Related Work

Depending on the particular circumstances, the term *distributed sensor network* can meaningfully be attached to a large number of systems varying widely across many distinct parameters, such as physical layout, network topology, memory resources, computational throughput, sensing capabilities, communication bandwidth, and usability. Accordingly, what qualifies as research into distributed sensor networks is just as general. In such a context, everything from tracing TCP/IP packet flow through the Internet to quantifying collective ant behavior can be considered as examples of research into distributed sensor networks. Nonetheless, there are very specific bodies of research that either have directly inspired or are very closely related to the work presented here. They can be divided roughly between theories/simulations and hardware implementations.

1.2.1 Theories and Simulations

The direct inspiration for this work is Butera's Paintable Computing work [3], which demonstrated in simulation a compelling and scalable programming model for a computing architecture composed of a large number of small processing nodes scattered across a surface. Each simulated node was allowed to execute a small amount of mobile code and communicate directly only with its spatially proximal neighbors. Pushpin Computing started out as an attempt to instantiate in hardware as closely as possible the Paintable simulations. This will be discussed in detail in Chapter 3.

Paintable Computing, in turn, is in part inspired by the work of the Amorphous Computing Group [4] at the MIT Artificial Intelligence Laboratory and Laboratory for Computer Science. That group's simulation work in synchronization and self-organizing coordinate systems is of particular interest.

Resnick's StarLogo programming language [5] provides an accessible but rich simulation environment for exploring decentralized emergent systems. Essentially, StarLogo is a parallelized version of the Logo programming language; where Logo has but a single 'turtle', StarLogo has many. The Pushpin programming model is influenced by StarLogo's intuitive approach to uncovering distributed algorithms.

Of course, speaking of StarLogo starts us down the slippery slope of Complex Adaptive Systems [6], a field too dispersed and vague to be properly treated here. Suffice it to say that there is a large body of related work to be found in the cellular automata and artificial life domains, including Conway's canonical 'Game of Life' [7]. One recent example that stands out is a body of theoretical work by Shalizi [8], dubbed Computational Mechanics, which proposes a functional definition of self-organization, whereby a system is considered self-organizing if it is computationally less expensive to simulate it than to predict it. Although demonstrated only within the confines of time series and cellular automata, this framework may prove to be applicable in a broader sense.

On a more applied level, there are several examples of simulations of distributed networks used for tuning parameters or testing algorithms for an actual distributed network [9]. Similarly, although MIT's μAMPS project does not produce hardware as such, it has produced software tools for profiling the energy and simulating the protocols of a distributed sensor network [10]. The group's work revolves almost exclusively around energy efficient algorithm design for distributed sensor networks [11, 12, 13].

On a higher level, Seetharamakrishnan is currently designing and implementing a programming language and compiler for distributed sensor networks [14]. Such a combination is meant to allow a programmer to write a single piece of code specifying the behavior of the global system. This one piece of code would then get compiled down to many pieces of smaller code meant to run on the devices that comprise the distributed sensor network. Although still in its infancy, the idea behind this work is a powerful one that may prove to be the key to realizing the full potential of distributed sensor networks.

1.2.2 Hardware Implementations

Although there are surely many more examples of computer simulation research that have some bearing on distributed sensor networks, there are relatively few hardware platforms designed in the same vain as the Pushpins. An early example of such a platform is the Amorphous Computing group's Gunk on the Wall project [15], which networked together many simple computing nodes. Each node is comprised of a microprocessor, perhaps a sensor, and a read-only memory where user programs are stored. Although a good proof-of-concept, this project is quite limited by the nature of the hardware used.

UC Berkeley's (now Intel Research Lab at Berkeley) SmartDust and its associated TinyOS software environment exemplify a more recent attempt at a hardware and software platform for distributed wireless sensing. The SmartDust/TinyOS platform was developed from the bottom up, shaped by the real-world energy limitations placed upon nodes in a distributed sensor network [16, 17]. As such, each node is relatively resource poor in terms of bandwidth and peripherals. Furthermore, the assumption is made that almost all communication within a distributed sensor network is for the purpose of communicating with a centralized base station [18]. In contrast, the Pushpin platform was built more from the top down, provides each node with a rich set of hardware, bandwidth, and software, and consumes correspondingly more energy per node.

The UCLA Laboratory for Embedded Collaborative Systems (LECS) also places a strong emphasis on distributed sensor networks, particularly network routing, time synchronization, and energy efficiency [19, 20, 21]. Most of their published work seems to be simulation based, although they are involved in a collaboration with USC's Robotics Embedded Systems Lab, which developed the Robomote [22], a very small autonomous two-wheeled robot equipped with a modest sensor suite.

The MIT Media Lab's now defunct Personal Information Architecture group produced a couple of proof-of-concept distributed networks, culminating in TephraNet, an ensemble of sensing nodes eventually deployed in Hawaii's Volcanoes National Park [23, 24]. Also from the Media Lab is the Epistemology and Learning group's Tiles project resulted in a set of children's blocks each with an embedded microcontroller that could pass mobile code from neighbor to neighbor [25]. Although the primary motivations for the Tiles project and the Pushpin project differ substantially, the final results are surprisingly similar. This topic will be discussed further in Chapter 5.

That the study of distributed sensor networks is becoming a field unto itself is evidenced by the fact that numerous efforts are leaving academia and resurfacing within industry as companies, such as Ember [26] and Sensoria [27].

1.3 Pushpin Computing Overview

Pushpin Computing is founded on principles of algorithmic self-assembly among many independent nodes, each capable of communicating with its immediate neighbors. Critically, the Pushpin platform not only provides a robust sensing platform, but also implements the unique programming model put forth by Paintable Computing. The initial test bed embodied by the Pushpin platform, provides a means, both in terms of hardware and software, for exploring algorithms and techniques for building self-organizing distributed sensor networks.

1.3.1 Design Points

The primary motivator for the Pushpin Computing project is to achieve the one goal inaccessible to computer simulations of distributed sensor networks – to sense and react to the physical world. The goal is to devise sensor networks that self-organize in such a way so as to preprocess and condense sensory data at the local sensor level before (optionally) sending it on to more centralized systems. Although the topologies may differ, this idea is somewhat analogous to the way the cells making up the various layers of a retina interact locally within and across layers to preprocess some aspects of contrast and movement before passing the information on to the optic nerve and then on to the visual cortex [28].

The Pushpin platform is comprised of approximately 100 computing nodes (Pushpins), each conforming to the following general criteria:

- Each Pushpin has the ability to communicate locally, however unreliably, with its spatially proximal neighbors, the neighborhood being defined by the range of the mode of communication employed.

- Each Pushpin must provide for a mechanism for installing, executing, and passing on to its neighbors code and data received over the communication channel.

- Each Pushpin has the ability to sense the world in some capacity and is able to operate on and/or store the resulting data.

1.3.2 Hardware

The Pushpin project embeds a 20 MIPS mixed-signal microcomputer system into the form factor of a bottle cap with the tangible affordances of a thumb tack or pushpin. As the name implies, protruding from the underside of each Pushpin device are a pair of pins of unequal length that can be easily pushed into a layered power plane at arbitrary positions. This novel setup satisfies power and usability requirements (no changing of batteries or rewiring of power connections – simply push the Pushpin into the substrate) and hints at the idea of physically merging sensing and computing networks with their surroundings.

Pushpins themselves are easily modifiable modular devices ideal for prototyping a wide variety of projects requiring sensing, processing, or communicating capabilities. Each Pushpin is comprised of a processing, a communications, a sensing, and a power module (a battery pack can replace the two power pins). Pushpins are equipped with ample analog and digital sensing and communication resources, such as a UART, ADCs, and DACs. Communication between Pushpins is short-range and local by design; each Pushpin belongs to a neighborhood of approximately six other Pushpins over an area on the order 15cm by 15cm. Infrared, capacitive coupling, and serial (RS232) communications modules have been developed.

1.3.3 Software

At the software level, each Pushpin is governed by its own Bertha, a small custom-built operating system charged with overseeing communications and managing mobile process fragments written by the user. A minimal integrated development environment (IDE) and macro language for process fragment authoring is provided as well.

Table 1.1: An analogy between process fragments and gas particles.

Algorithmic	Physical
process fragments	gas particles
Pushpin memory	physical space
program structure	particle interaction modes
Pushpin sensor data	outside forces acting on particles

At the heart of the Pushpin programming model is the process fragment, consisting of up to 2-Kbytes of executable bytecode coupled with up to 256-bytes of persistent state information used and modified by that code. Bertha is capable of concurrently managing approximately a dozen process fragments within each Pushpin. Process fragments call upon Bertha for basic system functions such as pseudo-random number generation, access to analog and digital peripherals, and communication needs. Process fragments can interact locally with other process fragments located in the same Pushpin through a shared memory space (the 'bulletin board system' or 'BBS') local to each Pushpin. They can interact remotely with process fragments located on neighboring Pushpins through a locally mirrored synopsis of each of the neighboring Pushpins' BBSs (the 'neighborhood watch' or 'NW'). Provided sufficient memory and communication bandwidth, process fragments are free to transfer or copy themselves (with the help of Bertha) to neighboring Pushpins, thus allowing for user programs to be diffused into an entire network from a single point of access. Indeed, the Pushpin programming model is intentionally meant to conjure an analogy between an algorithmic system of process fragments and a physical system of gas particles, as illustrated in Table 1.1.

All told, the instant-on nature, easily reconfigurable network and physical topologies, autonomous mobile process software architecture, and modular plug-and-play hardware architecture make Pushpins well suited for quickly configuring and evaluating a wide range of sensor networks varying in density, distribution, sensing modalities, and network characteristics.

Chapter 2

Hardware

"It's alive! It's alive!"

Dr. Frankenstein

This chapter describes the hardware components of the Pushpin Computing platform. Enough technical detail is provided to illustrate the major design decisions and the capabilities and limitations of the resulting implementation. Detailed circuit layouts and schematics can be found in Appendix A.

See Abelson, Knight, and Sussman's *Amorphous Computing Manifesto* [4] for the initial motivation of the Pushpin hardware concept. It gives a concise summary of many of the underlying principles and design decisions governing the Pushpin Computing hardware platform.

Before delving into the details of the Pushpin platform, it may be useful to review the ideological context in which it developed. In an idealized extreme case, the hardware comprising a distributed sensor network consists of sand granule-sized nodes capable of computing, sensing, actuating, communicating with each other, and deriving enough power to function. These nodes would presumably permeate our everyday environment, imbuing commonplace materials such as plywood, cardboard, and paint with the ability to sense, make sense of, and respond to the surrounding environment. References to the hopes for such "smart" materials are readily found in both academic [29] and popular media [30]. In some respects, current technology is not at all far from realizing such a vision. In other respects, a large chasm separates that vision from reality.

That computing elements can now be manufactured small enough to fit this bill is so well-known as to be considered but an obvious result of Moore's Law.[1] More recently, similarly fast-paced advances in micro electro mechanical systems (MEMS) technology hint at the development of sensors and actuators of the desired scale. As for communications, very little exists in terms of such minute hardware. Even less so in terms of hardware communicating over very short distances – as will be argued later, the ideal distance over which to communicate is on the order of the distance between neighboring nodes, presumably only a couple of centimeters at the very most. Nonetheless, the process technology required for manufacturing such small communication technology exists, even if the communication technology itself has yet to be developed.[2] It is not difficult to imagine systems based on radio frequency (RF), optical, or capacitive coupling. More radically, it may be possible to communicate chemically, essentially mimicking the processes used by some living cells (e.g. neurons) to communicate. Power considerations reveal perhaps the most technically challenging aspects of realizing this extreme vision of a distributed sensor network. Many options exist for powering each node, including various parasitic [33], wired, wireless, mechanical, and chemical techniques. But no existing technology can meet the projected energy requirements of each node in the ideal scenario depicted above. This presents itself as an acute technological limit that may be overcome with considerable effort, rather than an impassable fundamental limit.

[1] For example, the ARM9TDMI ARM 32-bit RISC core [31], using a 0.18μm fabrication process, has a die size of 1.1mm^2, power consumption of 0.3mW/MHz, and can run at 220MIPS @ 200MHz.

[2] This scenario may rapidly change over the next 10 years if Intel's recently announced plans for including a radio on every silicon chip pan out [32].

2.1 Design Points

The Pushpin hardware design is predicated on the following set of goals and constraints (in no particular order):

- easily reconfigurable
- usability
- small physical footprint
- low-maintenance
- ample processing power
- ample memory
- omnidirectional short-range communication
- accessible software development
- general-purpose analog and digital peripherals

Clearly, some of the items in the above list oppose one another or could be met in a number of ways. Whatever the final balance was to be, though, it had to conform to Pushpin Computing's single overriding goal – to develop and demonstrate a testbed for studying self-organization as it applies to distributed sensor networks.

2.2 Initial Prototypes

Several proof-of-concept prototypes led up to the present incarnation of the Pushpin Computing hardware platform. This section follows the evolution of the Pushpin hardware from concept to present realization.

2.2.1 Proto-Pushpins

The initial concept, derived from Butera's ERJ (Epoxied Resistor Jungle) [3], meant to include three electrically insulated (except for the very tips) pins of unequal length – one for power, one for ground, and one for communication. The Pushpins would be embedded in a layered substrate, each pin contacting an electrically active layer separated by an intervening insulating layer. The communication pin would contact a resistive layer through which the Pushpin could electrically signal its neighboring Pushpins. See Figure 2-1. A limited realization of this concept can be seen in Figure 2-2, which shows a three-layer substrate with embedded proto-Pushpins. The top layer is a resistive foam, the type commonly used for storing static-sensitive integrated circuits because of its ability to dissipate static charge. The bottom layer is a conductive sheet of silicone rubber mixed with carbon. The middle layer is insulating silicone rubber. Each proto-Pushpin is simply an LED and resistor in series connecting to the resistive layer at one end and the conductive layer at the other end. See Figure 2-3.

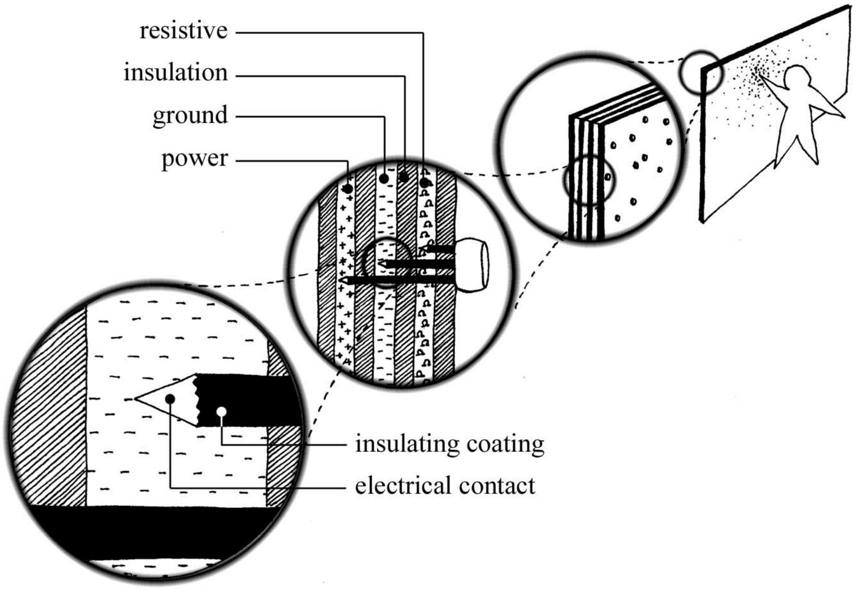


Figure 2-1: Original 3-prong Pushpin concept, including a resistive layer through which to communicate.

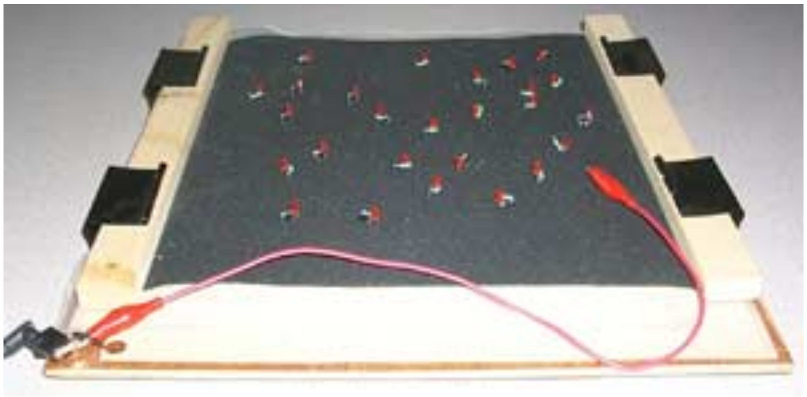


Figure 2-2: Proto-Pushpins embedded in a communication substrate. The red wire, held at 5 volts, acts as a probe for testing the range of a signal through the black resistive foam embedded with proto-Pushpins. The intensity of the LED of a proto-Pushpin indicates the signal strength at that location.

As noted in §1.3.1, communication must be limited to only the spatially proximal neighbors. The purpose of the proto-Pushpins prototype was to demonstrate such local communication through a resistive layer. The brightness of each proto-Pushpin's LED directly corresponds to the voltage of the resistive medium at that point. Thus, given a signal (voltage) passed into the resistive layer at a given location, the brightness of each proto-Pushpin's LED indicates how strong the signal is at the location of that proto-Pushpin. Casual observation of this system clearly indicates that the signal is indeed confined to a local area, the exact size and shape of which is determined by the strength of the signal, the surface resistivity of the resistive layer, the value of the resistor used in the proto-Pushpins, and the density of the proto-Pushpins in the substrate. There are two points to note in this setup. First, it is easily shown that the point-to-point resistance on a resistive surface is highly invariant, regardless of the distance between the points or the size of the surface. Second, that the applied signal remains local is a direct consequence of the neighboring proto-Pushpins, which essentially block the signal from propagating any further. Together, these suggest the setup might be scalable both in physical size and number of proto-Pushpins; the surface in which they are embedded can be nearly any size without affecting its resistive characteristics and the communication radius of each proto-Pushpin will expand or contract according to the

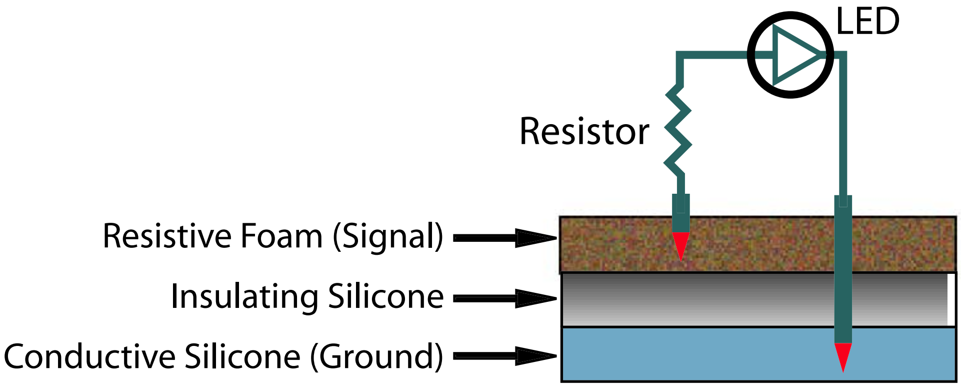


Figure 2-3: Schematic diagram of a proto-Pushpin, comprised of an LED in series with a resistor, embedded in a layered substrate providing a ground plane and a resistive signal plane.

density of its neighbors. Also important to note, however, is that capacitive effects have not yet been fully explored; the large capacitances involved are expected to restrict the possible communication bandwidth.

2.2.2 Resistive Layer Prototype

A minimal hardware implementation of the resistive layer communication scheme depicted in Figure 2-1 was carried out to further test its feasibility. This prototype consists of roughly a dozen Pushpins that can be inserted into a seven-layer silicone rubber substrate (three electrically active layers and four insulating layers). See Figures 2-4 and 2-5. This version of the Pushpin is based on the Microchip PIC 16F84 [34], an 8-MHz microcontroller with 1-Kbyte ROM and 68-bytes RAM. Although minimal communication between two or even three Pushpins through the resistive layer was achieved, subsequent trials with a larger number of Pushpins made it clear that, while possible in theory, this communication scheme will require a considerable amount of research before it can be reliably characterized to the point of deploying as a means of processor-to-processor communication. Although not carried any further in the scope of the Pushpin Computing project, the resistive layer communication scheme remains an intriguing possibility for future work.

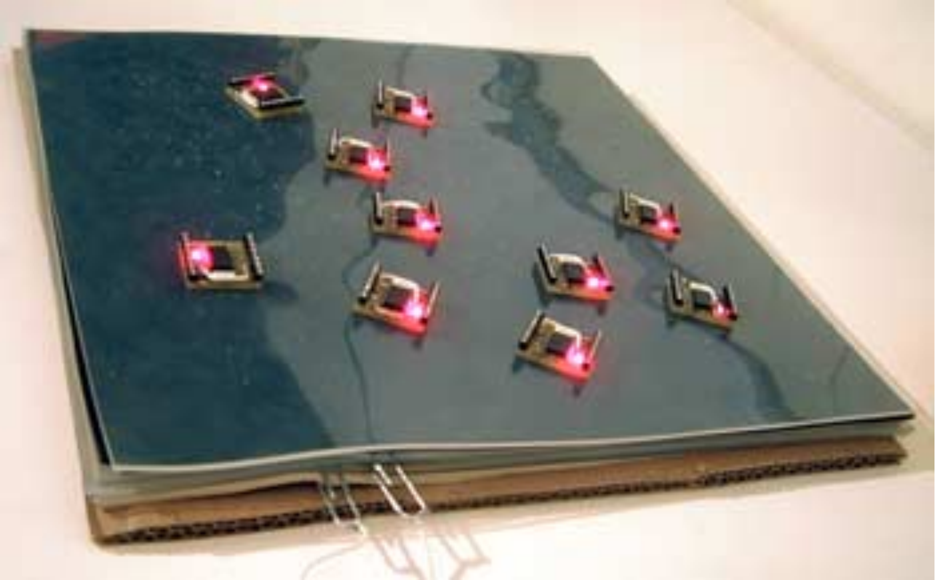


Figure 2-4: Initial PIC-based 3-prong Pushpins embedded in a seven-layer substrate (power layer, ground layer, resistive communication layer, and intervening insulating layers). Each prong electrically contacts a different layer to provide the Pushpin with power and a communication channel.

2.2.3 Media Matrix

In addition to prototypes of communication schemes, a prototype application was built, known as the Media Matrix [35]. The Media Matrix project demonstrates a simple application of distributed networks to the problem of managing a collection of objects, in this case a collection of approximately 30 mini digital video (mini DV) cassettes and their shelving unit. See Figure 2-6. Each mini DV cassette case was outfitted with a tag consisting of a small microprocessor, the hardware needed to communicate wirelessly (via IR) with other nearby tags, and a small light to serve as an indicator to the user. The shelving unit provides power to the tags when the mini DV is shelved. Each of the tags contains information stored on the microprocessor about the contents of the mini DV it is tagged to. For example, a list of keywords describing the contents of the cassette or other meta-data could be stored electronically within the tag. In this way, items in the collection can query their neighbors and gain an understanding of what information is in the immediate vicinity.

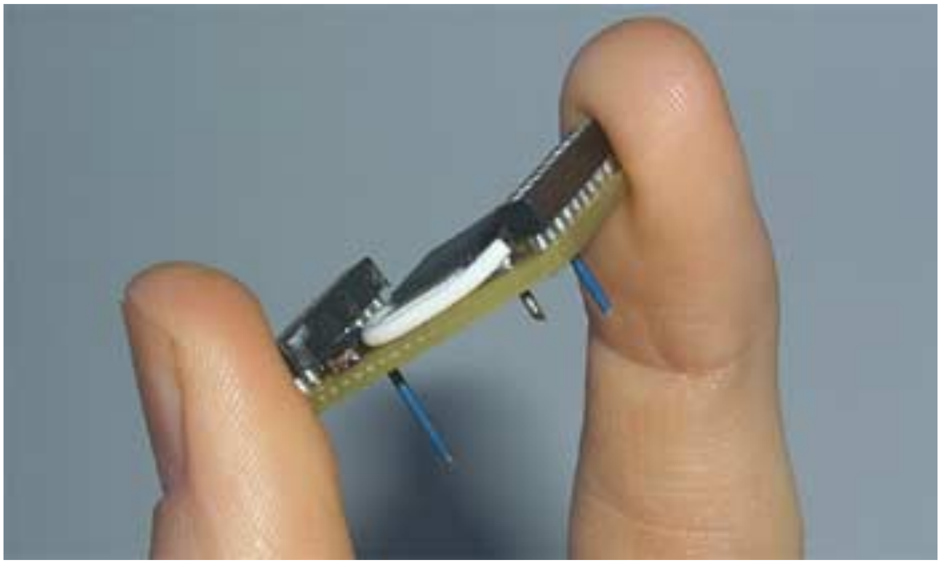


Figure 2-5: Initial PIC-based 3-prong Pushpin. As in the final Pushpin design, a prong passing through more than one electrically active layer of the substrate is coated with insulation everywhere except at the tip to prevent shorts.

This allows for a system that does not require any sorting at all, but rather relies on items in the collection passing the query on to their neighbors until a match is found.

Although the Media Matrix functions as designed, it lacks any ability to sense and is quite limited computationally. The general ideas applied and lessons learned, though, have carried over to the current hardware implementation of the Pushpin Computing project.

2.3 Implementation

The Pushpin hardware implementation that emerged from the various prototypes includes a large laminate panel that provides power and ground (but not communication) to approximately 100 Pushpin devices. See Figure 2-7. The actual implementation can be seen in Figure 2-8.

The Pushpin device design embodies the principles of structural and functional modularity. Each Pushpin is composed of a stacked ensemble of a power module, a communication mod-

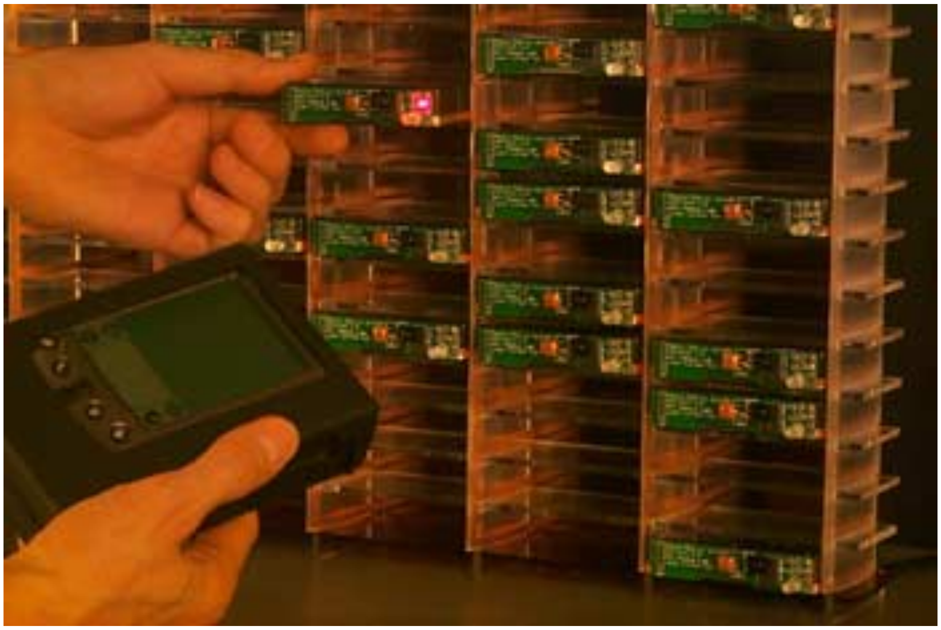


Figure 2-6: Media Matrix physical database. Each mini DV is tagged with a microcontroller capable of communicating with its neighbors, allowing for a user query entered wirelessly from a PDA to be passed from one tag to the next until it is satisfied.

ule, a processing module, and an application-specific expansion module, as demonstrated in Figure 2-9. The logical connections between modules are shown in Figure 2-10. Each module can be separated from the others by hand without the use of special tools. Modules of the same type (e.g. an infrared communication module and a capacitive coupling communication module) can be freely interchanged. The remainder of this chapter is devoted to describing these four modules in some detail. Refer to Appendix A for circuit layouts and schematics.

2.3.1 Power Module and Layered Substrate

The Pushpin moniker derives from the power delivery scheme employed. Protruding from the underside of each Pushpin device are a pair of pins of unequal length that can be easily pushed into a laminate power plane made from two layers of aluminum foil sandwiched between insulating layers of stiff polyurethane foam [36]. This plane (see Figure 2-11) is

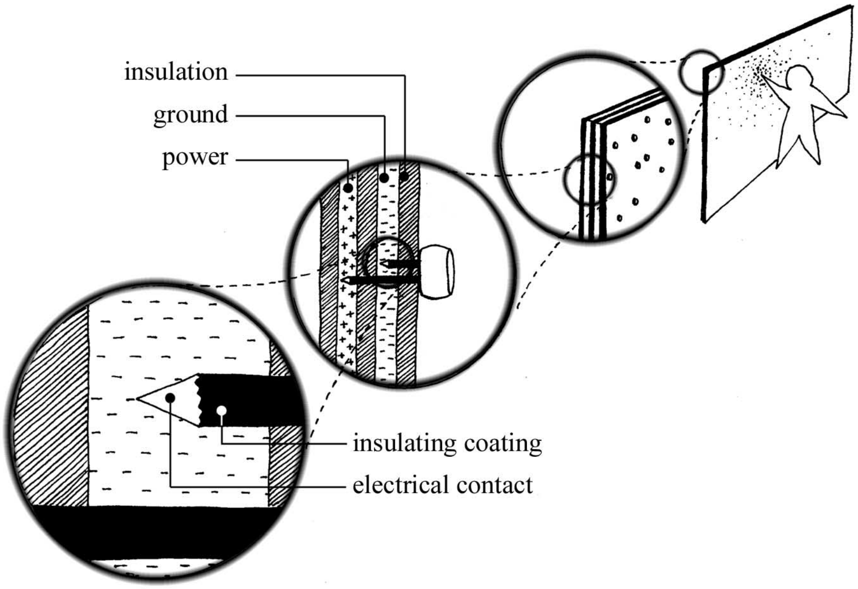


Figure 2-7: Finalized Pushpin concept for user interaction and providing power.

available commercially and is made for arbitrary mounting of small halogen lights. The piece used here measures approximately 125-cm x 125-cm x 2-cm. One of the foil layers provides power and the other ground. This novel setup satisfies power and usability requirements (no changing of batteries or rewiring of power connections, simply push the Pushpin into the substrate to start it up) and hints at the idea of both physically and functionally merging sensing and computing networks with their surroundings. While this solution blatantly sidesteps the important issue of power consumption (the powered substrate is plugged into a power supply), it allows for very quick prototyping and minimal maintenance overhead. Other power sources can easily take the place of the pins and substrate as long as they provide 2.7-VDC to 3.3-VDC.[3]

The total power consumed depends strongly on the particular expansion, processing, and communication modules employed and how they are used. For example, the processing module has several different modes of operation, each requiring a different amount of power.

[3] Two AAA batteries in series is a simple, if bulky alternative.

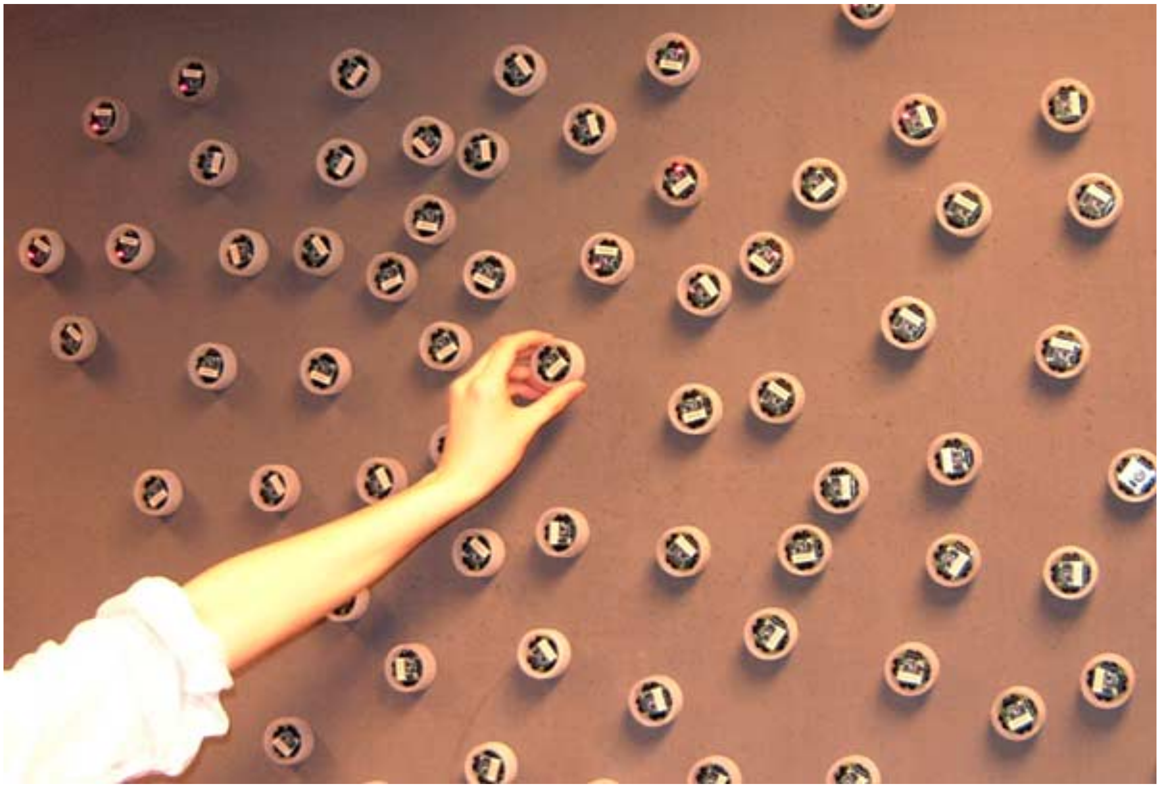


Figure 2-8: Finalized Pushpin implementation, composed of approximately 100 Pushpins and a polyurethane and aluminum foil layered substrate measuring ∼1.25 meters on a side, which provides power. The Pushpins can be arbitrarily positioned on the substrate. Each Pushpin is composed of a two-prong power module, a ∼22-MIPS 8-bit processing module, a 166kbps IR communication module, and light sensing and LED display expansion module. The white ring surrounding each Pushpin acts as an optical diffuser for the infrared signals.

Typical current consumption of the processing module running at 22-MHz with all necessary peripherals enabled is roughly 10-mA, whereas the processing module running in a low-power mode off of an internal 32-kHz clock requires roughly 10-μA. With the clock shutdown, this falls to about 5-μA. If a Pushpin were run off a battery, for example, the lifespan of a power source can here vary from hours to years depending on the particular circumstances. The current 100-Pushpin ensemble draws approximately 1.5-A at 3.3-V.

Layers of conductive silicone rubber were also used as a preliminary power substrate, but the electrical connection to the pins proved quite erratic. Silicone does seem to cure somewhat better than the polyurethane/aluminum substrate, although the polyurethane/aluminum

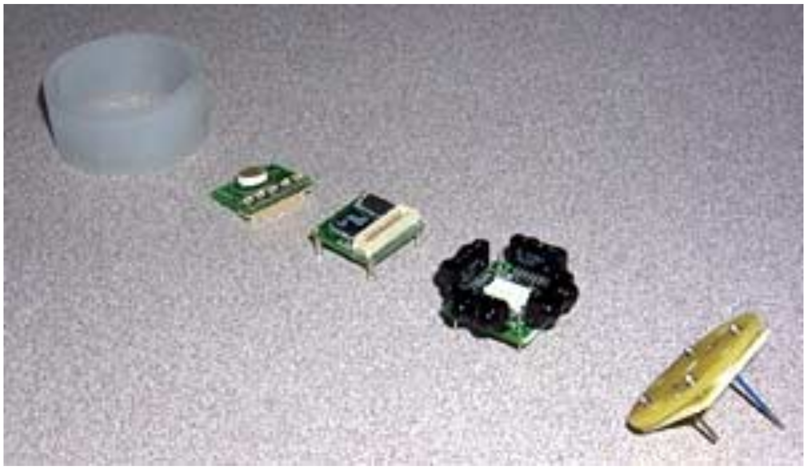


Figure 2-9: A disassembled Pushpin composed of (from upper left to lower right) an IR diffusive collar, a light-sensing and LED display expansion module, a processing module, an IR communication module, and a pronged power module. These modules stack vertically, with the diffusive collar surrounding them.

panel has not yet needed replacement. The actual pins that make electrical contact with the aluminum foil layers are standard 20-gauge wire brads custom coated with an insulating material similar to those used on non-stick cooking pans [37]. The tips of the coated wire brads have been sanded to remove the insulating coating so as to allow electrical contact with the aluminum foil layers of the substrate. The length of the pin determines the particular foil layer it makes contact with. See Figure 2-7.

2.3.2 Communication Module

Anything containing all the necessary hardware for effectively exchanging data with the hardware UART on the processing module qualifies as a communication module. That is, the communication board consists of all communication hardware except the UART itself, which is built into the processor on the processing module. Currently, infrared (IR), capacitive coupling, and RS232 communication modules are available for Pushpins. See Figure 2-12. All three of these modules are capable of running at up to 166kbps, although actual data rates achieved are typically slower due to software concerns (see Chapter 3).

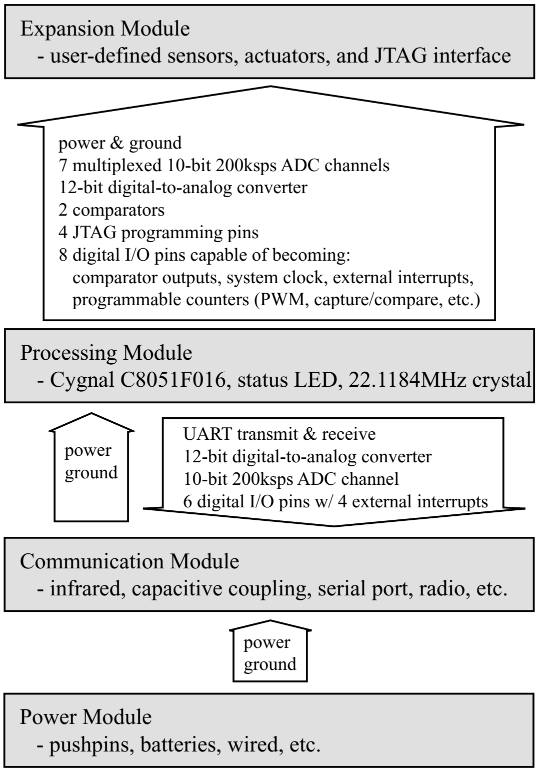

Expansion Module
- user-defined sensors, actuators, and JTAG interface
power & ground
7 multiplexed 10-bit 200ksps ADC channels
12-bit digital-to-analog converter
2 comparators
4 JTAG programming pins
8 digital I/O pins capable of becoming:
comparator outputs, system clock, external interrupts,
programmable counters (PWM, capture/compare, etc.)
Processing Module
- Cygnal C8051F016, status LED, 22.1184MHz crystal
power
ground
UART transmit & receive
12-bit digital-to-analog converter
10-bit 200ksps ADC channel
6 digital I/O pins w/ 4 external interrupts
Communication Module
- infrared, capacitive coupling, serial port, radio, etc.
power
ground
Power Module
- pushpins, batteries, wired, etc.

Figure 2-10: The Pushpin hardware specification. The shaded boxes represent different hardware modules. The arrows represent resources that the module at the tail of the arrow provides to the module at the head of the arrow.

Both the IR and capacitive coupling communication modules are half-duplex and meant to allow Pushpins to communicate amongst themselves, whereas the RS232 communication module is full-duplex and meant to allow a single Pushpin to communicate with a desktop computer through a COM port.

2.3.2.1 IR Communication Module

The IR communication module consists of four multiplexed transceivers, one pointing in each direction of two orthogonal axes. Transmission occurs simultaneously on all four transmitters, but reception can only occur from one receiver at a time. The received signal is shaped by a monostable multivibrator acting as a one-shot before being passed on to the

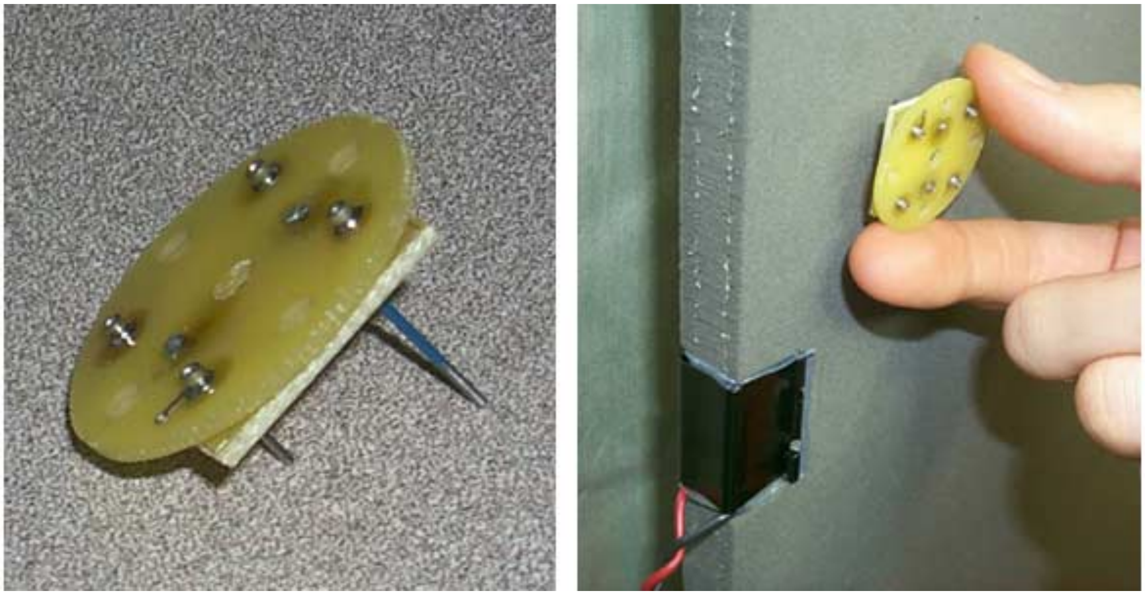


Figure 2-11: Pushpin power module (left) consists simply of two prongs of unequal length, one for power and one for ground. An insulating coating protects the longer power pin from causing a short as it passes through the ground layer to get to the power layer. The layered substrate providing power (right) consists of two layers (power and ground) of aluminum foil separated and surrounded by three layers of polyurethane foam. Power is provided to the substrate by a power supply, of which only the connector is shown in this picture. The layered substrate was donated courtesy of Steelcase, Inc.

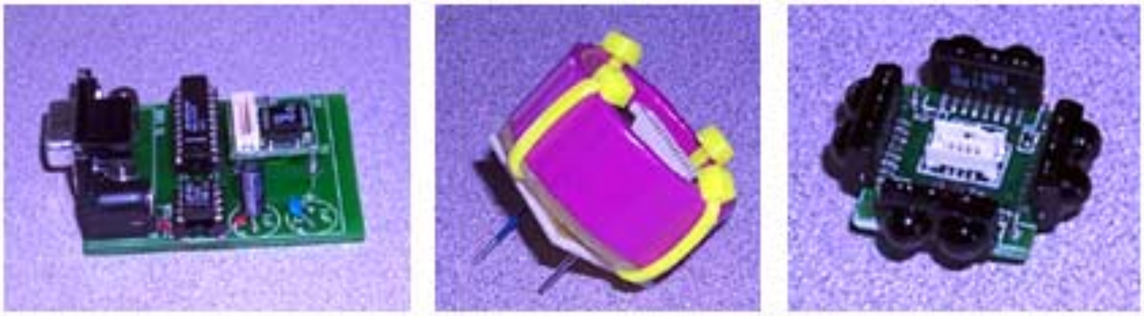


Figure 2-12: RS232 serial communication module with attached processing module (left), capacitive coupling communication module (purple antenna) with attached processing module and power module (center), and IR communication module (right).

UART. The pulse width of the one-shot, and hence the baud rate, is set in hardware by a resistor and capacitor pair. Since only the incoming edge of each IR pulse is detected, it is necessary that each pulse of infrared transmitted be of the same duration, as opposed to longer duration pulses corresponding to more than one bit. In practice, although it not the most bandwidth efficient method, this translates to interleaving a 1 between each bit of data to be sent.[4] Thus, the actual bit rate is half the baud rate.

Although knowledge of the direction a received communication is available to the Pushpin, it is not used; knowing directionality is a violation of the omnidirectionality constraint imposed in §2.1. In this sense, the use of four transceivers is admittedly not ideal. In addition, even with four transceivers and an optically diffusive shield (see Figure 2-13), the communication range remains somewhat anisotropic, ranging from 4 to 15 centimeters depending on the exact configuration of the Pushpins.

2.3.2.2 Capacitive Coupling Communication Module

The capacitive coupling communication module makes use of a single cylindrical antenna (see Figures 2-12 and 2-14) for both transmitting and receiving data. The Pushpin is surrounded by the antenna, but electrically shielded from it by a ground plane. When

[4] See Sklar [38] for comprehensive treatment of channel coding.

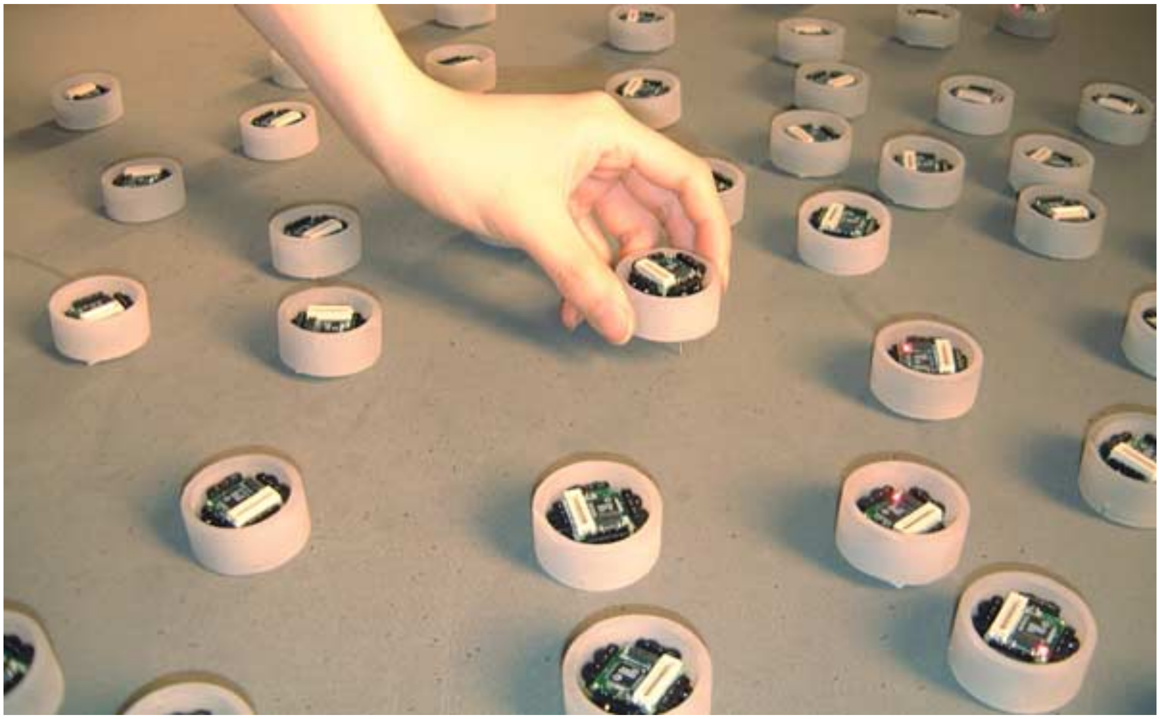


Figure 2-13: Pushpins equipped with IR communication modules. The white collar surrounding each Pushpin is an optically diffusive shield meant to reduce communication range anisotropy.

transmitting, the antenna is connected directly to the transmit pin of the UART. When receiving, the antenna leads into a series of three high-gain inverting amplifiers followed by the same one-shot circuit used in the IR module. A digital-to-analog converter (DAC) provided by the processing module sets the trigger level of the one-shot, allowing for a programmable threshold of reception. This can be used to avoid collisions and noise by listening at a low threshold when trying to determine if the communication channel is free before transmitting and listening at a high threshold when trying to receive data from neighbors. A programmable listening threshold helps minimize the hidden node problem prevalent in many ad-hoc wireless networks. The threshold can be set to receive signals originating from 0 to 10 centimeters away. Furthermore, unlike the IR module, both transmitting and receiving are nearly perfectly omnidirectional. Unfortunately, the capacitive coupling communication module proved essentially unusable due to interference from ambient electrical noise, as it coupled on edges and hence had a broadband response.[5]

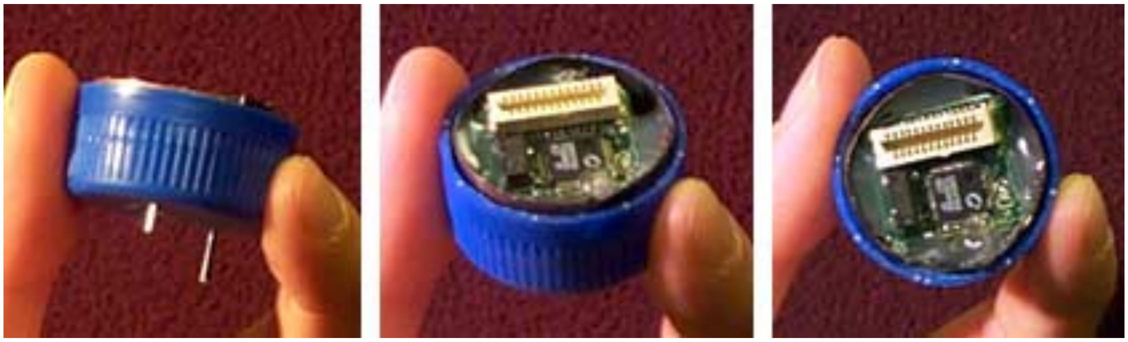


Figure 2-14: An early prototype of the capacitive coupling-based Pushpin was housed in a bottle cap. A copper antenna lines the inside of the bottle cap, encircling the processing module.

2.3.2.3 RS232 Communication Module

The RS232 communication module employs a standard level converter (the MAX233) to allow for communication between a Pushpin and a computer's serial port. It is powered by a 9-VDC power adapter, providing power to the processing and expansion modules as well. The RS232 communication module is primarily meant as an aid in debugging and as a means of loading user code into the operating system. Chapter 3 goes into this last point in more detail.

2.3.3 Processing Module

The processing module essentially defines the core of a Pushpin. See Figure 2-15. The only currently available processing module is designed around the Cygnal C8051F016 – an 8-bit, mixed-signal, 25-MIPS (peak), 8051-core microprocessor [39]. The Cygnal chip is equipped with 2.25-Kbytes of RAM and 32-Kbytes of in-system programmable (ISP) flash memory.

[5] Plans are under way to modify the capacitive coupling design so as to communicate over a carrier, essentially turning it into a small AM radio and hopefully solving the problem of noise. The processing module is capable of generating in hardware a 5.5-MHz square wave, which may be suitable as a carrier.

All hardware supporting the operation of the microprocessor as well as the microprocessor itself is contained on the Pushpin processing module. The microprocessor runs off of a 22.1184-MHz external crystal but also has its own adjustable internal clock for lower power modes. A simple LED indicates the status of the microprocessor. Connectors providing access to the microprocessor's analog and digital peripherals, as detailed schematically in Figure 2-10, comprise the remainder of the processing module.

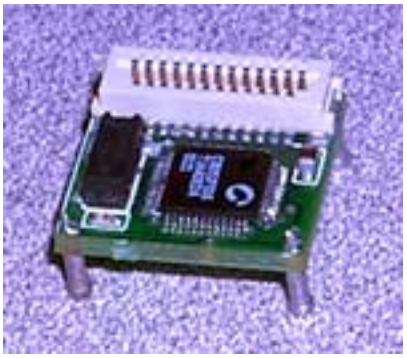


Figure 2-15: Pushpin Processing Module, based on the Cygnal C8051016 8-bit 8051-core microprocessor.

2.3.4 Expansion Module

The expansion module is where most of the user hardware customization takes place for any given Pushpin. The expansion module has access to all the processing module's analog and digital peripherals not devoted to the communication module. This includes general purpose digital I/O, two comparators, seven analog-to-digital converter (ADC) channels, capture/compare counters, and IEEE standard JTAG programming and debugging pins, among others. The expansion module contains application-specific sensors, actuators, and external interrupt sources. Possible examples include sonar transducers, LED displays, microphones, light sensors, and supplementary microcontrollers. Thus far, three types of expansion module have been implemented. The first is a JTAG programming module, which acts as a connector between the Cygnal microprocessor and a serial programming adaptor hooked up to a computer's serial port. This arrangement allows for direct programming of the Cygnal microprocessor. The second is a through-hole prototyping board, which provides access to all the processing module's available analog and digital peripherals. The third is a combination of a five-element LED display and a light sensor comprised of a light-dependent resistor (LDR) as part of a voltage divider read by an ADC channel. Figure 2-16 shows all three expansion modules.

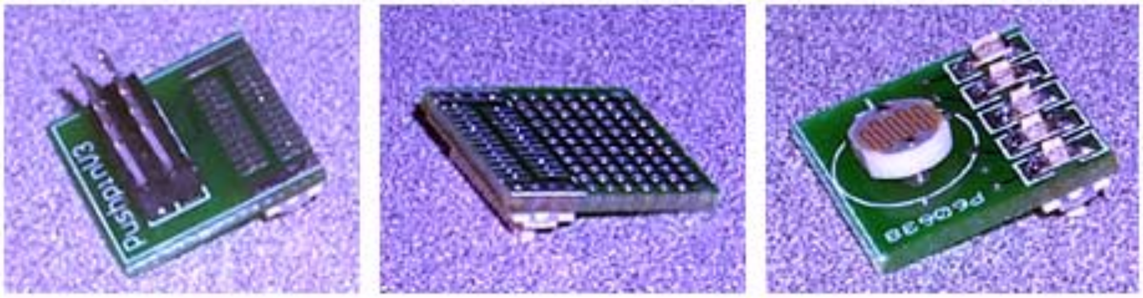


Figure 2-16: JTAG programming expansion module (left), prototyping expansion module, and light sensing and LED display expansion module.

Chapter 3

Software

> *"A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes."*
>
> *Abelson, Sussman, and Sussman in* Structure and Interpretation of Computer Programs *[40]*

The primary goal of the Pushpin software suite is to effect a proper programming model for the type of distributed sensor networks embodied by the Pushpins. The particular programming model implemented here is centered on the concept of algorithmic self-assembly, as laid out in Butera's Paintable Computing work [3]. Pushpin Computing attempts to follow this model as closely as possible. The occasional deviations are due to somewhat limited computational resources and reasons of practicality. A brief introduction to Paintable Computing will clarify some of the core concepts.

3.1 Paintable Computing

Paintable Computing begins with the premise that, from an engineering standpoint, we are not very far away from being able to mix thousands or millions of sand grain-sized computers into a bucket of paint, coat the walls with the resulting computationally enhanced paint, and expect a good portion of the processors to actually function and communicate with their neighbors. The main problem with this scenario, is that we don't yet have a compelling programming model suitable for such a system. Paintable Computing attempts to put forth just such a model, as well as a suite of example applications. To this end, Paintable Computing is a simulation of many (tens of thousands) independent computing nodes pseudo-randomly strewn across a surface. See Figure 3-1. Each Paintable node is capable of communicating omnidirectionally with other nodes located within a limited radius, although no node knows *a priori* anything about its physical location on the surface.

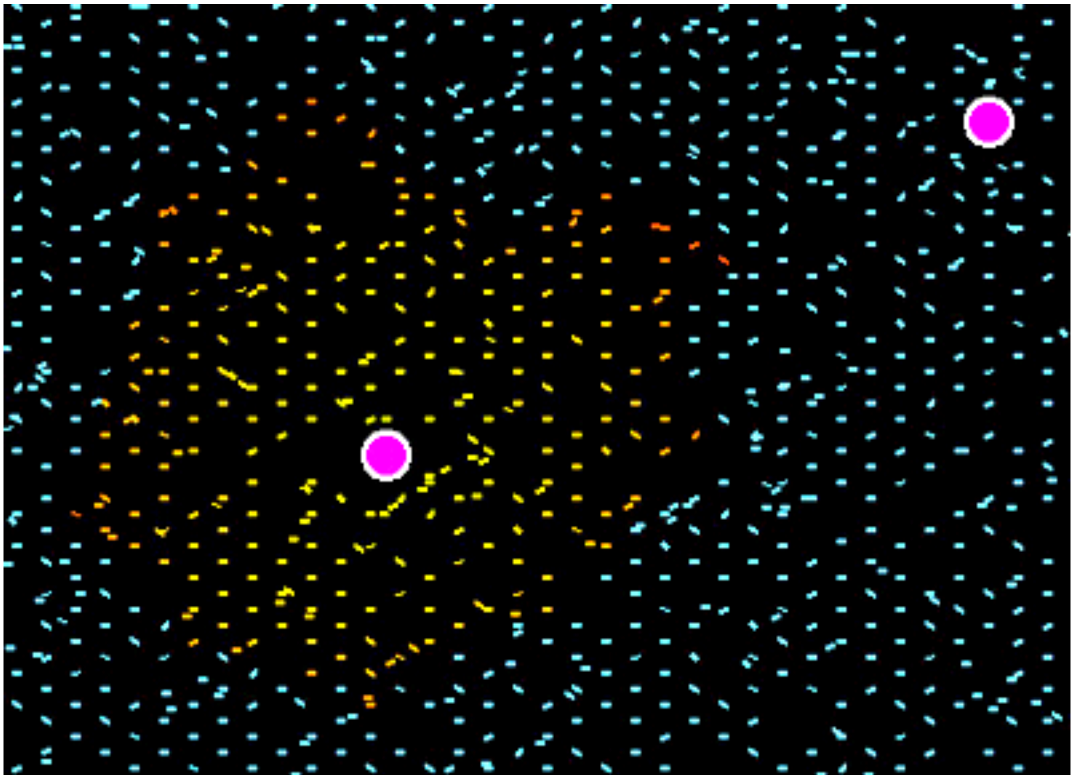


Figure 3-1: A Paintable Computing simulation showing a process fragment diffusing from a central point (large, lower left node). Each colored spec represents a processing node, the color indicating the state of the diffusing process fragment. The warmer the color, the closer to the originating node the process fragment believes it is.

In essence, the programming model employed to organize this proposed architecture is based on *algorithmic self-assembly*; the idea that small algorithmic process fragments exhibiting simple local interactions with other process fragments can result in complex global algorithmic behavior. In a sense, algorithmic self-assembly treats algorithms in the same way thermodynamics treats gas particles [41]; when the number of particles is large, $pV = nRT$ becomes more useful than knowing the position and momentum of each particle. From the seemingly simple, if chaotic system architecture presented above, Paintable Computing demonstrates the utility of algorithmic self-assembly (in the form of process fragments migrating among processing nodes) to build up complex algorithms from simple constituents.

Pushpin Computing is an attempt to bring the results of the Paintable simulations to bear on distributed sensor networks, each Pushpin corresponding to a single Paintable processing node. The Pushpin programming model provides a suite of tools for exploring algorithmic self-assembly as it relates to sensory data extracted from the real world. To this end, an operating system, including a networking protocol, and an integrated development environment (IDE) have been implemented.

3.2 Bertha Design

Management of a Pushpin's resources is vested in that Pushpin's own instance of Bertha - the Pushpin OS. The functions of Bertha can be divided into the following subsystems:

- Process Fragment Management Subsystem
- Bulletin Board System Subsystem
- Neighborhood Watch Subsystem
- Network subsystem

Before discussing the details of design, it is useful to first understand the memory organization of the operating system. This organization is based on the memory structure of a

Component	**Size (in bytes)**
Program Memory (In-System Programmable Flash)	32,896
Internal RAM	256
External RAM	2048

Table 3.1: Cygnal C8051F016 memory

particular variety of Cygnal 8051-core processor, as noted in §2.3.3. Table 3.1 summarizes the Cygnal's memory structure.

Figure 3-2 shows how different subsystems of Bertha are organized on this memory structure. The details will be presented as the various components of the operating system are discussed. All source code pertaining to the Bertha operating system is listed in full in Appendix D.

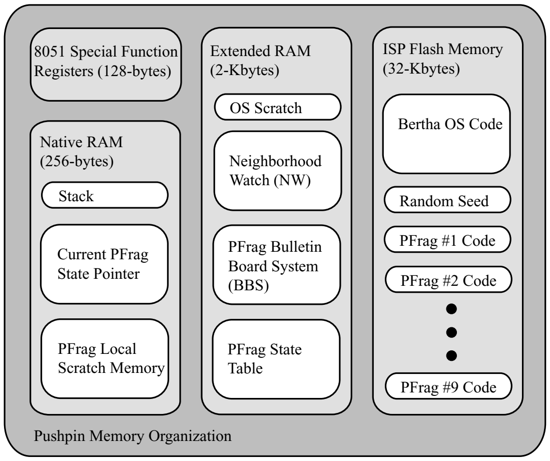

8051 Special Function Registers (128-bytes)
Native RAM (256-bytes)
Stack
Current PFrag State Pointer
PFrag Local Scratch Memory
Extended RAM (2-Kbytes)
OS Scratch
Neighborhood Watch (NW)
PFrag Bulletin Board System (BBS)
PFrag State Table
ISP Flash Memory (32-Kbytes)
Bertha OS Code
Random Seed
PFrag #1 Code
PFrag #2 Code
PFrag #9 Code
Pushpin Memory Organization

Figure 3-2: Pushpin Memory Organization

3.3 Process Fragment Management Subsystem

This subsystem manages all aspects of storing, running, and transferring process fragments (*PFrags*).

Data structure	**Field**	**Size (bytes)**	**Description**
PFrag	Size	2	number of bytes this entire PFrag occupies
	UID	2	hash of this PFrag's byte code, generated at compile time
	CRC	1	cyclic redundancy check of this PFrag, generated at compile time
	State Size	2	number of bytes this PFrag's state variables occupy
	Code	0 - 2041	this PFrag's executable code
PFrag State	Size	2	number of bytes this entire PFrag State occupies
	Local ID	1	identifies the local PFrag to which this PFrag State belongs
	State	0 - 445	persistent variables used by the PFrag to which this PFrag State belongs

Table 3.2: PFrag code and state data structures.

3.3.1 PFrag Memory Management

A Pushpin has enough memory to host up to nine simultaneous PFrags. A PFrag is composed of two data structures - a code component and a state component. These two components are presented in Table 3.2.

3.3.1.1 PFrag Code

Currently, PFrags can be written in C (a subset of ANSI C) using a custom built IDE (described below). As shown in Figure 3-2, the PFrag code is stored in program memory, of which 18-Kbytes is allotted for this purpose. This 18-Kbyte block is divided into nine equal-sized (2-Kbyte) segments. Bertha assigns one of these segments to every incoming PFrag, irrespective of its size (with a pre-specified maximum PFrag size of 2-Kbytes). If a Pushpin already has nine PFrags on hand, any incoming stream of bytes from a PFrag migration is ignored.

The PFrags are sandboxed within their assigned 2-Kbyte space using an 8051-core feature - the processor supports a special form of call and jump instructions named ACALL and AJUMP. These instructions are used with a special address mode, in which only the lower order 11 bits of the 16-bit program counter are modified when a jump or call is executed. This effectively limits a PFrag's address space to 2-Kbytes. As a result, the entire PFrag must fit within 2-Kbytes. This 2-Kbyte address limit serves two important purposes. First, the 2-Kbyte AJMP and ACALL instructions make it difficult for PFrags to accidentally access memory that doesn't belong to them.[1] Second, PFrags can be executed on Pushpins without any special purpose address translation. All 16 bits of the program counter are modified while switching from one PFrag to another, but the most significant five bits of the program counter remain constant during execution of any particular PFrag. Consider an example - a piece of PFrag code has a jump instruction from address 0x000A to 0x00AB. To execute this jump, only bits 0, 5 and 7 need to be modified. Assuming the PFrag starts at address 0x4000, the program counter at the beginning of the jump will read 0x400A. After the jump is made, it will read 0x40AB. As long as each PFrag starts on a 2-Kbyte boundary, this type of relative addressing will be valid.

PFrag local variables (not persistent state) are stored in the internal RAM. Each PFrag can have a maximum of 158 bytes of local variables.

3.3.1.2 PFrag State

State is information that a PFrag wants to maintain across executions and devices. For example, a PFrag might need to keep a count of the number of Pushpins it has visited. Each PFrag can have a maximum of 445 bytes of state. The state information of all local PFrags is stored in a contiguous block - locations 128-575 of the external RAM. There are two reasons for storing state in external RAM separate from the code. First, since the state might be rewritten during execution, storing it on a medium that has fast write access speeds is important. The write access speed of RAM is much higher than that of

[1] Although intentional misuse of pointers is still possible.

flash memory. Second, the flash memory on which code resides has a maximum limit on the number of rewrite/erase cycles (Cygnal guarantees for at least 10,000 cycles, although 100,000 cycles is typical) whereas RAM does not have any such limitation.

3.3.2 PFrag Execution

PFrags execute on Bertha by means of a well-defined PFrag interface, a set of Bertha system calls, and an execution schedule.

3.3.2.1 PFrag Interface

In order for Bertha to interact with and execute a PFrag, the PFrag must define the following three methods: `install()`, `update()`, and `deinstall()`. Other methods can be added at the user's discretion, but they will go unused unless called by one of the three previously mentioned methods.

Bertha cannot actually call any PFrag methods directly, since the addresses of the PFrag methods are not known in advance and could be positioned anywhere within a PFrag's code. To facilitate calls between Bertha and PFrags, the IDE introduces during the compilation process a `PFragEntry()` method into every PFrag. This method is set by the compiler to reside at a fixed location within the Pfrag (currently the 8^{th} byte of the PFrag code). The signature of this method is `unsigned int PFragEntry(unsigned char methodID, unsigned int arg0, unsigned int arg1)`, where a `char` is one byte and an `int` is two bytes. Bertha invokes the remaining PFrag interface methods indirectly through the `PFragEntry()` method. The `PFragEntry()` method then dispatches calls to the specified method by correctly interpreting the `methodID` parameter and invoking the corresponding method with arguments arg0 and arg1. This indirect invocation is possible because `PFragEntry()` is always at a known memory location within the PFrag code and has knowledge of the locations of the other three methods at PFrag compile time. Those three method interfaces are described below. In all cases, the arguments and return values are currently unspecified and left available for future use.

- `unsigned int install(unsigned int arg0, unsigned int arg1)` - All the code related to PFrag initialization is defined here. This is the first method Bertha calls when a PFrag enters a Pushpin. It is called only once by Bertha, although it may be called again by code within the PFrag.

- `unsigned int update(unsigned int arg0, unsigned int arg1)` - This method is called to inform the Pfrag that it should start a round of execution. It is important to note that this method may be called many times - once during every round of execution.

- `unsigned int deinstall(unsigned int arg0, unsigned int arg1)` - Depending on several variables, Bertha may call this method prior to deleting the PFrag, but no guarantee is made that it will do so. This allows the PFrag to prepare for its demise. For example, it may want to migrate to another Pushpin or notify other PFrags.

3.3.2.2 System Calls

Bertha provides a suite of services to PFrags through system calls. These calls can be used to read from and write to the BBS and NW, request transfer, influence execution, and access analog and digital peripherals. The PFrags access the system calls through the System Call Table located in code memory (addresses 30720 - 31231). The table holds up to 256 2-byte pointers to system calls. In order to use this table, each process fragment must know both the location of the table within memory and the organization of the function pointers within the table. Both of these pieces of information are included in the PFrag by the IDE during the PFrag compilation in the form of an absolute memory address and an enumeration of available system calls. Bertha provides a number of system calls that can be used by PFrags to control how they are executed by Bertha. For example:

- `void die()` - The PFrags call this method to have themselves removed from the host Pushpin. When this method is called, Bertha deletes the calling PFrag and all its associated information such as BBS posts, state, etc.

A full listing of the system calls Bertha provides to PFrags is given in Appendix B.

3.3.2.3 Execution Schedule

As in any other uniprocessor system, only one process (PFrag or system) can be active at anytime. The OS executes all the resident PFrags in a round-robin fashion by calling sequentially calling their `update()` methods. Each PFrag's `update()` method runs to completion and therefore defines the temporal granularity of the time sharing.

Although not yet implemented, Bertha could start a watchdog timer to limit the time a PFrag can take to execute its update method so as to avoid any PFrag from monopolizing all the CPU time. Between executing one PFrag and the next, the OS performs various system functions such as synchronizing with neighbors and validating PFrags. The order in which a PFrag gets executed is determined by its position in memory. Since Bertha randomly assigns each incoming PFrag a segment in memory, no PFrag can make any assumptions about its position in the execution queue.

3.3.3 PFrag Transfers

PFrags migrate amongst Pushpins to complete their tasks. Since PFrags are designed to be self-contained within a 2-Kbyte memory space, and all the relevant elements of the operating system are placed at standard locations, it is easy for PFrags to move from one Pushpin to another. PFrags can request Bertha to transfer them to neighboring Pushpins by calling a Bertha system function:

- `unsigned char requestTransfer(unsigned char neighborID)` - Request a transfer to the neighboring Pushpin whose local neighborhood ID is passed as the argument. An argument of 0 is read as the 'global address' and a transfer to all neighbors is attempted. Returns 1 if the transfer has been initiated by the local Bertha, 0 otherwise. Does not guarantee the success of the transfer.

Since a Pushpin can communicate only with its neighbors, PFrag transfers from that Pushpin can only be made to immediate neighbors. PFrags know about their neighbors by examining a local mirror (held in the Neighborhood Watch) of a synopsis of each neighbor's BBS. As the local NW gets updated after an attempted transfer, a PFrag can watch for signs that the transfer was successful; i.e. a post might appear in a neighbor's BBS indicating the PFrag was properly installed. Thus, although there is no explicit negotiation or acknowledgment of PFrag transfer, the possibility exists for implicit negotiation and acknowledgment to be carried out by the PFrags themselves.

3.3.4 PFrag Lifecycle

PFrags go through different states while executing on a Pushpin. The state transitions of a PFrag on a Pushpin are given in Figure 3-3. The `install()` and `deinstall()` methods are executed only once per PFrag per visit to a Pushpin. However, the `update()` method may be called during every round of execution. It is important to note that these transitions may not be immediate and the PFrags may have to wait in one state before transitioning to another.

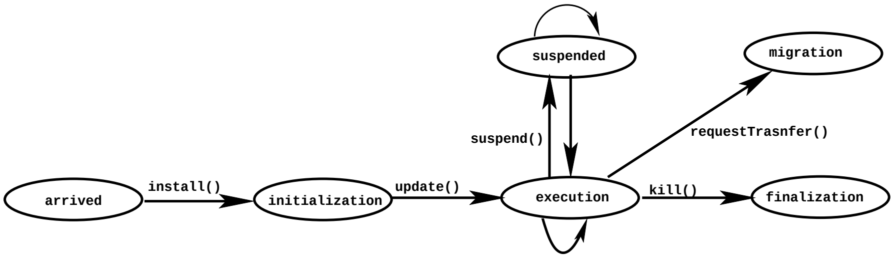


Figure 3-3: PFrag state transition diagram.

3.4 Bulletin Board System Subsystem

As previously mentioned, PFrags on the same device communicate among themselves by means of the Bulletin Board System. PFrags can post to and read other PFrags' posts

Data structure	Field	Size (bytes)	Description
BBS Post	Size	2	number of bytes this entire BBS post occupies
	Local ID	1	indicates which local PFrag made this BBS post
	UID	2	byte code hash of the PFrag that made this BBS post, generated at compile time
	Post ID	1	an ID generated by the posting PFrag at the time the post is made
	Content	0 - 570	arbitrary data decided upon by the posting PFrag

Table 3.3: BBS Post data structure.

from the BBS. As shown in Figure 3-2, 576 bytes of external RAM are allotted for BBS use. Bertha maintains the BBS as a linked list of posts. Each post is composed of multiple fields, as shown in Table 3.3.

All PFrag access to the BBS is arbitrated by Bertha via a set of system calls, leading to two main advantages. First, since PFrags are not responsible for low-level details such as memory management, it becomes easier to author PFrags and the PFrags themselves are lighter weight. Second, Bertha has complete control over the BBS without depending on the correctness of PFrag code. The following system calls are provided to PFrags for interacting with the BBS:

- `unsigned char postToBBS (unsigned char postId, unsigned int length, unsigned char data * post)` - This method writes a single post in the BBS. The post is rejected if it is larger than the BBS can accommodate. The argument `postId` is the post ID assigned to the post. The argument `length` is the number of bytes in the post. The argument `post` is a pointer to the contents of the post. The `data` specifier indicates the pointer points to an address in internal RAM. Returns 1 if successfully posted, 0 otherwise.

- `unsigned char removePost(unsigned char postId)` - This function deletes a single post from the BBS. The first post in the BBS with postID matching the argument and originally posted by the active PFrag is removed. Returns 1 if the post was successfully removed, 0 otherwise.

- `unsigned char removeAllPosts()` - Removes all the entries posted by this PFrag. Returns 1 if any posts were removed, 0 otherwise.

- `unsigned char getBBSPostCount()` - Returns the number of posts in the BBS.

- `unsigned int getBBSPost(unsigned char postId, unsigned int length, unsigned char data * post)` - Copies up to `length` number of bytes from the BBS post whose post ID matches `postID` to the address `post`. Returns 0 if the post does not exist, 1 otherwise.

- `unsigned int getNthBBSPost(unsigned char n, unsigned int length, unsigned char data * post)` - Copies up to `length` number of bytes from the `n`th BBS post to the address `post`. Returns 0 if the post does not exist, 1 otherwise.

3.5 Neighborhood Watch Subsystem

Locally mirroring synopses of the BBSs of neighboring Pushpins informs both Bertha and resident PFrags of the status of their neighbors. Among other things, and as previously mentioned, this allows PFrags to autonomously keep track of their own migration status between Pushpins. However, due to the asynchronous nature of operations in this environment, the Neighborhood Watch status information might be stale by the time it reaches the PFrag.

As shown in Figure 3-2, 896 bytes of external RAM are allocated for storing the synopses of BBSs that belong to neighboring Pushpins. At the end of every round-robin PFrag cycle, Bertha transmits to all neighboring Pushpins all BBS posts. Each neighbor independently determines, based on available memory resources, how much of that transmitted BBS will be stored locally. Since newer posts are posted further down the BBS linked list than older

Data structure	**Field**	**Size** (bytes)	**Description**
Neighbor	Size	2	number of bytes this entire Neighbor occupies
	Neighbor ID	1	indicates which neighboring Pushpin this Neighbor represents
	Posts	0 - 893	subset of the posts in the BBS of the neighboring Pushpin corresponding to this Neighbor

Table 3.4: Data structure of an entry in the Neighborhood Watch.

posts, the older a post is, the more likely it will be included in the mirrored synopses on neighboring Pushpins. In any case, there is no guarantee that a post will ever make it to any neighboring Pushpins – as with many aspects of Pushpin Computing, inclusion in the synopsis is probabilistic. Each entry in the Neighborhood Watch contains the fields shown in Table 3.4; each NW entry is essentially a BBS with a small amount of additional information specifying the neighbor of origin.

The consistency guarantee of mirrors is rather weak in the sense that two synopses of the same BBS on two different Pushpins may not be the same. Since the devices operate asynchronously and communicate over a noisy medium (e.g. IR), achieving perfect coherence of synopses would be very expensive. Even if communication and timing were error free and synchronous, there is still no guarantee of finding identical synopses on all neighboring Pushpins, since each of the neighbors decides for itself how much of the transmitted synopsis to keep as a local mirror. Despite this real-world memory constraint, this manner of managing the Neighborhood Watch has the advantage of simplicity; it provides PFrags with complete information about some of the neighboring posts, as opposed to providing incomplete information about all neighboring posts, which would require a more complex protocol before complete information could be obtained.

Bertha provides system calls very similar to those listed in §3.4 for reading posts from synopses stored in the NW, the difference being that a neighbor ID argument is added to each of the system calls to specify which synopsis to read from. The corresponding writing system calls are, of course, not provided. A complete listing of system calls available for

PFrag use is given in Appendix B.

3.6 Network Subsystem

The network subsystem manages all communication between Pushpins. A Pushpin's neighborhood is defined as the collection of all Pushpins it can 'hear'. Ideally, neighborhood inclusion would be a reflexive property, but communication irregularities make it such that *A is able to hear B* does not imply *B is able to hear A*. In any case, it is up to each Pushpin to choose an 8-bit neighborhood ID that is unique to both its neighborhood and to the neighborhoods of each of its neighboring Pushpin's. This ID selection process is mathematically identical to a variation of the graph theoretic map coloring problem [42]. The fact that an 8-bit number is supposed to be unique in this scenario implicitly limits the number of neighbors a Pushpin is expected to have. ID assignment is through a combination of random guessing, listening to network traffic, and abiding by vetoes from neighbors. A 64-bit pseudo-random number is used to identify the Pushpin until an 8-bit number can be decided upon, so as to minimize confusion should a duplicate 8-bit ID be chosen initially. An 8-bit number is used for the ID (as opposed to always using a 64-bit number) so as to reduce network load, since each packet sent to neighbors includes the sender's neighborhood ID.

Note that there is no explicit packet forwarding; each device can communicate directly only with its immediate neighbors. Of course, if need be, a PFrag can be written to explicitly forward packets, but this is a choice the programmer must consciously make. Aside from being simpler to implement, excluding packet forwarding is meant to encourage algorithms based on local, not global, interaction.

The network subsystem consists of three layers - physical, data link, and network.

3.6.1 Physical Layer

The Cygnal processor used in the Pushpin's processing module contains a standard full-duplex hardware UART (Universal Asynchronous Receiver/Transmitter) with 1-byte buffering for both receive and transmit and interrupt generation upon completion of receiving or transmitting a full byte. Although the UART is capable of much faster baud rates, the fact that some data are written to flash memory directly from the UART requires that the UART run at 92160 baud so as to allow the relatively slow flash memory write process to complete. Also, although the UART is full-duplex, it is typically only used in half-duplex mode. Both transmit and receive functionality is completely interrupt driven.

While in theory many modes of communication could be used with the Pushpin processor module, currently only the IR communication module is used. See §2.3.2 for the details of this module. Also, recall that the actual bit rate for the IR communication module is half the baud rate. The RS232 communication module is used when a single Pushpin needs to communicate with a PC running the Pushpin IDE. The Bertha OS automatically detects during start-up which communication module is being used.

3.6.2 Data Link Layer

This layer is responsible for medium access control and reliable transmission of bytes. Since the UART defines a start bit and stop bit for transmitting bytes, Bertha does not explicitly handle framing. An 8-bit cyclic redundancy check (CRC) code, based on the polynomial $x^8+x^2+x^1+1$, serves as an error detection scheme.[2] Lookup tables are used to implement the CRC algorithm. Since the placement of devices can be arbitrary and the architecture is decentralized and distributed, these devices face hidden node and exposed node problems. To reduce collisions, a simplified version of the MACAW (Multiple Access with Collision Avoidance for Wireless LANs) protocol [43] is employed. In essence, Pushpins listen before they speak to make sure no neighbors are talking, and run a simple backoff algorithm if they are.

[2] See Sklar [38] for a comprehensive introduction to error detection.

Data structure	Field	Size (bytes)	Description
Network Packet	To Address	1	sending Pushpin's neighborhood ID
	From Address	1	intended recipient's neighborhood ID
	Type	1	kind of content included in this packet
	Content Size	2	number of bytes of content being sent
	CRC	1	8-bit error checking mechanism
	Content	0 - 2048	arbitrary contiguous block of data held in memory

Table 3.5: Network packet structure.

3.6.3 Network Layer

The OS uses this layer to transmit packets between Pushpins. The packet structure is shown in Table 3.5.

The network layer does not buffer data. This is true of both transmitting and receiving. Thus, all data to be sent as the packet's content must lie in a contiguous block of memory. Since the largest piece of content that Bertha would want to transfer is the code portion of a PFrag, the 2-Kbyte content size limitation is large enough to alleviate the need of ever breaking any content up into multiple packets. However, the constraint that all content must be contiguous means that the PFrag code and state must be sent as separate packets.

The Type field can take on one of the following values:

- 0 - Ten-byte general purpose message.
- 1 - Synopsis for inclusion in the NW.
- 2 - PFrag code.
- 3 - PFrag state.
- 4 - New random seed.

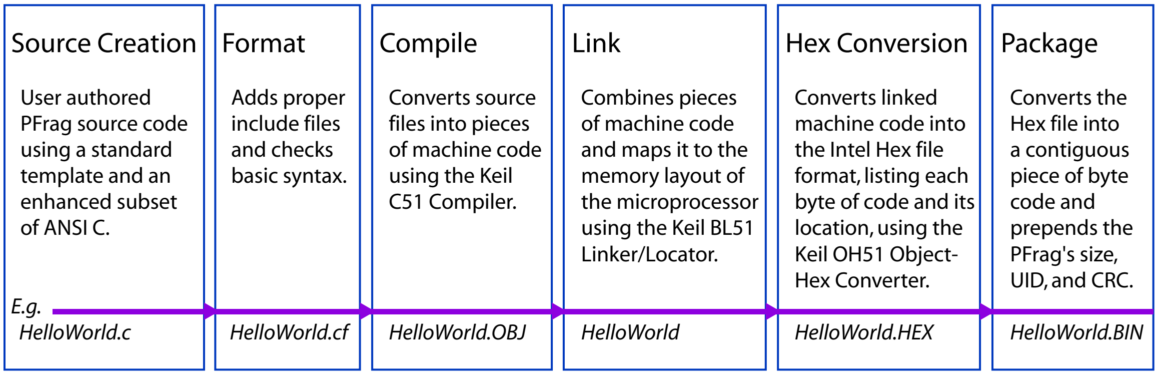
- 5 - Neighborhood building message.

The ten-byte general purpose message can be used as way for the user to control and debug the an ensemble of Pushpins. For example, PFrags can be erased and the time granularity of the main Bertha loop can be set by sending the appropriate ten-byte message to the Pushpin. The utility of each of the other message types should be clear from descriptions of the various parts of the OS. The only peculiarity to note is the packet type for a random seed. Each Pushpin's random seed is a hefty 128 bytes stored in flash memory. Certainly, there very little practical need exists for a random seed this large for use by the Pushpins. The random seed's abnormal size is a relic of the physical characteristics of the flash memory – 128 bytes is the smallest unit of flash memory that can be erased at one time.

3.7 Pushpin IDE

Users create custom process fragments using the Pushpin integrated development environment (IDE), a Java program implemented primarily by MIT undergraduate researcher Michael Broxton, that runs on a desktop computer. Figure 3-4 depicts the process the IDE goes through to create a PFrag. Process fragment source code is authored within the IDE using ready-made code templates, a subset of ANSI C supplemented by the system functions provided to process fragments by Bertha, preprocessor macro substitutions, and IDE pre-formatting. The IDE coordinates the formatting of source code, compilation of source code into object files, linking of object files, and transmission of complete process fragments over a serial port to an expectant Pushpin with Bertha installed and running. The IDE also enforces the process fragment structure requirements outlined in §3.3.2.1. Packaged with the IDE is a packet debugging tool for sending and receiving single packets, as defined in Table 3.5, to and from a single Pushpin.

Currently, the Pushpin IDE calls upon a free evaluation version of the Keil C51 compiler and Keil BL51 linker [44] to compile and link process fragments. Bertha is initially installed on a Pushpin by way of an IEEE standard JTAG interface. Note that Bertha need not be


Source Creation

User authored PFrag source code using a standard template and an enhanced subset of ANSI C.

Format

Adds proper include files and checks basic syntax.

Compile

Converts source files into pieces of machine code using the Keil C51 Compiler.

Link

Combines pieces of machine code and maps it to the memory layout of the microprocessor using the Keil BL51 Linker/Locator.

Hex Conversion

Converts linked machine code into the Intel Hex file format, listing each byte of code and its location, using the Keil OH51 Object-Hex Converter.

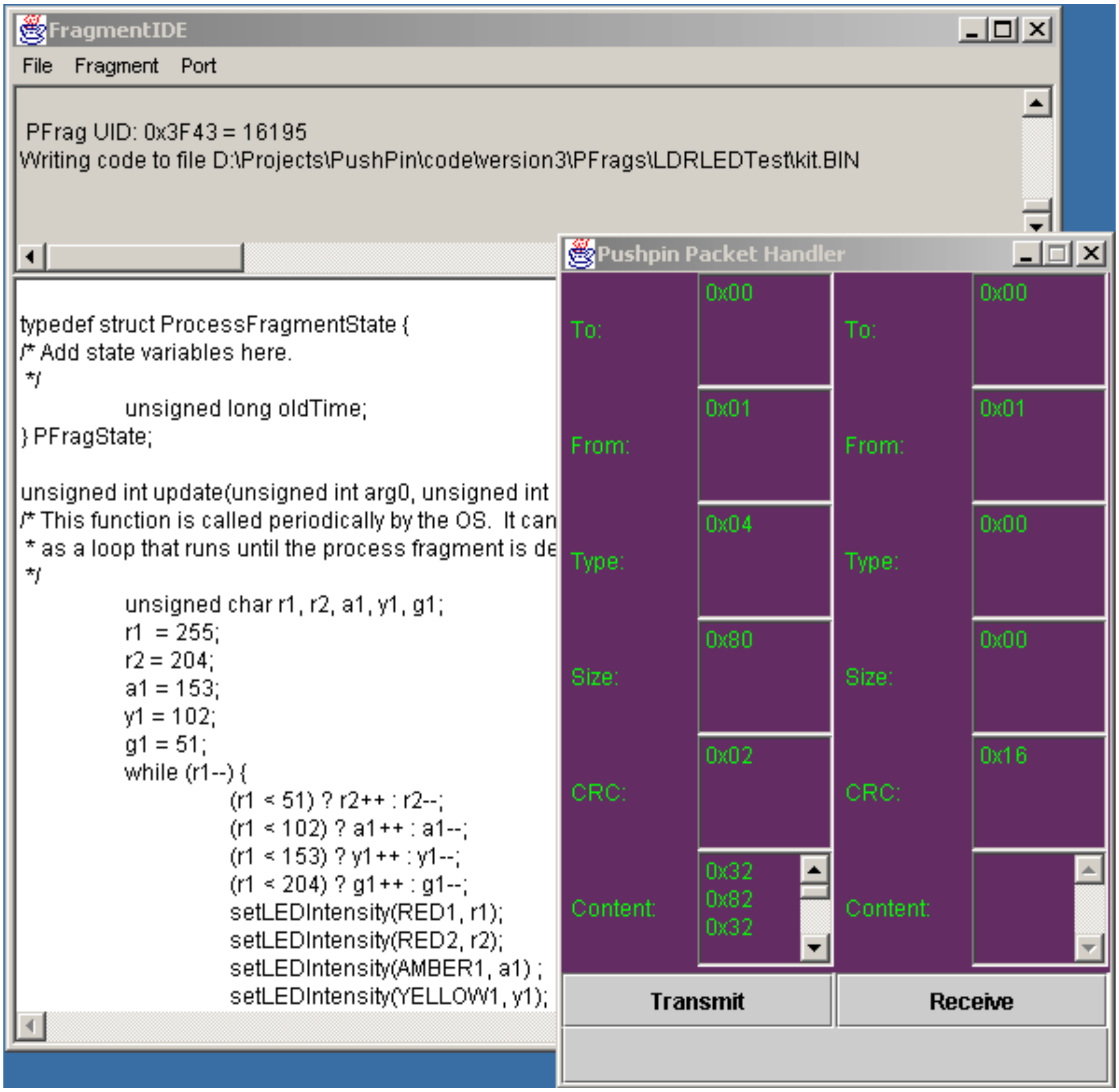
Package

Converts the Hex file into a contiguous piece of byte code and prepends the PFrag's size, UID, and CRC.

Figure 3-4: The logical sequence of operations the Pushpin IDE follows when creating a PFrag from user-specified source code. The user only authors the source code with the aid of a template; all subsequent steps are internal to the IDE and are initiated by a single menu command.

compiled with any specific knowledge of the process fragments to be used; arbitrary process fragments can be introduced to Pushpins during runtime.

Of course, Pushpins can be programmed directly as a regular 8051-core microprocessor without using either Bertha or the Pushpin IDE. One of the many advantages of Bertha and the Pushpin IDE, however, is that the details of the antiquated Intel 8051 architecture are hidden from the user.



```
 PFrag UID: 0x3F43 = 16195
Writing code to file D:\Projects\PushPin\code\version3\PFrags\LDRLEDTest\kit.BIN
```

```
typedef struct ProcessFragmentState {
/* Add state variables here.
 */
		unsigned long oldTime;
} PFragState;

unsigned int update(unsigned int arg0, unsigned int
/* This function is called periodically by the OS.  It can
 * as a loop that runs until the process fragment is de
 */
		unsigned char r1, r2, a1, y1, g1;
		r1  = 255;
		r2 = 204;
		a1 = 153;
		y1 = 102;
		g1 = 51;
		while (r1--) {
			(r1 < 51) ? r2++ : r2--;
			(r1 < 102) ? a1++ : a1--;
			(r1 < 153) ? y1++ : y1--;
			(r1 < 204) ? g1++ : g1--;
			setLEDIntensity(RED1, r1);
			setLEDIntensity(RED2, r2);
			setLEDIntensity(AMBER1, a1) ;
			setLEDIntensity(YELLOW1, y1);
```

To:	0x00	To:	0x00
From:	0x01	From:	0x01
Type:	0x04	Type:	0x00
Size:	0x80	Size:	0x00
CRC:	0x02	CRC:	0x16
Content:	0x32 0x82 0x32	Content:	

Figure 3-5: A screenshot of the Pushpin IDE. The purple window in the foreground is the packet debugger. The background window is a combination of a status display and PFrag code text editor.

3.8 Taking Stock

The definitions have been set, the innards of the hardware exposed, and the workings of the software laid bare, but what does it all add up to? This thesis began by stating that it lies at the meeting point of sensing and self-organization. This being the case, essentially two things have thus far been achieved. First, we have a platform for building distributed sensing networks that explicitly emphasizes the central role self-organization must play. This is, apparently, the first such platform of its kind. Furthermore, with the notable exception of the SmartDust motes and their TinyOS operating system [45], it is one of the only flexible platforms available for short-range distributed sensing networks in general. Second, it is one of the few attempts to bring simulation of self-organization into the real world of hardware.

Even casual inspection of the programming model reveals a similarity between PFrags and what are known as mobile agents. Researchers have invested a lot of effort in developing mobile agents, which has led to the development of several mobile agent systems, such as Telescript [46], Aglets [47], and Hive [48]. A comprehensive survey of mobile agent technologies can be found in the collection edited by Milojicic et al [49]. All the work in this area focuses on applying agents technology to data mining, electronic commerce and other web related applications. Apparently, the mobile agents paradigm has never been applied explicitly to distributed sensor networks.

That said, it is equally important to point out what Pushpin Computing is not. For example, although the programming model and system architecture are new, Bertha borrows many well-known algorithms and concepts found in conventional operating systems. As Tanenbaum [50] says, in the field of operating systems, ontogeny recapitulates phylogeny. That is, each new species (mainframe, minicomputer, personal computer, embedded computer, smart card, etc) seems to go through the same development as its ancestors did. Moreover, those aspects of Pushpin Computing that are new are almost certainly far from optimal. This can be extended further to say that the Pushpin Computing platform was assembled from many disparate pieces with the goal of making a functioning whole, without much regard for the refinement or optimization of any of those pieces. This perhaps will fall under the category of future work or exercises left to the reader.

Most importantly, up until this point, nothing has been said of applications using Pushpin Computing or of how self-organization quantitatively plays a role. The former is the subject of the next chapter. The latter may be the subject of the next thesis.

Chapter 4

Applications

"Try not. Do, or do not. There is no try."

Master Yoda

This chapter focuses on applications using the Pushpin Computing platform as described in the previous chapters. To begin, the Diffusion PFrag provides a concise example of the creation and distribution of a PFrag. Two other elementary process fragments, the Gradient PFrag and the Retina PFrag, are introduced as well. With this as a basis, the implementation of a shape recognition algorithm currently under development will be detailed. Finally, various collaborations with other research groups will be briefly discussed. Overall system performance and performance of each PFrag are detailed in the next chapter.

4.1 Diffusion Process Fragment

The Diffusion PFrag provides a clean illustration as well as preliminary benchmarks of the Pushpin platform. Intuitively, the Diffusion PFrag does no more than replicate itself on every Pushpin it encounters until the entire ensemble is infected. Algorithmically, the Diffusion PFrag can be summarized as follows:

1. Upon entry into a Pushpin, post a message to the BBS stating that this Pushpin is infected.

2. Fade the LED display from red to green and back again to indicate to the user that this PFrag is still active.

3. Check the Neighborhood Watch for any uninfected neighbors.

4. Copy this PFrag over to the first neighbor found to be uninfected.

5. Repeat steps 2-4 each time this PFrag is updated.

6. Turn off all elements of the LED display upon deletion of this PFrag.

The Diffusion PFrag is perhaps the simplest example of a PFrag that takes advantage of the core features of the Bertha operating system – the Bulletin Board System, Neighborhood Watch, and PFrag management and transfer functionality. Source code for the Diffusion PFrag, as it would appear in the Pushpin IDE, is listed in Appendix C.1. Figure 4-1 depicts the Diffusion PFrag running on an ensemble of Pushpins.

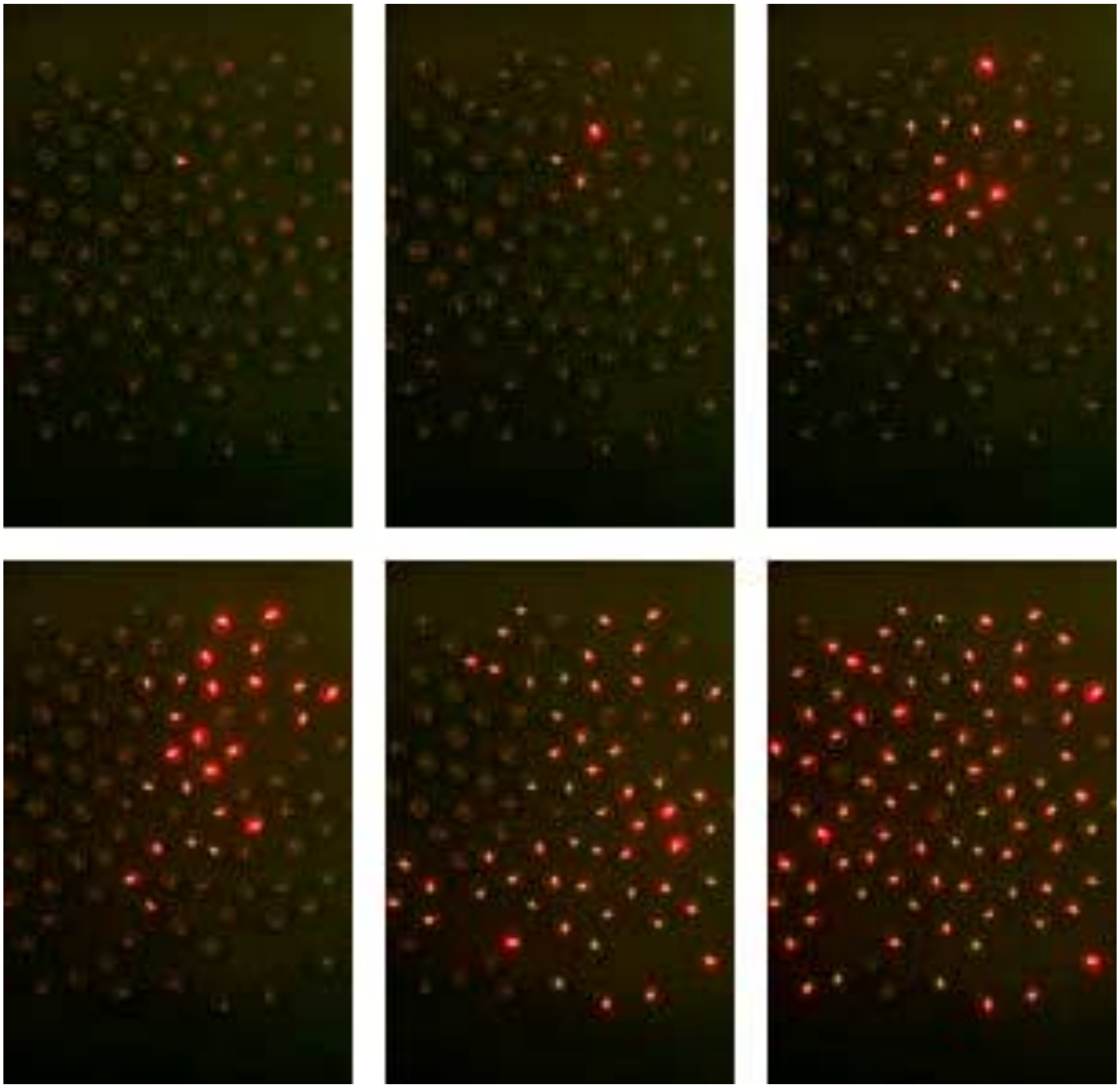


Figure 4-1: Time-lapsed images of a Diffusion PFrag propagating through a network of approximately 100 Pushpins. From left to right and top to bottom, the images show the replication and spreading of an initial single Diffusion PFrag inserted near the center of the ensemble. Dimly lit Pushpins contain no PFrags, brightly lit Pushpins contain a Diffusion PFrag – they are 'infected.' Some of the uninfected Pushpins are either not correctly receiving power from the substrate or have suffered an operating system failure. The PFrag code running on these Pushpins is identical to that listed in Appendix C.1.

4.2 Gradient Process Fragment

The Gradient PFrag builds upon the Diffusion PFrag by adding a sense of distance from an initial Pushpin of origin. That is, the PFrag keeps a running tally of the minimum number of hops between Pushpins necessary to travel between the origin and the Pushpin on which the PFrag is located. Algorithmically, the Gradient PFrag can be summarized as follows:

1. Upon entry into a Pushpin, if there is another Gradient PFrag already installed, then delete this PFrag.
2. Upon entry into a Pushpin, if any of the neighboring Pushpins contain a Gradient PFrag, set this PFrag's hops from the origin to 255. Otherwise this Pushpin must itself be the origin, so set the hops from the origin to zero. Post the number of hops from the origin to the BBS.
3. Wait a short amount of time to allow the states of neighboring Pushpins to equilibrate.
4. Compare the hops from the origin of all neighbors. For each neighbor, if that neighbor's hops from the origin is less than this PFrag's hops from the origin, set this PFrag's hops from the origin to the neighbor's hops from the origin plus one. If no neighbors' BBSs contain a post indicating hops from the origin and this PFrag is itself not at the origin, then deinstall this PFrag, as the origin no longer exists.
5. Copy this PFrag to the first neighboring Pushpin found to not already have a copy.
6. Update the LED display to indicate the number of hops from the origin. Red indicates the origin and those Pushpins one hop from the origin. Amber indicates two hops from the origin, yellow three hops, and green four hops.
7. Repeat steps 4-6 each time this PFrag is updated.
8. Turn off all elements of the LED display upon deletion of this PFrag.

The Gradient PFrag, aside from its potential usefulness building more complex algorithms as already demonstrated in the Paintable Computing simulations, also provides a concrete example of the analogy drawn in Table 1.1. Namely, just as gas particles maintain a global equilibrium of pressure, volume, and temperature by means of local interactions, Gradient PFrags maintain a global equilibrium of distance from the origin by constantly checking the states of their neighbors. The equilibrium is disturbed when, for example, the origin Pushpin is removed, causing a sort of phase transition. Source code for the Gradient PFrag, as it would appear in the Pushpin IDE, is listed in Appendix C.2.

4.3 Retina Process Fragment

The Retina PFrag, as the name implies, transforms an ensemble of Pushpins equipped with light sensors, as detailed in §2.3.4, into a very primitive retina capable of distinguishing light, dark, and the boundary between the two. Additionally, the Retina PFrag mimics the behavior of the Diffusion PFrag to spread itself among all Pushpins. Algorithmically, the Retina PFrag can be summarized as follows:

1. Upon entry into a Pushpin, query the light sensor for a baseline calibration value to be stored as part of this PFrag's persistent state. Record a brightness reading equal to the baseline reading. In addition, turn on the amber LED to indicate that this Pushpin contains a Retina PFrag.

2. Update the LED display to reflect the current state of the Pushpin. If the Pushpin is being exposed to light (relative to the initial baseline calibration of the light sensor) then turn on the red LED. Otherwise turn off the red LED.

3. Remove any previous BBS post indicating the state of this Pushpin's light exposure and replace it with an updated version.

4. Check the Neighborhood Watch for any uninfected neighbors. Copy this PFrag over to the first neighbor found to be uninfected.

5. Check the light exposures of neighboring Pushpins to determine if this Pushpin is on a boundary between light and dark. If this Pushpin is not being exposed to light and

at least one of its neighbors is being exposed to light, then turn on the green LED. Otherwise, turn off the green LED.

6. Repeat steps 2-5 each time this PFrag is updated.

7. Turn off all elements of the LED display upon deletion of this PFrag.

The Retina PFrag hints at a more complex PFrag for differentiating the shape of a particular light pattern. This will be discussed in the next section. Source code for the Retina PFrag, as it would appear in the Pushpin IDE, is listed in Appendix C.3. Figure 4-2 depicts the experimental setup for testing the Retina PFrag.

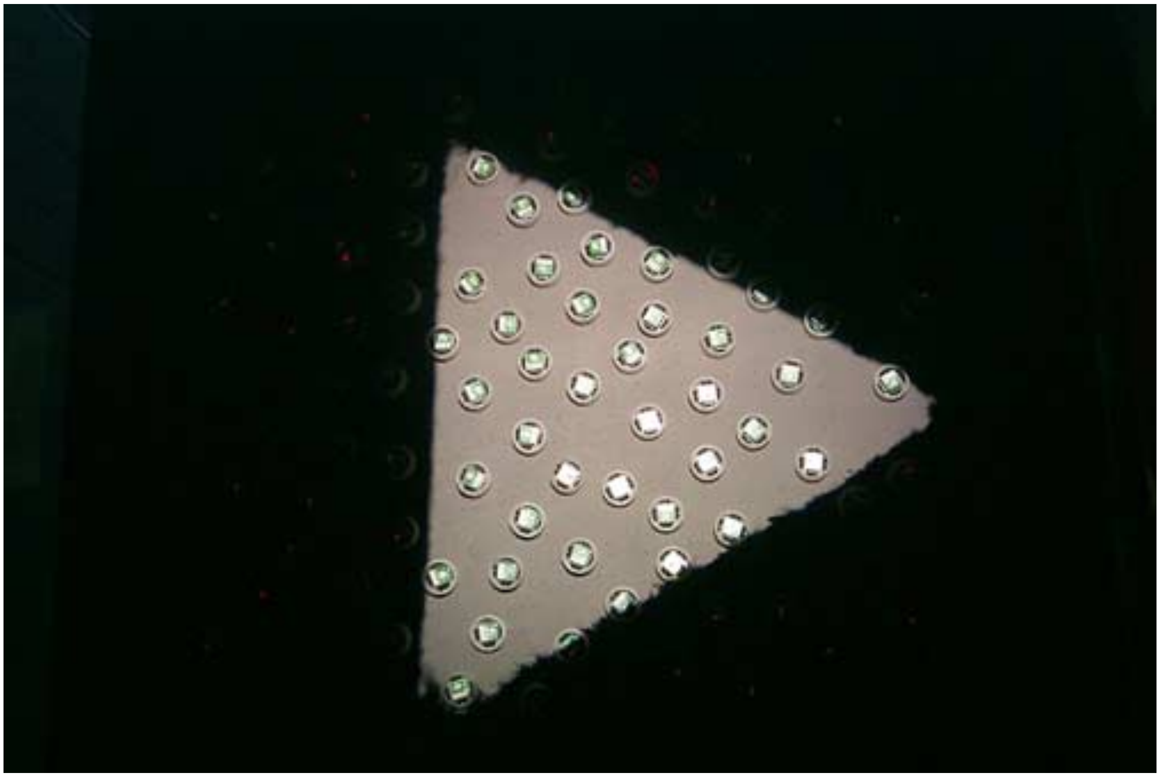


Figure 4-2: A slide projector containing an opaque slide of with the appropriate shape cut out casts light in the form of that shape onto the populated Pushpin substrate. Each Pushpin is here equipped with an expansion module consisting of a light sensor and a five-element LED display, as described in §2.3.4. This setup is used for testing the Retina PFrag and developing the Shape Recognition PFrag.

4.4 Shape Recognition Process Fragment

The application originally planned for first demonstrating the Pushpin platform is still under development at the time of this writing. The goal here is to get the ensemble to differentiate between a number of simple geometric patterns of light projected one at a time onto the Pushpins, as depicted in Figure 4-2. Currently, only the circle, square, and triangle patterns are being considered. This is an admittedly contrived scenario, but it is nonetheless complex enough to demonstrate the applicability of the Pushpin platform. Essentially three methods of distinguishing between these shapes were considered. They are listed below from the most general to the least general:

1. Build up a coordinate system and determine the shape using knowledge of basic geometry and the location of all Pushpins with light sensor readings above a certain threshold.

2. Compare the ratio of the total number of Pushpins that both detect light and have neighbors that do not detect light to the total number of Pushpins that detect light. This approach attempts to approximate a calculation of the shape's perimeter-to-area ratio, which is enough to distinguish it from other shapes in this scenario.

3. Determine the number of Pushpins that classify themselves as being in a corner of the illuminated shape. As above, this is accomplished by comparing sensor values with neighboring Pushpins.

By far the simplest of the three, and the approach adopted here, is the third method. The specific algorithm used was developed in collaboration with William Butera and tested using his Paintable Computing simulator modified to conform to the computational constraints (e.g. bandwidth and memory size) and physical constraints (e.g. node number and density) faced by the Pushpins themselves. The algorithm makes use of a single process fragment, summarized at a high level as follows:

1. Upon entry to a Pushpin, propagate a copy of this PFrag to all neighboring Pushpins not already populated.

2. If the light sensor indicates this Pushpin is in the dark, do nothing. Otherwise, carry out the remaining steps below.

3. Calculate the percentage of this Pushpin's neighbors with light sensors indicating those neighbors are in the light.

4. Based on this percentage, propagate to all other Pushpins in the lit region a guess as to whether this Pushpin is in a corner. The lower the percentage, the more likely this Pushpin is in a corner.

5. Count the number of Pushpins that believe they are corners and use this value to determine the shape of the lit region.

6. Set this Pushpin's LED display to reflect the shape of the region this Pushpin is believed to be a part of.

Note that the above algorithm has only been fully implemented in simulation and references to Pushpins are actually references to simulated Pushpins. The source code for this simulated PFrag is listed in Appendix C.4. Although initial results in simulation are very promising, there nonetheless remain several real-world challenges to overcome before porting this algorithm to actual Pushpins. Results of this simulation and the other three PFrags already mentioned will be overviewed in the next chapter.

4.5 Collaborations

In parallel with the development of the Pushpin Computing platform, several individuals and research groups have taken an interest in Pushpins. Mentioned below are those groups that have significantly incorporated Pushpins into their research one way or another.

- Aggelos Bletsas and Dean Christakos, doctoral candidates with the MIT Media Lab's Media and Networks Group, have contributed resources toward building approximately 20 IR-enabled Pushpins, which they have used in conjunction with a novel air hockey table arrangement to study dynamic network behavior. Furthermore, they have used the Pushpin hardware platform with their own software to demonstrate time synchronization within a network.

- Jeremy Silber, a recently graduated master's student also with the Media and Networks Group, used IR-enabled Pushpins and his own software to develop and demonstrate a "Cooperative Communication Protocol for Wireless Ad-hoc Networks," as documented in his master's of engineering thesis by the same name [51].

- Professor Peter Fisher of the MIT Physics Department is currently heading up an effort to use both the hardware and software components of the Pushpin platform to perform real-time processing of raw data collected from a type of multi-wire drift chamber common in high-energy particle physics experiments. This collaboration is expected to be ongoing.

- William Butera, Research Scientist affiliated with the MIT Media Lab and the recently formed MIT Center for Bits and Atoms (CBA), is using the Pushpin platform, along with his own simulation research and industry experience, as a guide for developing the next generation of hardware based on the Pushpin Computing and Paintable Computing specifications.

Chapter 5

Discussion & Evaluation

> *"Results! Why man, I have gotten a lot of results. I know several thousand things that won't work."*
>
> *Thomas Alva Edison*

This chapter serves as a broad wrap-up. The following sections detail and comment upon the observed results of the applications listed in the previous chapter, provide an evaluation of the design decisions made in chapters 2 and 3, give a final comparison of this work to select related works, and outline the possible future evolution of this work.

5.1 General Bertha OS Performance

Simply running the process fragments listed in chapter 4 on the Bertha operating system described in chapter 3 gives a glimpse of some basic performance characteristics. This does not take the place of rigorously defined and executed benchmark testing, which was not carried out in this work. Nonetheless, reviewing preliminary results provides a first-order approximation to general performance and serves as a gross guide to future development.

In isolation, a single Pushpin running the Bertha operating system meets design expectations and performs accordingly. However, with an increasing number of neighbors or, to be more precise, with increasing communication between neighbors a degradation in performance occurs. In particular, a Pushpin becomes increasingly more prone to long bouts of unresponsiveness with increasing communication activity. Additionally, bandwidth among Pushpins drops precariously with increasing communication activity among neighbors. These problems are particularly apparent when running the diffusion application described in §4.1; although the system does perform as indicated in Figure 4-1, it takes on the order of five minutes for an initial PFrag to replicate itself and propagate throughout the entire ensemble. Preliminary debugging indicates these are two separate problems that happen to compound each other's severity.

The first bug is due to the operating system falling into a pseudo-infinite loop ('pseudo' because there are special cases where the OS can break out of the loop given a particular interrupt event). This is caused by a corruption of local variables in one or more of the fundamental functions used to manipulate the most prevalent operating system data structure, the linked list. Presumably, the data becomes corrupted when the communication system (triggered by an interrupt) calls one of the fundamental functions in the middle of the main operating system process calling the same function. Although, the compiler used to build the operating system provides a recourse for avoiding such reentry problems, it is not yet known if that recourse was properly implemented in the operating system.

At least some portion of the decrease in bandwidth is due to the unresponsiveness just mentioned. However, by observing that a relatively small number all Pushpins fall into the pseudo-infinite loop described is evidence that other issues play a significant role as well. The relatively simple channel arbitration scheme employed certainly could be improved upon. That the bandwidth decreases with an increasing amount of information to be transmitted (e.g. with increasing size of the PFrags to be transferred) implies that errors due to collision play an important role. Although those errors are most likely detected (using the 8-bit cyclic redundancy check), there is no attempt made to correct those errors. These collision errors, in turn, are most likely due to hidden and exposed node issues. [1] Perhaps a more suitable, if more complicated scheme, would be one of the many variations of time division multiple access (TDMA) coupled with some form of error correction algorithm.

One way to minimize the above problems is to limit the number of PFrags allowed in each Pushpin, which effectively lowers the communication needs of each Pushpin. In any case, there is no fundamental reason why both problems can't be fixed given time to sufficiently analyze them.

5.2 Evaluation of Hardware Design

Regarding hardware, the design decisions made for the processing and expansion modules proved fortunate. In particular, the Cygnal 8051-core mixed-signal microprocessor and its associated development tools exceeded expectations and are well-suited for quick development of a multi-purpose platform such as the Pushpins. Also worthy of note, is the durability of the Conan connectors by Berg [52].

The power module and layered power substrate performed well for a system of this size, but could be improved on. Although the power substrate does not heal completely from being punctured by pins, it did not need to be replaced in the course of moderate use. The many design iterations of the two-pronged power module resulted in a mechanically sturdy base capable of being pushed into and extracted from the power substrate without breaking and while maintaining an electrical connection. The insulation coating the longer of the two power prongs stood up to much abuse as well. Approximately 5% of Pushpins stuck into the power substrate do not make a proper electrical connection the first time. This is an acceptable amount, but could be improved upon. Furthermore, previously stable electrical connections occasionally fail with time due to warping or loosening of the material surrounding the power prongs.

[1] A hidden node scenario is when A can communicate with B and B can communicate with C, but A cannot communicate with C. An exposed node scenario is when a node can transmit to all its neighbors, but cannot receive from any of its neighbors. These scenarios often result in nearby nodes attempting to transmit simultaneously, resulting in typically indecipherable collision.

The communications modules stand out as the hardware most in need of improvement. The RS232 communication module should incorporate another mode of communication, such as IR, so that it can not only act a link between a PC and a single Pushpin, but also a link between a PC and the entire Pushpin ensemble. That is, it should be able to use IR (or another appropriate medium) to broadcast to its neighbors messages received from the PC over the RS232 line. The current version of the IR communication module should be simplified to use only one transmitter and one receiver, rather than four of each. Since most IR transceivers are designed to be directional, this might require some custom hardware modifications, as exemplified in Rob Poor's Nami project [53], in which a reflective cone is used to disperse/collect infrared light. Omnidirectionality within the plane also needs improvement, despite already employing a diffuser. As mentioned in §2.3.2.2, the current version of the capacitive coupling module needs to be modified to work over a carrier frequency, thereby reducing noise sensitivity.

5.3 Evaluation of Software Design

In many ways, the version of the Bertha OS presented here is the result of following the path of least resistance to achieving the functionality described in chapter 3. That is, much of the functionality embodied in Bertha gets the job done, but can be arrived at in a more efficient manner. The round-robin scheme for executing PFrags, the assignment of analog and digital peripherals, and the method for controlling time granularity are all examples of functionality that might be better implemented in some other way. The two exceptions to this are the communication subsystem and the underlying linked list data structure subsystem. Both these subsystems are perhaps too complex in the name of efficiency and elegance for their own good, maybe even leading to the problems listed above. If anything, the contrast between the overly complex and overly simplistic components of the operating system points out the need for another layer of abstraction, a virtual machine, which will be discussed shortly. Overall, though, the general operating system architecture works well given the limited memory and processing resources available. That these resources are stretched to their limit can be taken as testament that the OS is well-suited to them.

The Pushpin IDE proved to be an invaluable tool in debugging and generally managing an ensemble of Pushpins and their resident process fragments. The user controls and receives quantitative feedback from the Pushpin platform primarily through the IDE. Thus, the IDE is important enough that as much effort should go into improving it as any other aspect of the Pushpin platform.

5.4 More Related Work

Given the current state of this work, at least four related projects are worth revisiting. On a visual level, the Diffusion PFrag described in §4.1 evokes a strong recollection of Rob Poor's Nami project, which was meant to serve as "an effective visualization of self-organizing networks [53]." The Pushpin platform takes some inspiration from this, although the ultimate purpose of the Pushpins is somewhat more ambitious, as reflected in the correspondingly more complex machinations that lay under the hood.

As previously mentioned, Kwin Kramer's Tiles project is "a child-scale platform for exploring issues related to networks, communication and computational process [25]." The goals of the Tiles project and the Pushpin project differ considerably; the former is concerned more with epistemology, the latter more with the theory and application of self-assembly as it relates to sensing. These ideological differences manifest themselves in many ways throughout both platforms. That said, it is perhaps more interesting to look at their similarities, which are surprisingly numerous. Both platforms emphasize expandability, mobile code, and ease of use. Of particular note is the conclusion that a virtual machine is needed. This will be discussed in the next section.

Sharing much of the same background as the Tiles project is the currently active Crickets project [54]. Based on the PIC microcontroller [34], a Cricket is a programmable brick (a relative of LEGO Mindstorms [55]) with modest sensing capabilities designed to be the nucleus for robotics and *in situ* data collection. Crickets are programmed in a variant of LOGO, an interpreted programming language designed for ease of use by non-experts. The

Crickets project essentially embodies a more evolved version of the Tiles and is under active development.

A primary motivation of the Pushpins was to implement in hardware the Paintable Computing [3] programming model and test aspects of its feasibility. In hindsight, some of the assumptions made in the Paintable Computing simulator should be looked at more carefully, especially those regarding neighbor-to-neighbor communication. The communication model used in the simulator holds that the radius of communication is time-invariant, perfectly circular, unaffected by occluding neighbors, and identical for all particles. All of these assumptions proved to be unrealistic in the case of the Pushpins. This result, although somewhat expected, should prompt careful inspection of the algorithms implemented on the Paintable simulator for any absolute reliance on the assumptions made about the communication radius. No insurmountable obstacle is foreseen to arise from this dependence, although it may require a more robust and complex implementation of the simulator algorithms.

In addition, the Paintable simulator assumes all communication is error-free. Given the state-of-the-art in error detection and correction, this is not an unreasonable assumption to make in many cases. But, no communication scheme will ever be 100% error-free and without building in some amount of error into the simulation, it is all too easy to design an algorithm that relies either explicitly or implicitly on 100% accuracy in communication. This is especially true when dealing with distributed algorithms characterized by frequent interaction between nodes. In practice, as was discovered with the Pushpins, this type of algorithm design flaw is easily avoided once caught, but difficult to catch without actually seeing it manifest itself at least once. Purposefully adding error to a simulation is one way to ensure good algorithm design from the very beginning.

5.5 Future Work

In some sense, the Pushpin Computing project is just beginning – the platform has been developed, but only minimal applications have been tested. Immediate future work, of course, includes patching the previously mentioned bugs affecting bandwidth. Near-term future applications include the as yet incomplete shape recognition application and some of the collaborations listed in §4. More full-featured sensing and actuation modules should not take long to develop, greatly expanding the utility of the Pushpins and potential applications. Additionally, a modest amount of interest has been shown regarding the Pushpins' potential utility in art and as a tangible interface for studying social networks.

Aside from using the Pushpin platform as it exists now, there are a couple key improvements that could be made. First, all the communications modules could stand to be redesigned and new ones built, such as low-power, near-field radio. This might include offloading all low-level communication processing to an additional processor, as originally suggested by the Paintable Computing specification. (Cygnal's new 300 series family of processors might be suitable for this). Second, the operating system should be broken up into two parts – a minimal kernel to manage very low-level functions and an updateable virtual machine to execute mobile bytecode. The advantages of this proposed fission include a less error-prone operating system, a more compact process fragment code size and correspondingly less need for bandwidth, and a more easily updated operating system. Third, as implied by switching to a virtual machine, an interpreted language should be developed (or an existing language should be modified) that is specifically suited for the primitives most important to distributed sensor networks. Even with the aid of the system functions provided by Bertha and a relatively high-level language like ANSI C, it is apparent after programming only a few process fragments that a higher level language would benefit the Pushpin endeavor greatly. Some work along these lines has been initiated by Seetharamakrishnan [14] and other work may yet exist, but even starting from a clean slate would be worthwhile as long as the goals and constraints of such a language were clearly set out ahead of time.

Finally, moving beyond the Pushpin platform laid out here and on to specialized ASICs (application-specific integrated circuits) is necessary to shrink down to smaller scales while increasing the overall computational and sensory capabilities of each node. The power and communication engineering challenges presented by this prospect are immense to be sure, but hardly insurmountable. Indeed, the first steps down this path have been taken –

recently secured NSF funding will support another two years of continued research [56]. If all goes well, it may not be long before artificial sensate skins with their own self-organizing 'nervous system' become a reality.

Chapter 6

Conclusions

"Remember; no matter where you go, there you are."

Buckaroo Banzai quoting Confucius

This work is motivated by ideas concerning self-organization, massively distributed sensor networks, and how the two might complement one another. The result is Pushpin Computing – a hardware and software platform designed for quickly prototyping and testing distributed sensor networks by employing a programming model based explicitly on algorithmic self-assembly.

At the hardware level, Pushpin Computing consists of an ensemble of approximately 100 identical Pushpins and a layered laminate plane to provide power. Every effort has been made to strike a balance between maintaining an expandable and general design and conforming to a host of constraints, such as small physical footprint, to ensure usability. Various power, communication, and sensor/actuator modules have been developed for use with the main processing module. Furthermore, developing other modules conforming to the Pushpin specification is relatively straightforward.

The Pushpin software model revolves around the concept of a process fragment, conforming closely to the specification laid out in Paintable Computing. The underlying operating system enabling process fragment operation, Bertha, handles everything from low-level communication and memory management to high-level process fragment execution scheduling and system calls. A first-generation integrated development environment (IDE) allows users to quickly author, compile, and upload process fragments, as well as adjust system parameters and collect debug information in real time.

A number of simple applications (embodied as process fragments) demonstrate how to use the Pushpin platform as a whole, and hint at several application domains and research areas where Pushpins might be of use. Initial performance feedback culled from these applications indicate that the system performs as designed, with the exception of what are expected to be minor bugs. These real-world results also suggest design points to be considered in future work in distributed sensor networks, both actual and simulated.

We do not yet understand how current or future sensor technology and what can be called algorithmic self-assembly will merge to form distributed sensor networks. Although, the Pushpin Computing platform is a potentially valuable research tool for designing and testing distributed sensor networks, there is much work to be done. Some of what lies ahead follows directly from and will build upon this work – design improvements, smaller and more integrated nodes, tuning of the operating system to fit run-time constraints, and more proof-of-concept applications all fall into this category. Other work presents significant engineering challenges that will require major breakthroughs to overcome – novel short-range, 'contactless' communication links between neighboring nodes and power generation/harvesting are the obvious challenges of this type. Finally, many fundamental theoretical concerns loom unanswered – agreeing upon a useful definition and theory of self-organization and determining the class of algorithms addressable by algorithmic self-assembly are only two of many such concerns.

In essence, the path toward realizing the full potential of distributed sensor networks is that of transforming the way we currently compute, sense, and change the world into the way the world itself computes, senses, and changes – distributedly and emergently. As usual, this path promises to be as interesting as the goal.

Appendix A

Circuit Layout and Schematic Diagrams

This appendix provides both the circuit layout and schematic diagrams for each of the Pushpin modules described in §2.3. All layout diagrams are of the same relative scale, except for the RS232 Communication Module and Power Module layout diagrams, which are at $\frac{3}{8}$ the scale as all the others.

A.1 Power Module

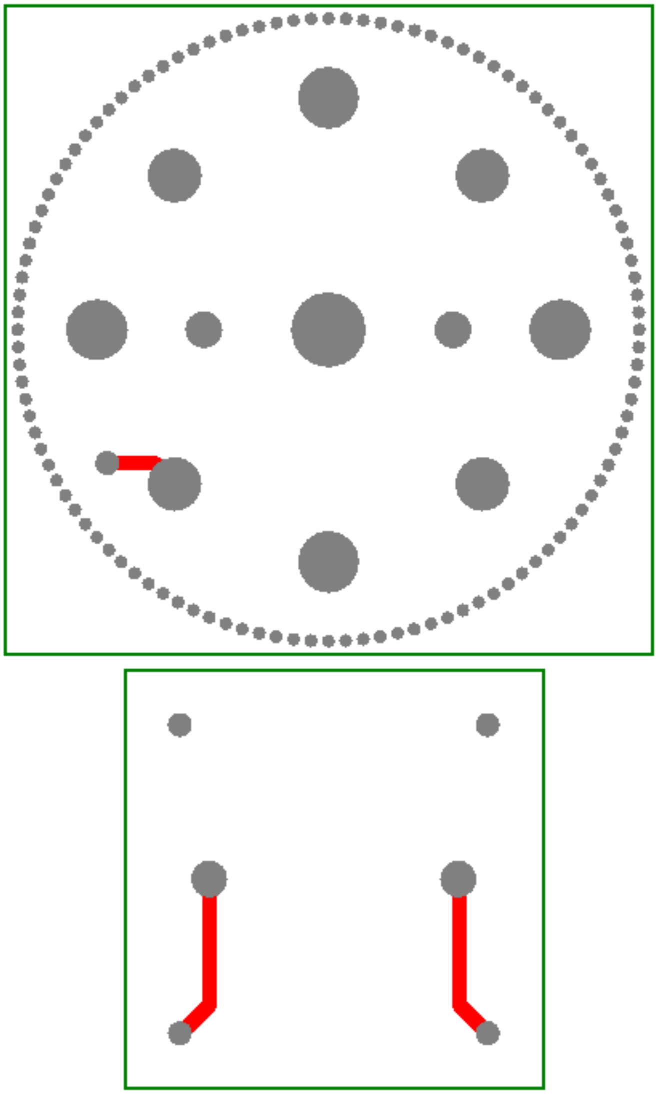


Figure A-1: Power Module – top board (top) and bottom board (bottom).

A.2 IR Communication Module

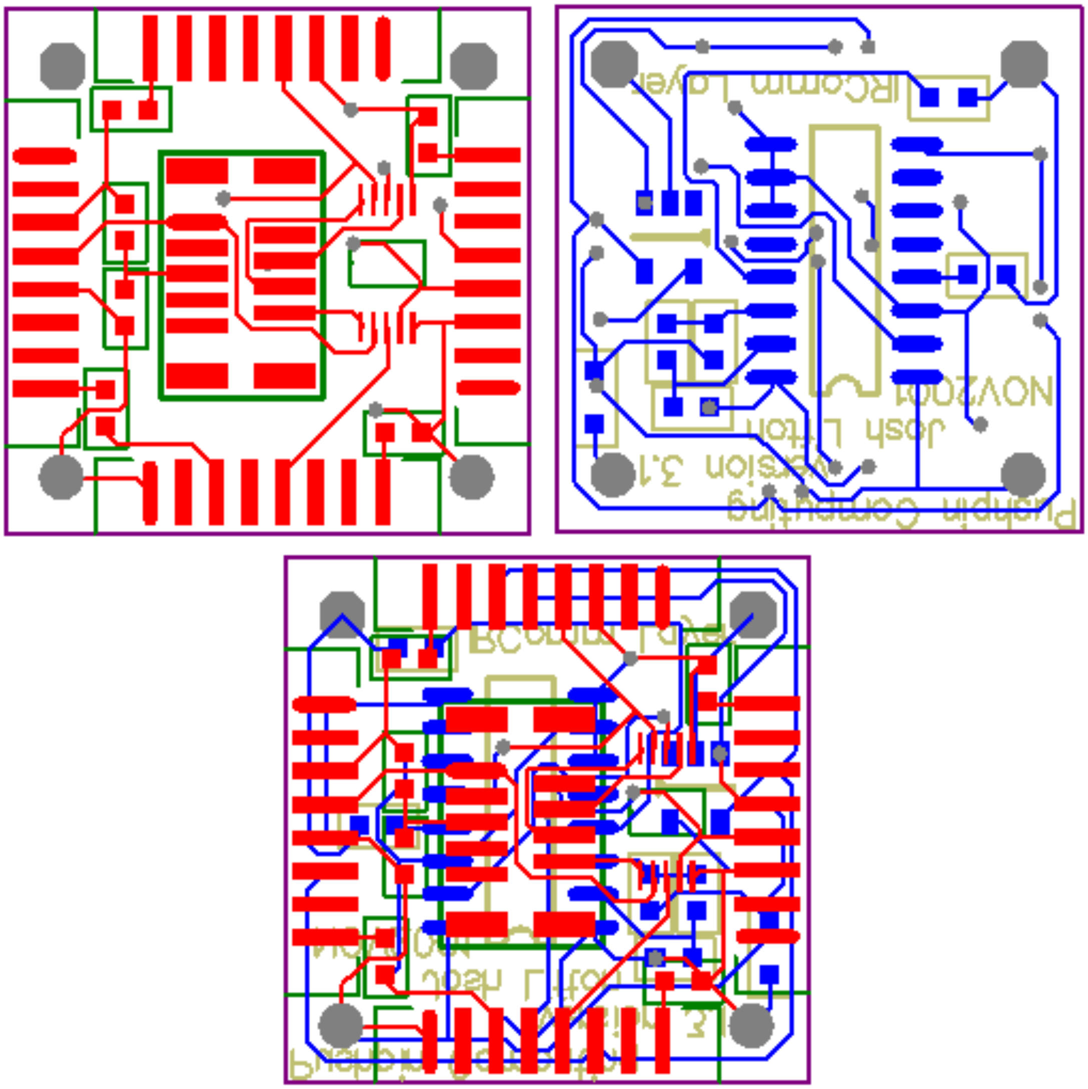


Figure A-2: IR Communication Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

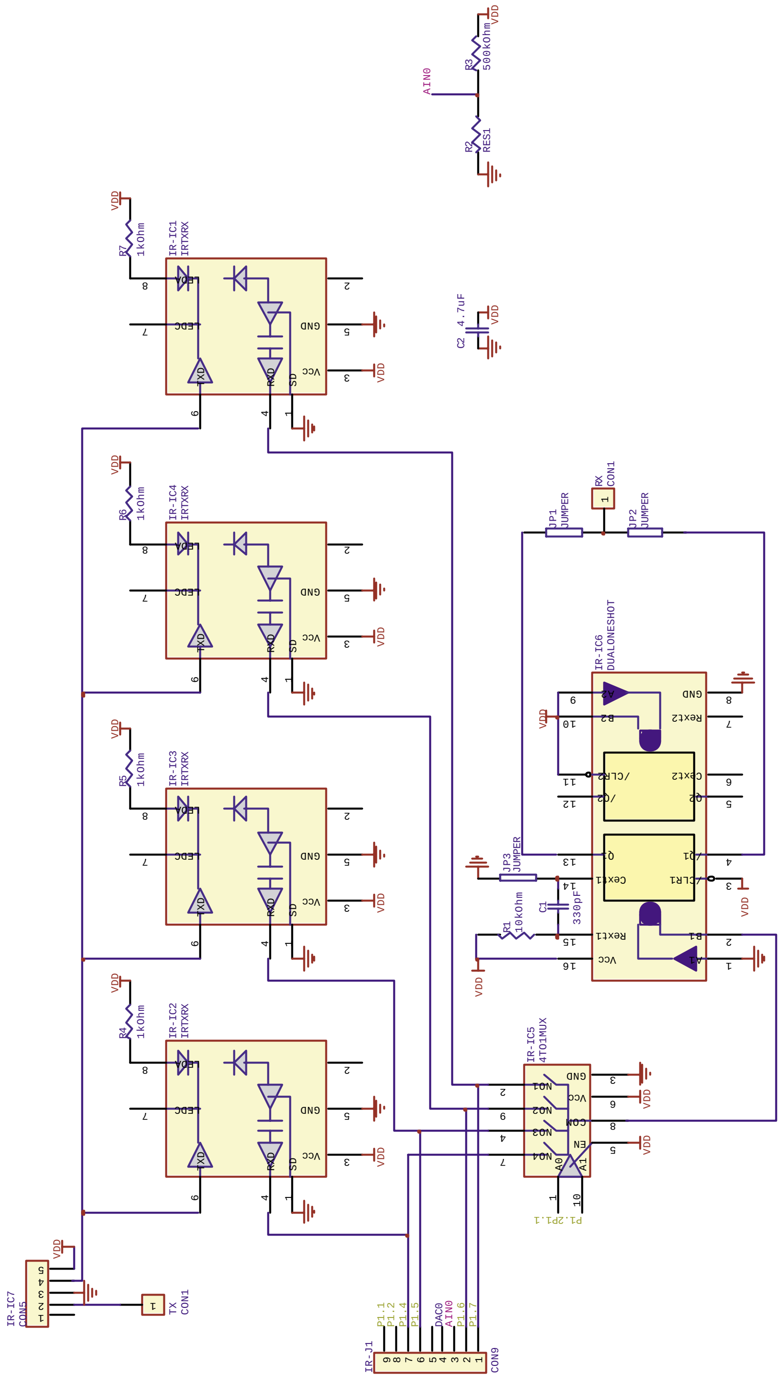


Figure A-3: IR Communication Module circuit diagram.

A.3 Capacitive Coupling Communication Module

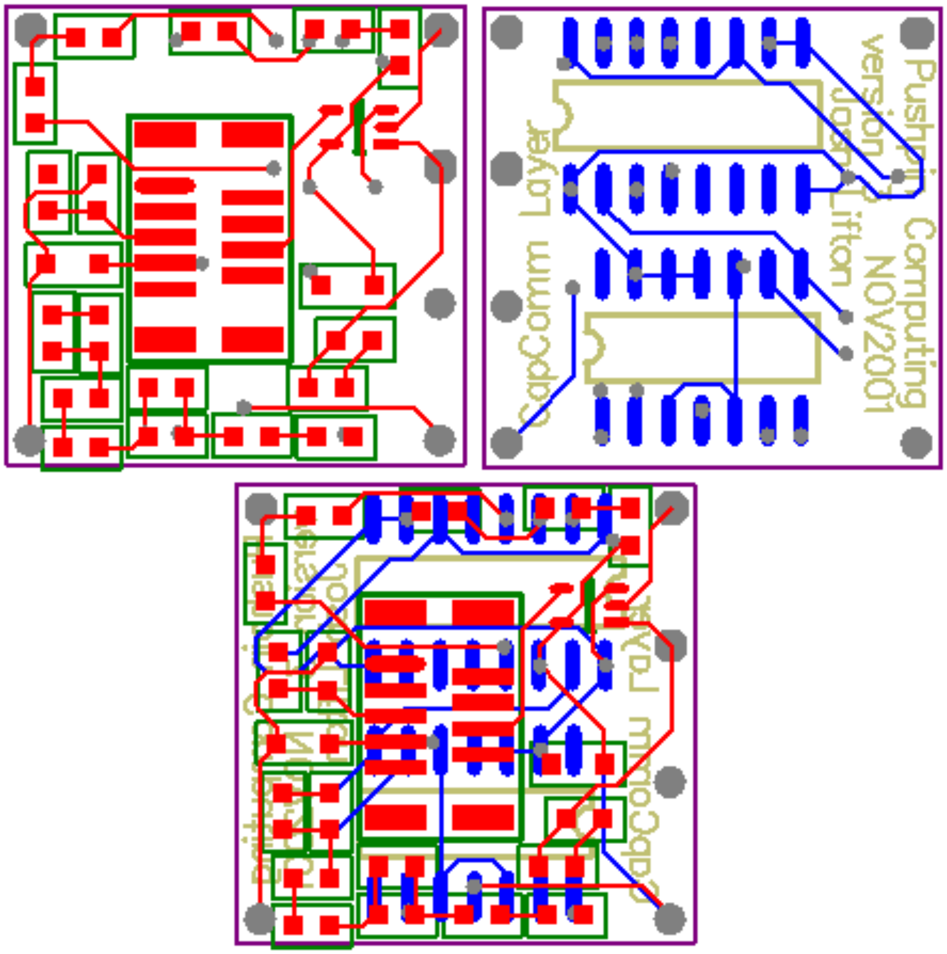


Figure A-4: Capacitive Coupling Communication Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

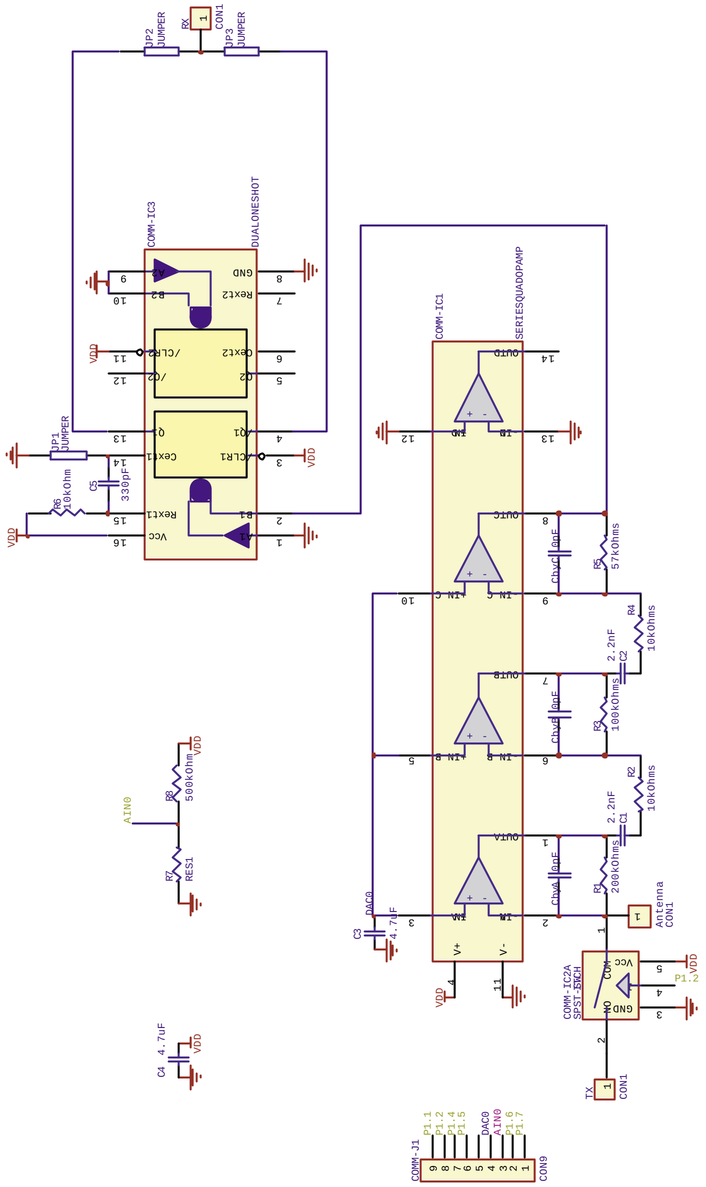


Figure A-5: Capacitive Coupling Communication Module circuit diagram.

A.4 RS232 Communication Module

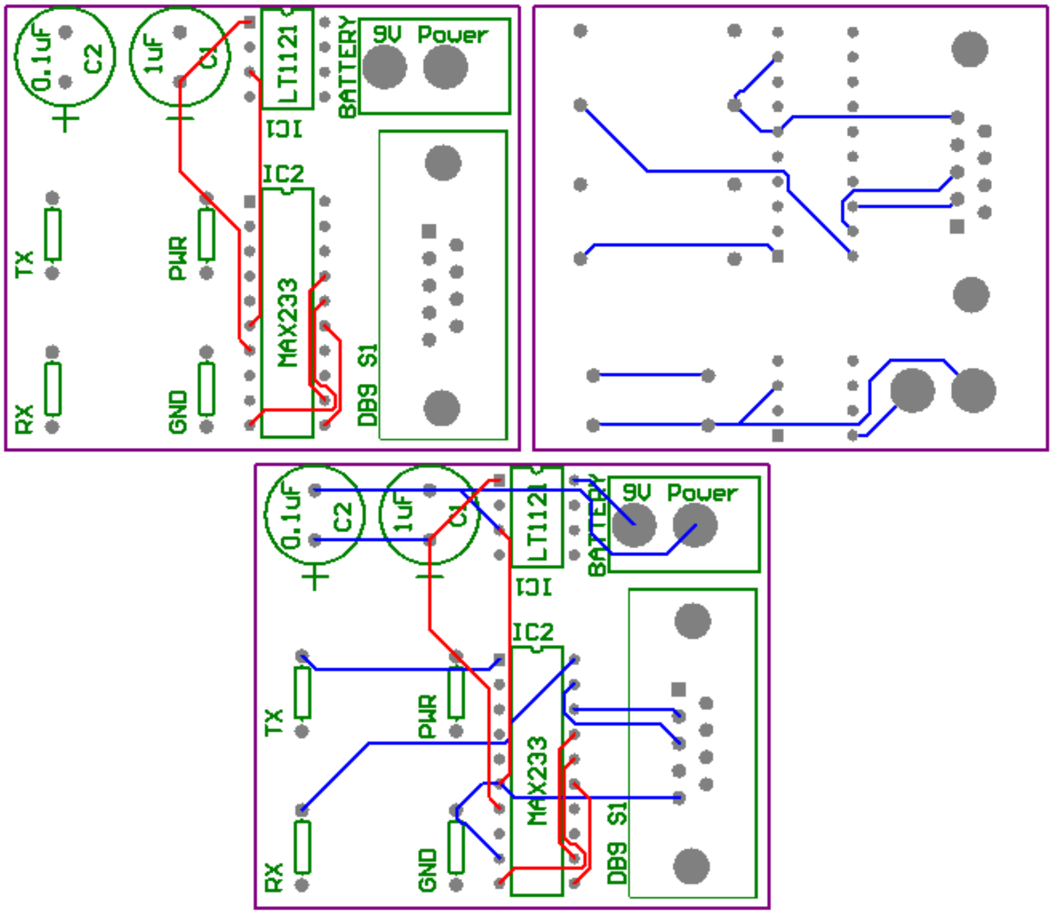


Figure A-6: RS232 Communication Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

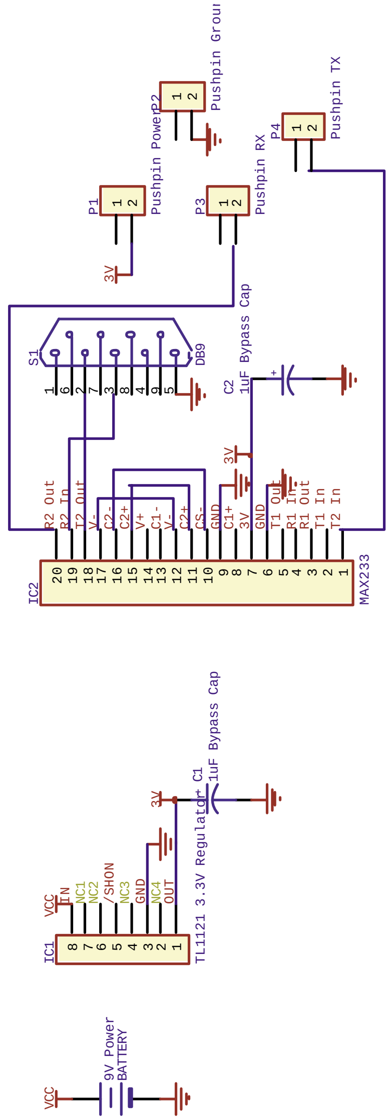


Figure A-7: RS232 Communication Module circuit diagram.

A.5 Processing Module

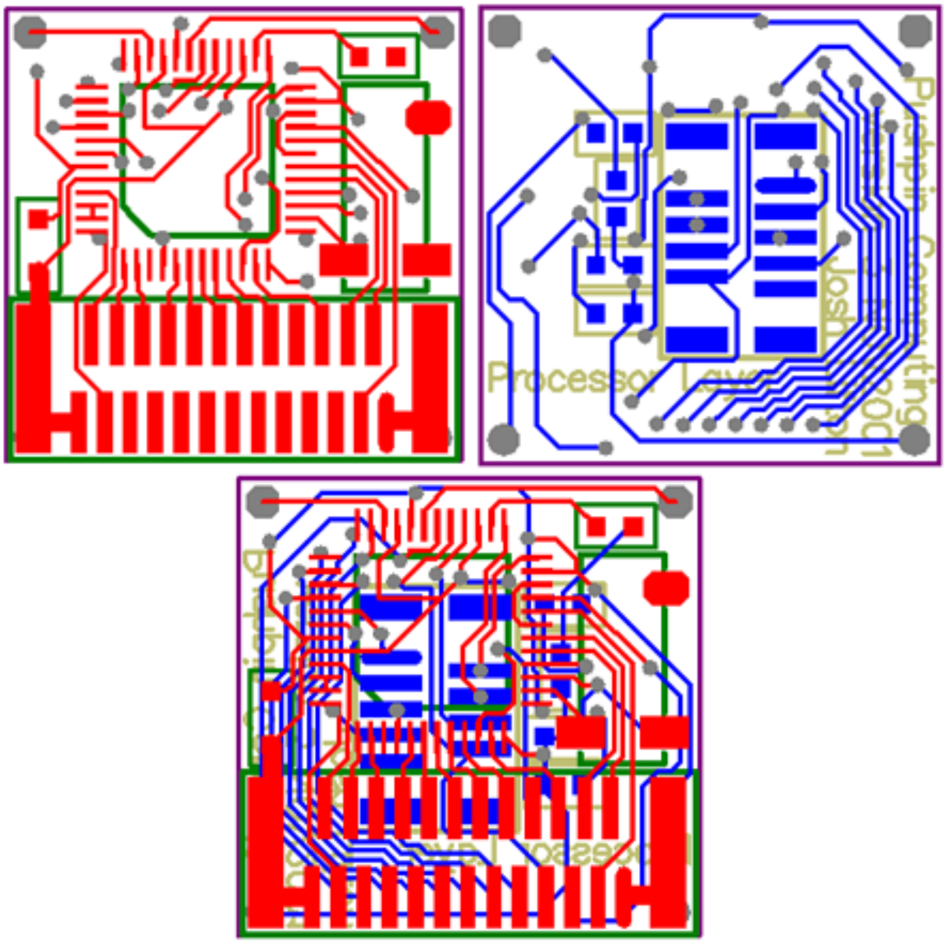


Figure A-8: Processing Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

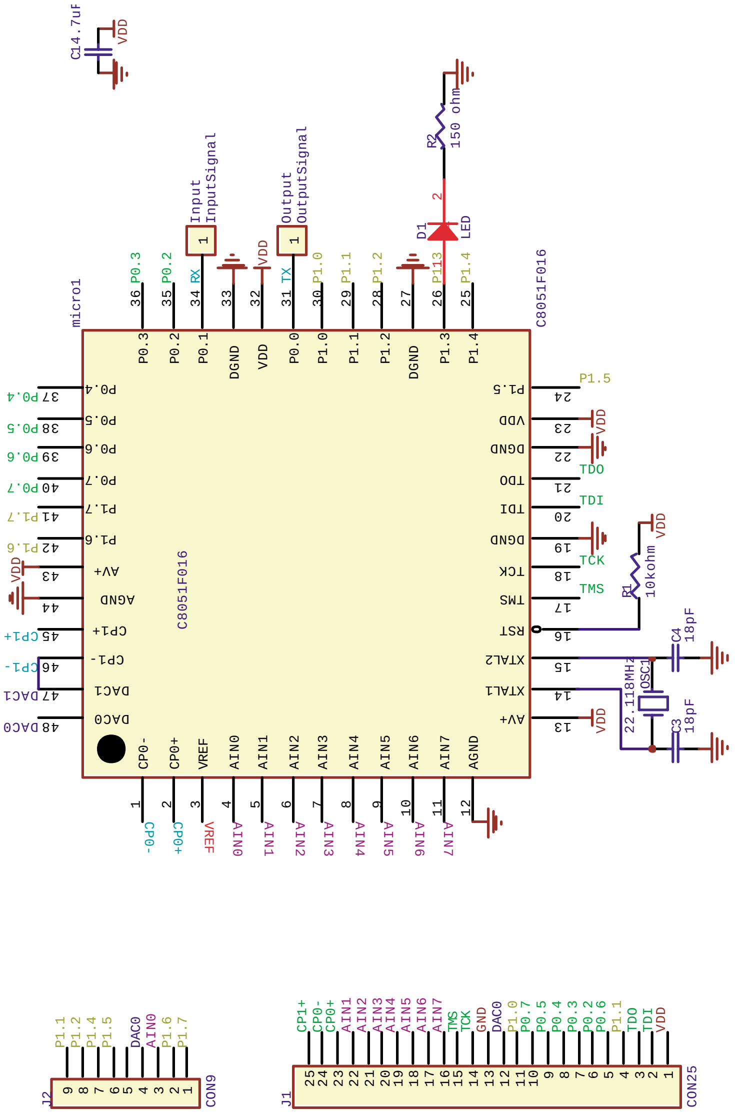


Figure A-9: Processing Module circuit diagram.

A.6 JTAG Programming Module

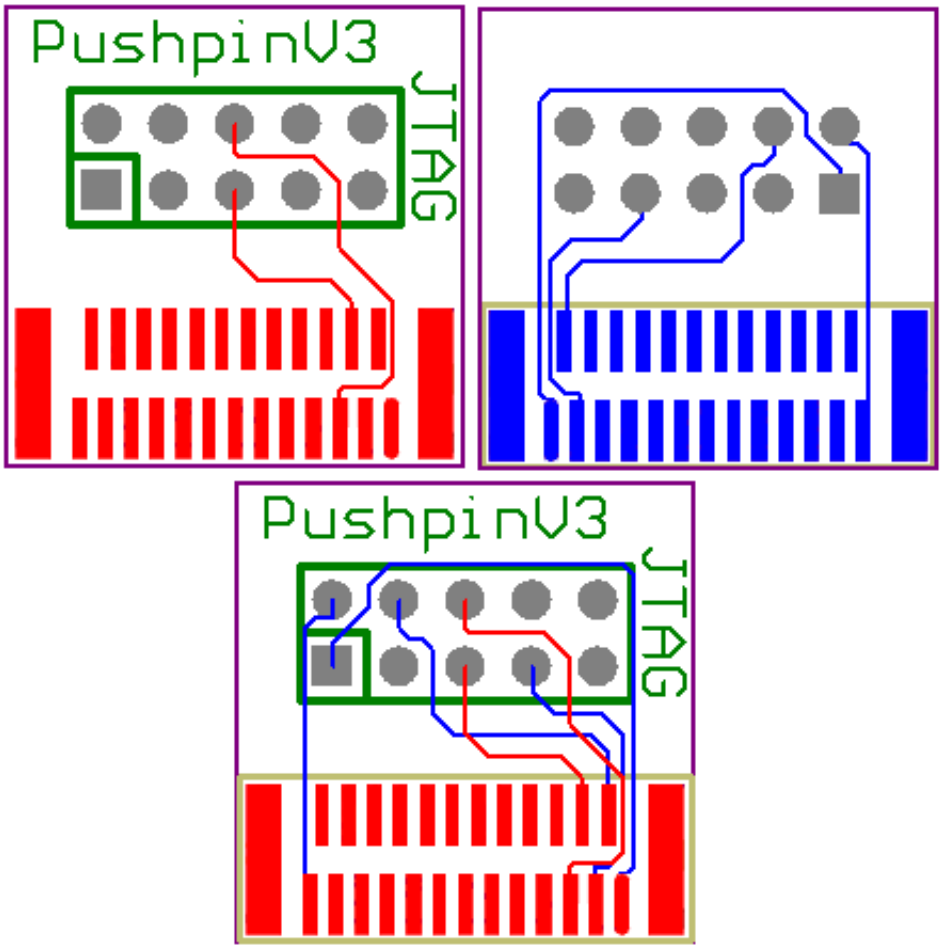


Figure A-10: JTAG Expansion Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

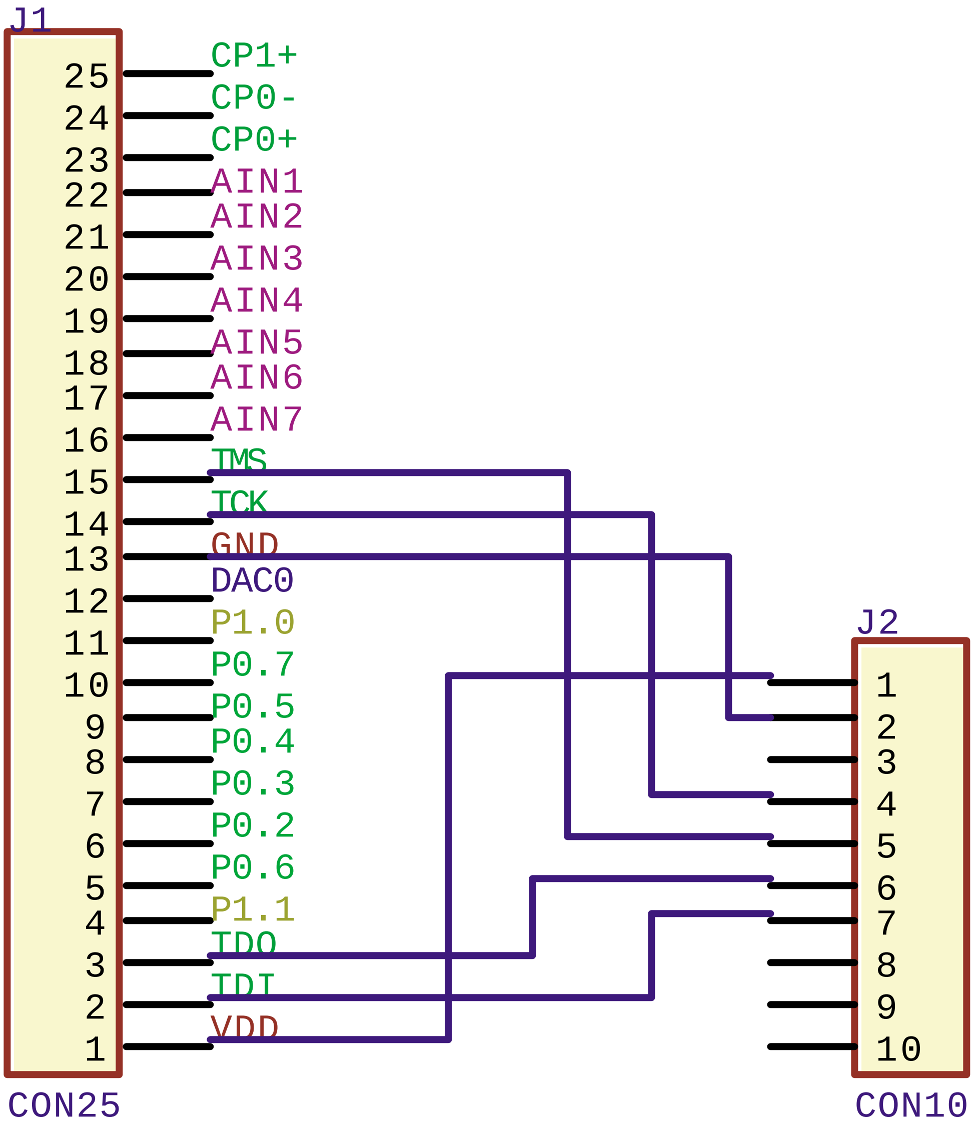


Figure A-11: JTAG Programming Module circuit diagram.

A.7 Prototyping Module

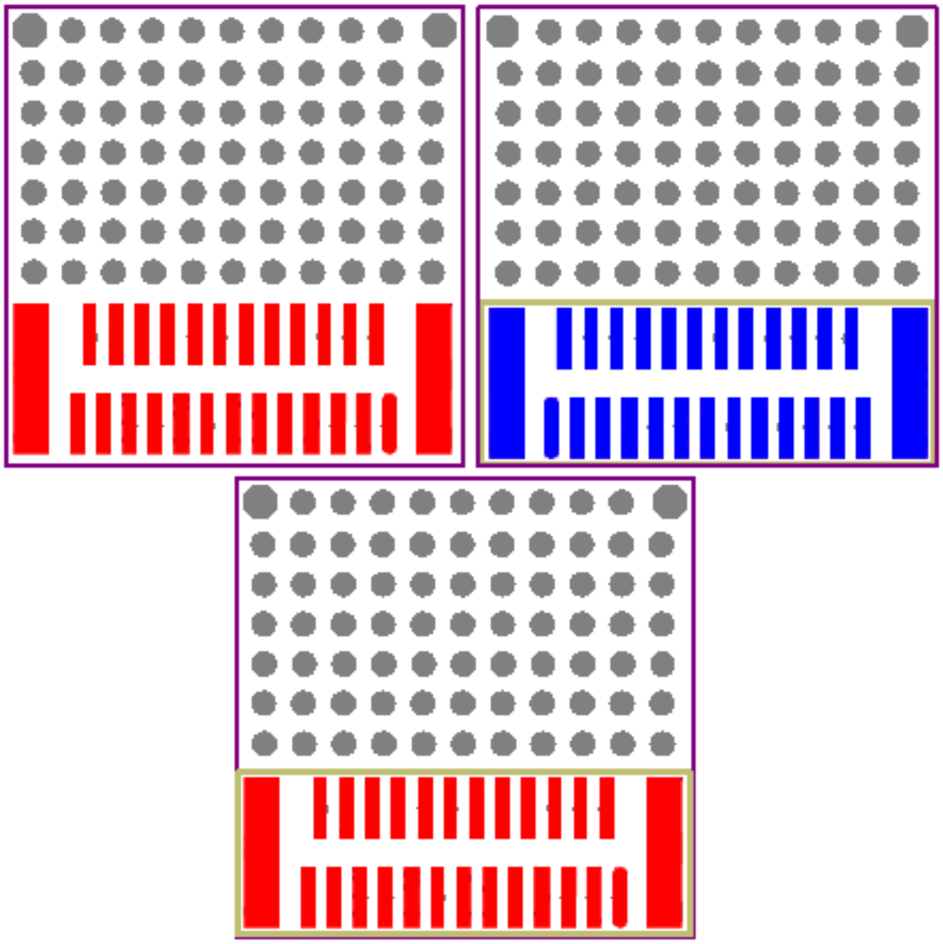


Figure A-12: Prototyping Expansion Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

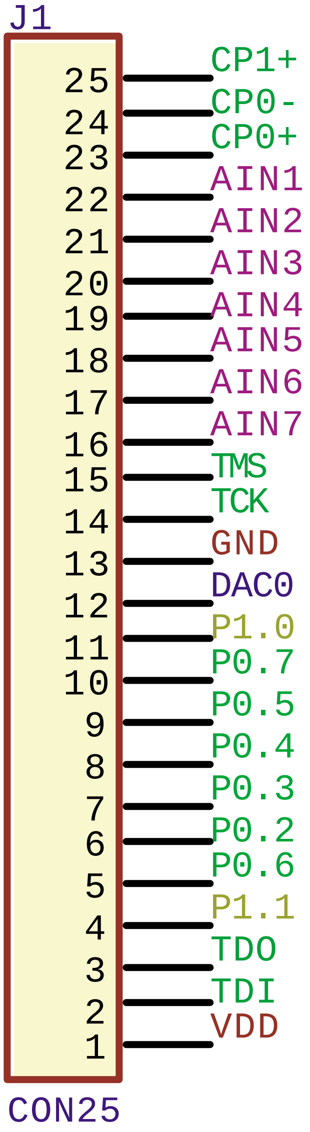


Figure A-13: Prototyping Module circuit diagram.

A.8 Light Sensing Module

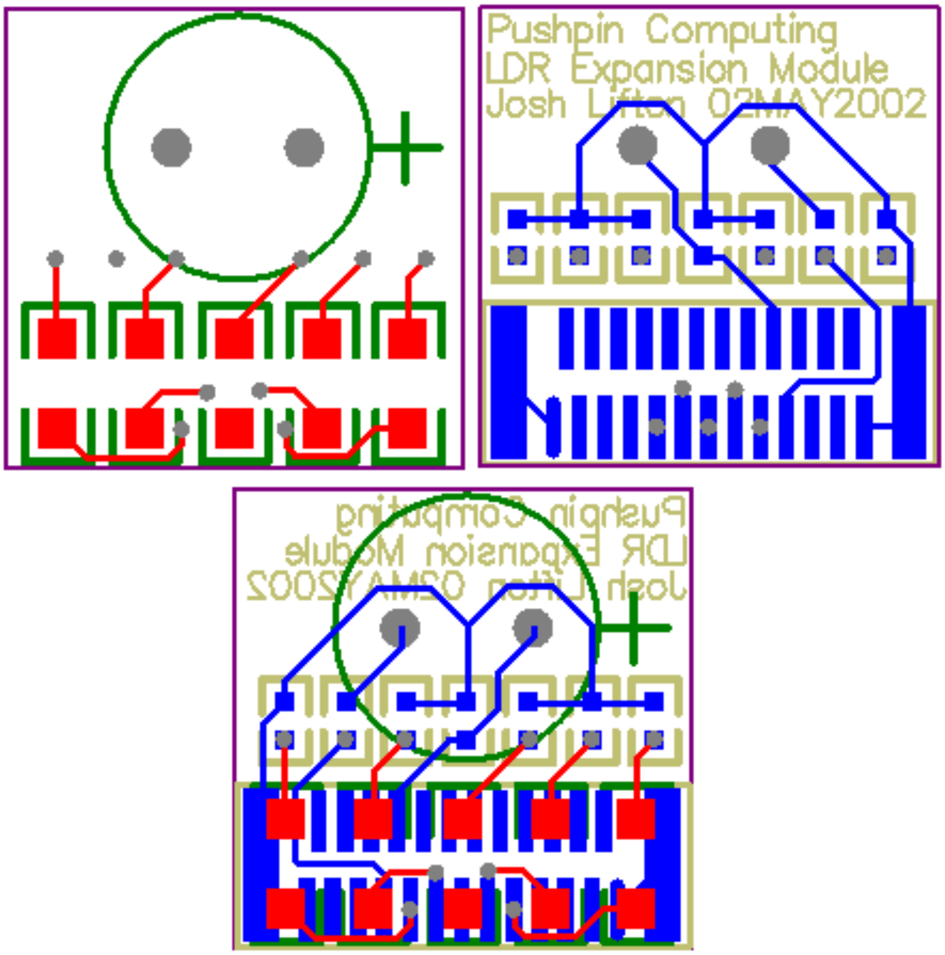


Figure A-14: Light Sensing Expansion Module – top layer (upper left), mirrored bottom layer (upper right), and both layers.

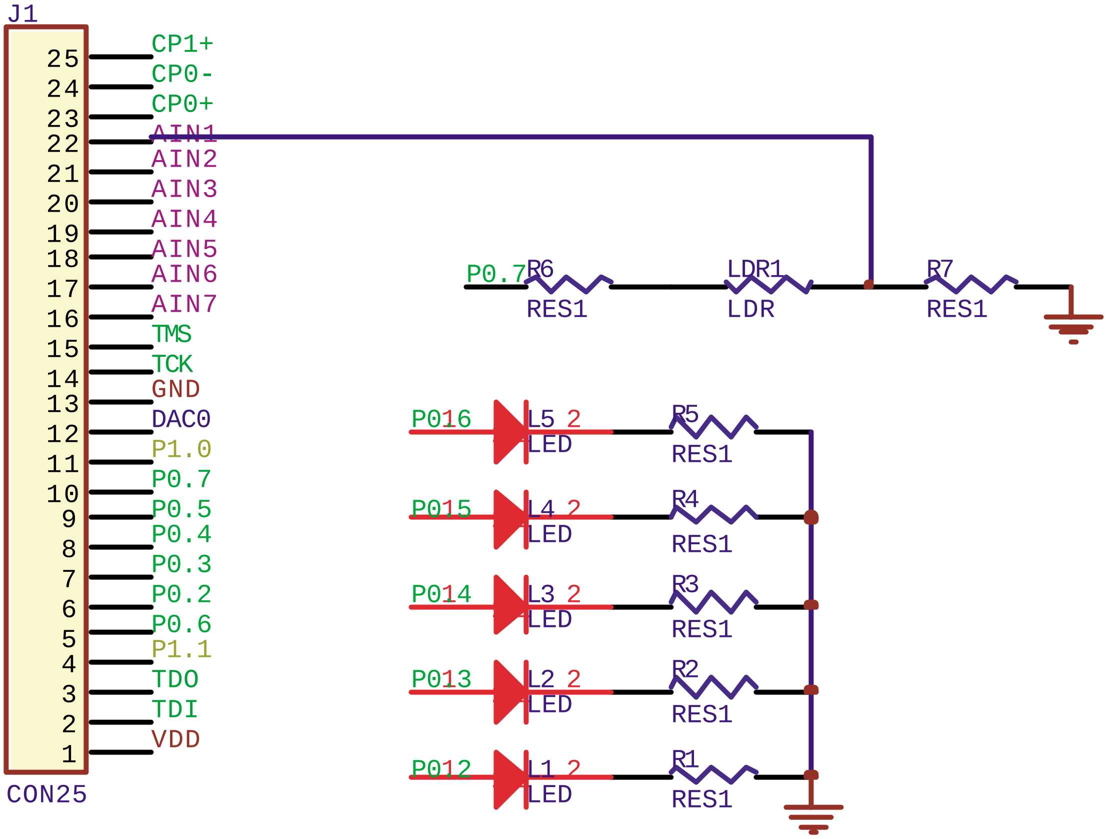


Figure A-15: Light Sensing Module circuit diagram.

Appendix B

Bertha System Calls

The following functions are provided by Bertha to all resident process fragments. These functions may be used when authoring a PFrag from within the Pushpin IDE.

`void delay(unsigned int)`

A blocking function that simply waits an amount of time roughly proportional to the value of the argument.

`void die(void)`

Indicates to Bertha that the current PFrag is ready to be removed. Bertha will shortly thereafter remove the PFrag's code, state, and BBS posts.

`void flashLDRLED(unsigned char, unsigned int, unsigned int)`

A blocking function that flashes the LED given by the first argument located on the LDR expansion module a number of times equal to the second argument at an interval given by the third argument.

`void flashLED(unsigned int, unsigned int)`

A blocking function that flashes the LED located on the processing module a number of times equal to the first argument at an interval given by the second argument.

`unsigned int getADCValue(unsigned char)`

Returns a 16-bit value read from the analog-to-digital converter channel passed as an argument. Values 0-8 are valid channels. Channel 8 is the internal temperature sensor. Only the least significant 10 bits of the 16-bit return value are valid.

`unsigned char getBBSPost(BBSPostID, unsigned int, unsigned char data *)`

Copies a number of bytes less than or equal to the second argument from the BBS post indicated by the first argument to the location given by the third argument. The first argument is the PFrag-specified ID of the post to be copied. Returns 0 if the post does not exist, 1 otherwise.

`unsigned char getBBSPostCount(void)`

Returns the total number posts currently contained in the BBS.

`unsigned char getNeighborCount(void)`

Returns the total number of neighbors.

`unsigned char getNthBBSPost(unsigned char, unsigned int, unsigned char data *)`

Copies a number of bytes less than or equal to the second argument from the BBS post indicated by the first argument to the location given by the third argument. The first argument is the numerical ordering in the BBS (e.g. first, second, third) of the post to be copied. Returns 0 if the post does not exist, 1 otherwise.

`PushpinLocalID getNthNeighborID(PushpinLocalID)`

Returns the local ID of the neighbor designated by the argument.

`unsigned char getNthPostFromMthSynopsis(unsigned char, PushpinLocalID, unsigned char data *)`

Copies the post designated by the first argument from the synopsis designated in the second argument to the location designated in the third argument. Returns 0 if the post does not exist, 1 otherwise. The number of bytes copied is the lesser of the number of bytes in the post to be copied and the listed size obtained when the third argument is cast as a BBSPost. The first argument indicates the numerical ordering in the synopsis (e.g. first, second, third) of the post to be copied. The second argument indicates the numerical ordering in the Neighborhood Watch of the synopsis to be copied from.

`unsigned char getNthPostFromSynopsis(unsigned char, PushpinLocalID, unsigned char data *)`

Copies the post designated by the first argument from the synopsis designated in the second argument to the location designated in the third argument. Returns 0 if the post does not exist, 1 otherwise. The number of bytes copied is the lesser of the number of bytes in the post to be copied and the listed size obtained when the third argument is cast as a BBSPost. The first argument indicates the numerical ordering in the synopsis (e.g. first, second, third) of the post to be copied. The second argument is the neighborhood ID of the Pushpin from which the desired synopsis originated.

`unsigned char getNthSynopsisPostCount(PushpinLocalID)`

Returns the total number of posts in the synopsis designated by the argument. Returns 0 if the synopsis does not exist.

`PFragUID getPFragUID(void)`

Returns the compiler-generated 16-bit unique identifier for this PFrag.

`unsigned char getPostFromMthSynopsis(BBSPostID, PushpinLocalID, unsigned char data *)`

Copies the post designated by the first argument from the synopsis designated in the second argument to the location designated in the third argument. Returns 0 if the post does not exist, 1 otherwise. The number of bytes copied is the lesser of the number of bytes in the post to be copied and the listed size obtained when the third argument is cast as a BBSPost. The first argument is the PFrag-specified ID of the of the post to be copied. The first post found matching this ID is the one copied. The second argument indicates the numerical ordering (e.g. first, second, third) in the Neighborhood Watch of the synopsis to be copied from.

`unsigned char getPostFromSynopsis(BBSPostID, PushpinLocalID, unsigned char data *)`

Copies the post designated by the first argument from the synopsis designated in the second argument to the location designated in the third argument. Returns 0 if the post does not exist, 1 otherwise. The number of bytes copied is the lesser of the number of bytes in the post to be copied and the listed size obtained when the third argument is cast as a BBSPost. The first argument is the PFrag-specified ID of the of the post to be copied. The first post found matching this ID is the one copied. The second argument is the neighborhood ID of the Pushpin from which the desired synopsis originated.

`unsigned char getSynopsisPostCount(PushpinLocalID)`

Returns the total number of posts in the synopsis belonging to the neighbor corresponding to the argument. Returns 0 if the neighbor does not exist.

`unsigned char isStateReplicated(void)`

Returns 1 if the current PFrag's initial state was transferred from another Pushpin, 0 otherwise.

`unsigned char postToBBS(BBSPostID, unsigned int, unsigned char data *)`

This function writes a single post to the BBS. The post is rejected if it is larger than the BBS can accommodate. The first argument is the post ID assigned to the post. The second argument is the number of bytes of content to be posted. The third argument is a pointer to the beginning of the message to be posted, which is assumed to be of the length indicated by the second argument. Returns 1 if the post was successfully written to the BBS, 0 otherwise.

`unsigned int random(void)`

A wrapper for the rand() function provided by stdlib.h. Returns a pseudo-random number between 0 and 32767 ($2^{15} - 1$). PFrags shouldn't call rand() directly because of seed initialization issues.

`unsigned char removeAllPosts(void)`

Removes all the BBS posts associated with the current PFrag.

`unsigned char removePost(BBSPostID)`

This function deletes a single post from the BBS. The first post in the BBS with an ID matching the argument and originally posted by the active PFrag is removed. Returns 1 if the post was successfully removed, 0 otherwise.

`void setLEDIntensity(unsigned char, unsigned char)`

When the Pushpin is equipped with the LDR expansion module, this function sets the duty cycle of the pulse width modulator corresponding to the LED given by the first argument (RED1, RED2, AMBER1, YELLOW1, GREEN1), based on the value of the second argument. The minimum duty cycle is 0xFF (OFF), ramping up linearly to a maximum duty cycle corresponding of 0x00 (ON).

`unsigned char transfer(PushpinLocalID)`

Initiates a transfer to the Pushpin indicated by the argument. Returns 1 if transfer was successfully initiated, 0 otherwise. Note that a successful transfer initiation does not guarantee a successful transfer.

Appendix C

Process Fragment Source Code

The following listings present process fragment source code as it would appear in the Pushpin IDE, with the exception of the Shape Recognition source code, which is the source code for a PFrag run on the Paintable Computing simulator.

C.1 Diffusion PFrag Source Code

```
/*
 * Name: diffuse.c
 * Author: Josh Lifton
 * Last Modified: 15JUN2002
 * Summary: Copies itself to any uninfected neighboring Pushpins.
 */

#define INFECTED 23
#define NUMBEROFLEDS 5
#define OFF 0xFF
#define ON 0x00


/*
 * Although this state variable isn't used, it is needed
 * to avoid a compile-time error.
 */
typedef struct ProcessFragmentState {
    unsigned char dumbyState;
} PFragState;

unsigned int update(unsigned int arg0, unsigned int arg1) {
    /*
     * Variables local to this function.
     */
    BBSPost post;
    unsigned char isPopulated;
    unsigned char currentNeighbor;
    unsigned char numberOfNeighbors;
    unsigned char currentPost;
    unsigned char numberOfPosts;
    unsigned char r1, r2, a1, y1, g1;

    /*
     * Change LED display.  Fades the LED display from
     * red to green and back again.
     */
    r1  = 255;
    r2 = 204;
    a1 = 153;
    y1 = 102;
    g1 = 51;
    while (r1--) {
        (r1 < 51) ? r2++ : r2--;
        (r1 < 102) ? a1++ : a1--;
        (r1 < 153) ? y1++ : y1--;
        (r1 < 204) ? g1++ : g1--;
        setLEDIntensity(RED1, r1);
        setLEDIntensity(RED2, r2);
        setLEDIntensity(AMBER1, a1) ;
        setLEDIntensity(YELLOW1, y1);
        setLEDIntensity(GREEN1, g1);
        delay(2000);
    }
    r1++;
    r2++;
    a1++;
    y1++;
    g1++;
    while (++r1) {
        (r1 >= 51) ? r2++ : r2--;
        (r1 >= 102) ? a1++ : a1--;
        (r1 >= 153) ? y1++ : y1--;
        (r1 >= 204) ? g1++ : g1--;
        setLEDIntensity(RED1, r1);
        setLEDIntensity(RED2, r2);
        setLEDIntensity(AMBER1, a1) ;
        setLEDIntensity(YELLOW1, y1);
        setLEDIntensity(GREEN1, g1);
        delay(2000);
    }

    /*
     * Check if there are any uninfected neighbors.  If so,
     * infect them.
     */
    numberOfNeighbors = getNeighborCount();
    isPopulated = 0;
    for (currentNeighbor = 1; currentNeighbor <= numberOfNeighbors;
         currentNeighbor++) {
        numberOfPosts = getNthSynopsisPostCount(currentNeighbor);
        for (currentPost = 1; currentPost <= numberOfPosts;
             currentPost++) {
            post.size = sizeof(BBSPost);
            if (getNthPostFromMthSynopsis(currentPost,
                currentNeighbor, (unsigned char data *) &post)) {
                if (post.uid == getPFragUID()) {
                    isPopulated = 1;
                    break;
                }
            }
        }
        if (!isPopulated) {
            transfer(getNthNeighborID(currentNeighbor));
            return 0;
        }
    }

    /*
     * Add a return statement to avoid a compile-time error.
     */
    return arg0 + arg1;
}

unsigned int install(unsigned int arg0, unsigned int arg1) {
    /*
     * Post a message stating this Pushpin is infected.
     */
    postToBBS(KITINFECTED, 0, 0);

    /*
     * Add a return statement to avoid a compile-time error.
     */
    return arg0 + arg1;
}

unsigned int deinstall(unsigned int arg0, unsigned int arg1) {
    /*
     * Turn off all LEDs and delete this PFrag.
    setLEDIntensity(RED2, OFF);

     */
    setLEDIntensity(RED1, OFF);



setLEDIntensity
(
AMBER1
,
OFF
);
    setLEDIntensity(YELLOW1, OFF);
    setLEDIntensity(GREEN1, OFF);
    die();

    /*
     * Add a return statement to avoid a compile-time error.
     */
    return arg0 + arg1;
}
```

C.2 Gradient PFrag Source Code

```c
/*
 * Name: grad.c
 * Author: Josh Lifton
 * Last Modified: 15JUN2002
 * Summary: Builds a gradient from a point of origin.
 */

void setColor(unsigned char);

typedef struct ProcessFragmentState {
    unsigned char color;
    unsigned char hops;
    unsigned char wait;
} PFragState;

#define numberOfColors 5
#define hopsID 0x29
#define defaultWait 5

unsigned int update(unsigned int arg0, unsigned int arg1) {
    /*
     * Variables local to this function.
     */
    BBSPost post;
    PushpinLocalID transmitTo;
    unsigned char isPopulated;
    unsigned char isParentAlive;
    unsigned char currentNeighbor;
    unsigned char numberOfNeighbors;
    unsigned char currentPost;
    unsigned char numberOfPosts;

    if (state.wait) {
        return state.wait--;
    }

    /*
     * Infect the last neighbor found to be uninfected.  At the same
     * time, compare the distance of the neighbors from the origin
     * with your own distance from the origin.  Choose the smallest
     * and add one if it's a neighbor's distance.
     */
    numberOfNeighbors = getNeighborCount();
    transmitTo = GLOBALADDRESS;
    isPopulated = 0;
    isParentAlive = 0;
    for (currentNeighbor = 1;
         currentNeighbor <= numberOfNeighbors;
             currentNeighbor++) {
        numberOfPosts = getNthSynopsisPostCount(currentNeighbor);
        for (currentPost = 1;
             currentPost <= numberOfPosts;
             currentPost++) {
            post.size = sizeof(BBSPost);
            if (getNthPostFromMthSynopsis(currentPost,
                                          currentNeighbor,
                                    (unsigned char data *) &post)) {
                if (post.uid == getPFragUID()) {
                    isPopulated = 1;
                    if ((post.postID == hopsID) &&
                        (post.message[0] < state.hops)) {
                        isParentAlive = 1;
                        state.hops = post.message[0] + 1;
                    }
                }
            }
        }
        if (!isPopulated) {
            transmitTo = currentNeighbor;
        }
    }

    /*
     * Wither and die if there isn't anyone closer to the origin
     * and you aren't the origin yourself.
     */
    if (!isParentAlive && state.hops != 0) {
        return deinstall(0,0);
    }

    /*
     * Change the 5-LED display to reflect the distance from origin.
     */
    setColor(state.hops);

    /*
     * Update the BBS posting listing distance from origin.
     */
    removePost(hopsID);
    post.message[0] = state.hops;
    postToBBS(hopsID, sizeof(unsigned char), post.message);

    /*
     * Infect the uninfected neighbor found above.
     */
    if (transmitTo != GLOBALADDRESS) {
        transfer(getNthNeighborID(transmitTo));
    }

    /*
     * Add a return statement using the passed arguments to avoid a
     * compile-time error.
     */
    return arg0 + arg1;
}

void setColor(unsigned char c) {
    setLEDIntensity(state.color, 0xFF);
    if (c < numberOfColors) {
        state.color = c;
    }
    else {
        state.color = numberOfColors - 1;
    }
    setLEDIntensity(state.color, 0x00);

    /*
     * Add a return statement to avoid a compile-time error.
     */
```

```
}

    return;



unsigned int install(unsigned int arg0, unsigned int arg1) {
    unsigned char currentNeighbor;
    unsigned char numberOfNeighbors;
    unsigned char currentPost;
    unsigned char numberOfPosts;
    BBSPost post;

    /*
     * Check for other Grad PFrags.  Delete this PFrag if it is not
     * the first instance of a Grad PFrag on this Pushpin.
     */
    numberOfPosts = getBBSPostCount();
    for (currentPost = 1;
         currentPost <= numberOfPosts;
             currentPost++) {
        if (getNthBBSPost(currentPost,
                          sizeof(BBSPost),
                          (unsigned char data *) &post)) {
            if (post.uid == getPFragUID()) {
                return deinstall(0,0);
            }
        }
    }


    /*
     * Initialize state variables.
     */
    state.wait = defaultWait;
    state.hops = 0;

    /*
     * Determine the number of hops this Pushpin is from the origin.
     */
    numberOfNeighbors = getNeighborCount();
    for (currentNeighbor = 1;
         currentNeighbor <= numberOfNeighbors;
             currentNeighbor++) {
        numberOfPosts = getNthSynopsisPostCount(currentNeighbor);
        for (currentPost = 1;
             currentPost <= numberOfPosts;
                 currentPost++) {
            post.size = sizeof(BBSPost);
            if (getNthPostFromMthSynopsis(currentPost,
                                          currentNeighbor,
                                    (unsigned char data *) &post)) {
                if (post.uid == getPFragUID()) {
                    state.hops = 0xff;
                    break;
                }
            }
        }
        if (state.hops) {
            break;
```

```
            }
        }

        /*
         * Update the LEDs and BBS posting listing distance from origin.
         */
        removePost(hopsID);
        post.message[0] = state.hops;
        postToBBS(hopsID, sizeof(unsigned char), post.message);
        setColor(state.hops);

        /*
         * Add a return statement using the passed arguments to avoid a
         * compile-time error.
         */
        return arg0 + arg1;
    }

    unsigned int deinstall(unsigned int arg0, unsigned int arg1) {
        /*
         * Turn off all LEDs before deletion.
         */
        setLEDIntensity(RED1,0xFF);
        setLEDIntensity(RED2,0xFF);
        setLEDIntensity(AMBER1,0xFF);
        setLEDIntensity(YELLOW1,0xFF);
        setLEDIntensity(GREEN1,0xFF);
        die();

        /*
         * Add a return statement using the passed arguments to avoid a
         * compile-time error.
         */
        return arg0 + arg1;
    }
```

C.3 Retina PFrag Source Code

```
/*
 * Name: retina.c
 * Author: Josh Lifton
 * Last Modified: 15JUN2002
 * Summary: Divides an ensemble of Pushpins equipped with
 * light sensors into those in the dark, those in the light
 * and those on the dark side of the border between the two.
 */

void setColor(unsigned char);

typedef struct ProcessFragmentState {
    unsigned char brightness;
    unsigned int baseline;
} PFragState;

#define DARK 0
#define LIGHT 1

#define BRIGHTNESS 0x38

unsigned int update(unsigned int arg0, unsigned int arg1) {
    /*
     * Variables local to this function.
     */
    BBSPost post;
    unsigned char onBorder;
    unsigned char isPopulated;
    unsigned char currentNeighbor;
    unsigned char numberOfNeighbors;
    unsigned char currentPost;
    unsigned char numberOfPosts;

    /*
     * Change the 5-LED display to reflect light conditions.
     */
    if (getADCValue(1) > state.baseline) {
        state.brightness = LIGHT;
        setLEDIntensity(RED2, 0x00);
    }
    else {
        state.brightness = DARK;
        setLEDIntensity(RED2, 0xFF);
    }

    /*
     * Update the BBS posting listing number of neighbors.
     */
    removePost(BRIGHTNESS);
    post.message[0] = state.brightness;
    postToBBS(BRIGHTNESS, sizeof(unsigned char), post.message);

    /*
     * Attempt to infect the first uninfected neighbor found.  At
     * the same time, compare the brightness reading from each
     * neighbor with the local brightness reading.  Based on this
     * comparison, determine if this Pushpin is located in the dark,
     * in the light, or on the dark side of the border between light
     * and dark.
     */
    numberOfNeighbors = getNeighborCount();
    onBorder = 0;
    isPopulated = 0;
    for (currentNeighbor = 1;
         currentNeighbor <= numberOfNeighbors;
             currentNeighbor++) {
        numberOfPosts = getNthSynopsisPostCount(currentNeighbor);
        for (currentPost = 1;
             currentPost <= numberOfPosts;
                 currentPost++) {
            post.size = sizeof(BBSPost);
            if (getNthPostFromMthSynopsis(currentPost,
                                          currentNeighbor,
                                    (unsigned char data *) &post)) {
                if (post.uid == getPFragUID()) {
                    isPopulated = 1;
          if (post.postID==BRIGHTNESS && post.message[0]==LIGHT) {
                        onBorder = 1;
                    }
                }
            }
        }
        if (!isPopulated) {
            transfer(getNthNeighborID(currentNeighbor));
            break;
        }
    }

    if (onBorder && state.brightness == DARK) {
        setLEDIntensity(GREEN1, 0x00);
    }
    else {
        setLEDIntensity(GREEN1, 0xFF);
    }

    /*
     * Add a return statement using the passed arguments to avoid a
     * compile-time error.
     */
    return arg0 + arg1;
}

unsigned int install(unsigned int arg0, unsigned int arg1) {
    /*
     * Calibrate the light sensor to current lighting conditions.
     */
    getADCValue(1);
    delay(0xFFFF);
    state.baseline = getADCValue(1);
    state.brightness = state.baseline;
    setLEDIntensity(AMBER1, 0x00);

    /*
     * Add a return statement using the passed arguments to avoid a
     * compile-time error.
     */
    return arg0 + arg1;
}
```

```
unsigned int deinstall(unsigned int arg0, unsigned int arg1) {


/*
     * Turn off all elements of the LED display.
     */
    setLEDIntensity(RED1,0xFF);
    setLEDIntensity(RED2,0xFF);
    setLEDIntensity(AMBER1,0xFF);
    setLEDIntensity(YELLOW1,0xFF);
    setLEDIntensity(GREEN1,0xFF);

    /*
     * Add a return statement using the passed arguments to avoid a
     * compile-time error.
     */
    return arg0 + arg1;
}
```

C.4 Shape Recognition PFrag Source Code for Paintable Computing Simulator

```
/*
 * Name: PatRecCS.java
 * Author: Bill Butera
 * Last Modified: 26APR2002
 * Summary: Copies itself to any uninfected neighboring Pushpins
 * and then queries neighbors to determine if it is in a corner
 * of a light pattern being projected across all Pushpins.  The
 * number of corners in the ensemble is tallied to determine the
 * shape of the light pattern -- circle, square, or triangle.
 */

package CSdir ;

import java.io.Serializable;
import Simulator.* ;


/*
 *  PatternRecCS                                    Bill Butera
 *                                                  MIT Media Lab
 *
 *  Class definition for a process fragment which enters the
 *  computing substrate at a selected point and propagates a copy
 *  of itself to every particle, where it looks for posts from a
 *  binary sensor. In particles where the sensors are on, this
 *  pfrag carries out a multi-stage algorithm to classify the
 *  continuous region as a square, a triangle or a circle.
 *
 *  A simple corner counting scheme is used to classify the region
 *  into one of the three shapes. Each pfrag estimates the
 *  likelihood that the local particle is a corner by computing the
 *  precentage of the particle's neighbors with their sensors
 *  active.  Pfrags that believe that they are on a corner
 *  propagate this belief as entries in their post. Pfrags use this
 *  radiating info to count the number of corners in the entire
 *  region, which serves as a discriminant function to classify the
 *  region.
 *
 *  On entry, the first task is propagate a copy of itself to every
 *  particle.  Once all the neighbors are infected, it checks the
 *  local HomePage for posts from binary sensors. If a sensor
 *  reading set to "ON" is found, this pfrag assumes that it is
 *  within a closed region and then attempts to classify that
 *  region.
 *
 *  PatRec's posting to the home page is a vairable length post with
 *  a maximum of 19 integer words (a 4 word header + 5 corners):
 *
 *   [0]  Tag ID:        Shape Classifier
 *   [1]  active flag    (int: flag values defined in CSTags)
 *   [2]  corner score   (int: percentage of neighbors within the
 *                             closed region)
 *   [3]  nEntries       (int: number of 3-byte entries for corners
 *                             (max 5))
 *
 *  Each "corner entry" consists of three words:
 *
 *   [0]  ID tag for corner  (randomly selected value between 0-255)
 *   [1]  hop count          (distance from source particle for this
 *                                  corner entry)
 *   [2]  corner score       (int: percentage of neighbors within
 *                                  the closed region)
 *
 */


public class PatRecCS      implements CodeSegment, Serializable   {

  private static int            ioPortID=CSTags.IOPortID;
  private static int            patRecID=CSTags.patRecID;

  // Current Particle-of-residence
  private Particle              p;

  // Flag to indicate whether a transfer has been queued
  private boolean               trQueued;

  // The Data Segment portion of the Process Fragment
  private double[]              cfPost;

  // Table of corner entries from neighboring posts.
  private double[][]            cornerTable;

  // Number of active entries in the table.
  private int                   nCorners;

  // Maximum number of corner-entries in cornerTable.
  private static int            maxCorners=5;

  // Array for data from corner centered at current particle.
  private double[]              localCorner;

  // Unique ID for tagging corner posts from this pfrag.
  // For now, cheat and use the particle ID.
  private double                localTagID;


  //------ Constructors


  public PatRecCS () {

    cfPost = new double[19];
    cfPost[0] = (double) patRecID;
    cfPost[1] = (double) CSTags.patRecInactiveID;
    cfPost[2] = 0.0;
    cfPost[3] = 0.0;
    for (int i=4; i<19; ++i)
      cfPost[i] = 0.0;
    cornerTable = null;
    nCorners = 0;
    localCorner = new double[3];
    for (int i=0; i<3; ++i)
    p = null;

      localCorner[i] = 0.0;



trQueued
 =
false
;
    return;
  }


  //
  // The 8 methods required by the CodeSegment interface.
  //


  // Install Process Fragment on current Particle
  public void installCS () {
    p.postOnCSList(this);
    p.postOnHomePage(this, cfPost);
    localTagID = (double) p.getID();
    return;
  }


  // Remove Process Fragment from Particle
  public void deInstallCS () {
    p.purgeHomePage(this);
    p.purgeCSList(this);
    return;
  }



  public void transferRefused () {
    trQueued = false;
    return;
  }


  // Transfer Process Fragment to destination Particle
  public void execTransfer (Particle pNew) {

    PatRecCS csCopy = new PatRecCS ();
    csCopy.setParticle (pNew);
    csCopy.installCS();
    trQueued = false;
    return;
  }



  public void updateCS () {
    if (trQueued)
      return;

    //----------------------------------------------------------------
    //  Check local HomePage for post with "sensor ON" reading.
```

```
//

HomePage hp = p.getHomePage();
int nPosts = hp.getNPosts ();
double[][] post = hp.getValues();

boolean localSensorON = false;
boolean isInIOPort = false;
if (post != null)  {
  for (int i=0; i<nPosts; i++) {
    if (post[i][0] == CSTags.sensorID)  {
      if (post[i][1] == CSTags.sensorOn)
        localSensorON = true; }

    else if (post[i][0] == CSTags.IOPortID)
      isInIOPort = true; }}

//----------------------------------------------------------------
//  Search the neighborhood looking for an uninfected particle.
//  If no uninfected particles detected,
//  and if the local HomePage contains a "SensorON" post,
//  begin classification.

int nHp = 0;
int sensorCount = 0;
initCornerTable ();
Particle emptyPart = null;
Particle[] daHood = p.getDaHood();

if (daHood != null) {
  nHp = daHood.length;
  int offset=(int) Math.rint((double)nHp * p.getRandomDouble());

  for (int i=0; i<nHp; ++i) {
    boolean foundPatRec = false;
    boolean foundSensor = false;
    boolean foundActiveSensor = false;
    boolean isIOPort = false;
    int j = (i + offset) % nHp;
    post = ((daHood[j]).getHomePage()).getValues();
    if (post != null) {
      nPosts = post.length;
      for (int k=0; k<nPosts; ++k) {
        if (post[k][0] == CSTags.patRecID) {
          foundPatRec = true;
          buildCornerTable (post[k]); }

        else if (post[k][0] == CSTags.sensorID) {
          foundSensor = true;
          if (post[k][1] == CSTags.sensorOn)
            foundActiveSensor = true; }

        else if (post[k][0] == CSTags.IOPortID)
          isIOPort = true; }}

    if ((foundSensor)  &&  (!foundPatRec)  &&  (!isIOPort)) {
      emptyPart = daHood[j];
      break; }
    else if (foundActiveSensor  &&  (!isIOPort))
      sensorCount++; }}
  if (emptyPart != null)
    queueTransfer (emptyPart);
  else {
    boolean repost = false;
    if (!localSensorON) {
      if (cfPost[1] == CSTags.patRecActiveID) {
        repost = true;
        cfPost[1] = CSTags.patRecInactiveID;
        cfPost[2] = 0.0;
        cfPost[3] = 0.0; }}
    else {
      int newScore = 0;
      for (int j=0; j<3; ++j)
        localCorner[j] = 0.0;
      if (nHp > 0)
        newScore = 129 - ((int) Math.rint (128.0 *
                  (double) sensorCount / (double) nHp));
      if (newScore >= 78.0) {
        localCorner[0] = localTagID;
        localCorner[1] = 0.0;
        localCorner[2] = newScore; }
      repost = updateLocalPost (newScore); }
    if (repost) {
      p.purgeHomePage(this);
      p.postOnHomePage(this, cfPost); }}

  return;
}



public void setParticle (Particle p) {
  this.p = p;
  return;
}


public double[] getDataSeg () {
  int i;
  if (cfPost == null)
    return (null);
  int n = cfPost.length;
  double[] ds = new double[n];
  for (i=0; i<n; ++i)
    ds[i] = cfPost[i];
  return (ds);
}



public String getName () {
  String s1 = "Shape_Classifier_with_activity_flag_=_";
  String s2 = s1.concat((new Integer((int)cfPost[1])).toString());
  String s3 = s2.concat("_and_estimator_=_");
  String s4 = s3.concat((new Integer((int)cfPost[2])).toString());
  return (s4);
}
```

```
//
// Other public methods
//



//
// private Methods
//


private void queueTransfer (Particle pNew) {
  if (trQueued)
    System.err.println ("queueTransfer():_Unexpected_transfer
__________________________request_to_particle_at_"+pNew.getPos());
  if (pNew != null) {
    trQueued = true;
    pNew.postTrReq(this); }
  return;
}



private void initCornerTable () {
  cornerTable = new double[maxCorners][4];
  nCorners = 0;
  return;
}



// Scan post for corner-entries and add them to the running table
private void buildCornerTable (double[] post) {
  if (post[3] > 0) {
    for (int i=0; i<post[3]; ++i) {
      int baseOffset = 4 + (i * 3);
      int IDtag = (int) post [baseOffset];
      int hc = (int) post [baseOffset + 1];
      int score = (int) post [baseOffset + 2];
      if ( (IDtag != (int) localTagID)  &&
           (IDtag >= 0.0) ) {
        if (nCorners == 0) {
          cornerTable [nCorners][0] = IDtag;
          cornerTable [nCorners][1] = hc;
          cornerTable [nCorners][2] = score;
          nCorners++; }
        else {
          boolean cornerInTable = false;
          for (int j=0; j<nCorners; ++j) {
            if (IDtag == cornerTable[j][0])  {
              cornerInTable = true;
              if (hc < cornerTable[j][1]) {
                cornerTable[j][1] = hc;
                cornerTable[j][2] = score; }
              else if ((hc == cornerTable[j][1]) && (score > 0.0))
                cornerTable[j][2] = score; }}
          if (!cornerInTable) {
            if (nCorners < (maxCorners - 1)) {
              cornerTable [nCorners][0] = (double) IDtag;
              cornerTable [nCorners][1] = (double) hc;
              cornerTable [nCorners][2] = (double) score;
              nCorners++; }
            else
              System.err.println ("CSdir.PatRecCS.buildCornerTable
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣():␣Too␣many␣corners␣floating␣around"); }}} }}

    return;
  }



  // Compare the local post against the contents of the cornerTable
  // Return a "repost" / "no repost" decision.
  // This function is a two stage process. In the first stage,
  // every corner entry in the local post is compared against the
  // contents of the cornerTable. If necessary, the local corner
  // info is updated.  In the second stage, all those entries in
  // the cornerTable that are not already accounted for in the local
  // post are added to the local post.
  private boolean updateLocalPost (int  newScore)  {

    boolean repost = false;
    if (cfPost[2] != newScore) {
      cfPost[2] = newScore;
      repost = true; }

    if (addLocalCorner ())
      repost = true;

    cullDeadLocalPosts ();

    if (cfPost[1] != CSTags.patRecActiveID) {
      cfPost[1] = CSTags.patRecActiveID;
      repost = true; }
    // For every corner in the local post, compare it against the
    // cornerTable and update its state.
    clearCornerTableFlags ();
    if (cfPost[3] > 0.0) {
      double[] corner = new double[3];
      for (int i=0; i<cfPost[3]; ++i) {
        int baseOffset = 4 + (i * 3);
        for (int j=0; j<3; ++j)
          corner[j] = cfPost [baseOffset + j];
        if (updateCorner (corner)) {
          repost = true;
          for (int j=0; j<3; ++j)
            cfPost [baseOffset + j] = corner[j]; }}}

    cullDeadLocalPosts ();

    if (nCorners > 0) {
      for (int i=0; i<nCorners; ++i)
        // corner is unaccounted for
        if (cornerTable[i][3] == 0.0)
          // corner is not tagged for kill (marked
          // with a negative score)
          if (cornerTable[i][2] >= 0.0) {
            if (cfPost[3] < (double)maxCorners) {
```

```
              int baseOffset = 4 + ((int)cfPost[3] * 3);
              cfPost [baseOffset] = cornerTable[i][0];
              cfPost [baseOffset+1] = cornerTable[i][1] + 1;
              cfPost [baseOffset+2] = cornerTable[i][2];
              cfPost[3] = cfPost[3] + 1;
              repost = true; }
            else
              System.err.println ("CSdir.PatRecCS.updateLocalPost():
␣␣␣␣␣␣␣␣␣␣␣␣␣Attempt␣to␣append␣too␣many␣corners␣to␣local␣post"); }}

    return (repost);
  }



  // u p d a t e C o r n e r
  // Compare a single input corner to the corners from the
  // cornerTable, and if necessary update the corner's state. If
  // the input corner is a local corner (a corner centered at
  // this particle), do nothing, because the local corner is
  // treated in the previously called function addLocalCorner().
  //
  // This was once-upon-a-time, a very simple routine. But then I
  // decided to add the ability for a corner to signal its demise
  // via propagation of a termination condition (by posting
  // a negative score). As a space saving measure I insisted on
  // continued use of the cornerTable.  And that has required an
  // unwelcome exercise in logic.
  //
  // For any other given input corner, there is presummably at most
  // one corner in the cornerTable.  More to the point, there are 2
  // possible states for the input corner (marked and unmarked)
  // and 6 possible states for the matched corner from the
  // cornerTable (nothing found, no source found, source found - and
  // each of these 3 modified by marked/unmarked state).
  // The truth table in words:
  //
  //   Input corner      matched corner      modifier     action
  //   ============      ==============      ========     ======
  //
  //   unmarked          nothing found                    do nothing
  //   marked                 "                           remove
  //
  //   unmarked          no source found                  mark
  //   marked                 "              marked       remove
  //   marked                 "              unmarked     do nothing
  //
  //   unmarked          source found        unmarked     do nothing
  //   marked                 "              unmarked     unmark
  //                                              (an unlikely state)
  //
  //   unmarked          source found        marked       mark
  //   marked                 "              marked       do nothing
  //
  // Some of these states are of dubious relevance. But I just want
  // to see if this approach is any good. If the approach proves
  // out, I will do a squeeze later.
  //
  // Whenever a match is found from the cornerTable, the matching
  // table entry must be flagged as "accounted for" by setting a
  // flag in an array element.
```

```
private boolean updateCorner (double[] corner) {
  boolean cornerMarked = false;
  if (corner[2] < 0.0)
    cornerMarked = true;
  double IDtag = corner[0];
  double hc =    corner[1];
  double score = corner[2];
  boolean sourceFound = false;
  boolean matchFound = false;
  boolean matchMarked = false;
  for (int c=0; c<nCorners; ++c) {
    if (IDtag == cornerTable[c][0])  {
      matchFound = true;
      cornerTable[c][3] = 1.0;
      if (hc > cornerTable[c][1])
        sourceFound=true;
      if (cornerTable[c][2] < 0.0)
        matchMarked = true;
      break; }}

  boolean mark = false;
  // it's called "depost" because "remove" seems to be a
  // reserved word.
  boolean depost = false;
  boolean repost = false;
  // no match found in cornerTable
  if (!matchFound) {
    if (cornerMarked)
      depost = true; }
  // match found, but hop count is too high
  else if (!sourceFound) {
    if (!cornerMarked)
      mark = true;
    else if (matchMarked)
      depost = true; }
  // source found  --  and it's unmarked
  else if (!matchMarked) {
    if (cornerMarked) {
      corner[2] = -1.0 * corner[2];
      repost = true; }}
  // source found  --  and it's marked
  else {
    if (!cornerMarked)
      mark = true; }

  if (depost  &&  (corner[0] > 0.0)) {
    corner[0] = -1.0 * corner[0];
    repost = true; }
  if (mark  &&  (corner[2] >= 0.0)) {
    corner[2] = -1.0 * corner[2];
    repost = true; }

  return (repost);
}

// a d d L o c a l C o r n e r
// If a nonzero score is found in the localCorner array, check if
// the local corner should be added to the local post. This
// decision is a function of three conditions and a binary
// modifier. The conditions:
```

```
  //  1) a nonzero score in the localCorner array
  //  2) The local corner is inhibited by a neighbor
  //      (stored in the cornerTable)
  //  3) the localCorner is already present in the localPost
  //
  // The modifier: the state of the state of the local corner entry
  // in the post (marked or unmarked for termination)
  //
  // The truth table in words:
//
//localCorner|local corner|local corner   |corner|neighbor   |action
//score      |inhibited by|already present|marked|from       |
//           |neighbor    |in local post  |      |CornerTable|
//           |            |               |      |marked     |
//===========|============|===============|======|===========|=======
//
//score = 0.0                                                 nothing
//
//score >=    present     not inhibited                       nothing
//threshold
//
//     ”      not present        ”                            add to
//                                                             post
//
//     ”      not present     inhibited                       nothing
//
//     ”      present            ”           no               mark it
//
//     ”          ”              ”           yes     no       nothing
//
//     ”          ”              ”           yes     no/absent remove
//
  // Mark any occurance of the localCorner that appears in the
  // cornerTable as "accounted for"
  //
  // Like the function "updateCorner ()", this was
  // once-long-ago-in-a-far-away-land a simple function. But then I
  // decided to incorporate inhibition, at which point simplicity
  // and compactness both went rocketing out the window.


  private boolean addLocalCorner () {
    boolean repost = false;
    if (localCorner[2] > 0.0) {
      boolean inhibited = false;
      boolean cornerFound = false;
      boolean cornerMarked = false;
      int     cornerIndex = 0;
      boolean neighborFound = false;
      boolean neighborMarked = false;
      boolean addPost = false;
      boolean mark = false;
      boolean depost = false;

      double IDtag = localCorner[0];
      double hc =    localCorner[1];
      double score = localCorner[2];
      for (int c=0; c<cfPost[3]; ++c) {
        int i = 4 + (c * 3);
        if (cfPost[i] == IDtag) {
          cornerFound = true;
        cornerIndex = c;
        if (cfPost[i+2] < 0.0)
          cornerMarked = true;
        break; }}

    for (int c=0; c<nCorners; ++c) {
      if (cornerTable[c][0] == IDtag) {
        cornerTable[c][3] = 1.0;
        neighborFound = true;
        if (cornerTable[c][2] < 0.0)
          neighborMarked = true; }
      else if (cornerTable[c][1] <= 2.0) {
        if (cornerTable[c][2] > score)
          inhibited = true;
        else if ((cornerTable[c][2] == score)  &&
                 (cornerTable[c][0] > IDtag))
          inhibited = true; }}

    if (!inhibited) {
      if (!cornerFound)
        addPost = true; }
    else if (cornerFound) {
      if (!cornerMarked)
        mark = true;
      else if ((!neighborFound)  ||  neighborMarked)
        depost = true; }

    if (addPost) {
      if (cfPost[3] >= maxCorners)
        System.err.println ("CSdir.PatRecCS.addLocalCorner_():
_____________Attempt_to_append_too_many_corners_to_local_post");
      else {
        repost = true;
        int c = (int)cfPost[3];
        int k = 4 + (c * 3);
        cfPost[k] = localCorner[0];
        cfPost[k+1] = localCorner[1];
        cfPost[k+2] = localCorner[2];
        cfPost[3] = cfPost[3] + 1.0; }}
    else if (depost) {
      int c = cornerIndex;
      int k = 4 + (c * 3);
      if (cfPost[k] > 0.0) {
        cfPost[k] = -1.0 * cfPost[k];
        repost = true; }}
    else if (mark) {
      int c = cornerIndex;
      int k = 4 + (c * 3);
      if (cfPost[k+2] > 0.0) {
        cfPost[k+2] = -1.0 * cfPost[k+2];
        repost = true; }}}

    return (repost);
  }



  private void clearCornerTableFlags () {
    for (int i=0; i<maxCorners; ++i)
      // mark the corner as "unaccounted for"
      cornerTable[i][3] = 0.0;
    return;
  }



  //  "Dead" corners are marked by a negative ID number. Find these
  //  and remove them from the local post.
  private void cullDeadLocalPosts () {
    if (cfPost[3] > 0.0) {
      boolean rewrite = false;
      for (int i=0; i<cfPost[3]; ++i)  {
        if (cfPost [4 + (i * 3)] < 0.0)
          rewrite = true;
        if (rewrite) {
          if (i < (int)cfPost[3]-1) {
            for (int j=i; j<((int)cfPost[3]-1); ++j) {
              int k = 4 + (j * 3);
              int l = k + 3;
              for (int m=0; m<3; ++m, ++k, ++l)
                cfPost[k] = cfPost[l]; }}
          cfPost[3] = cfPost[3] - 1.0;
          // overwrite the last vacated corner with zeros
          int k = (int) cfPost[3];
          int l = 4 + (k * 3);
          cfPost[l] = 0.0;
          cfPost[l+1] = 0.0;
          cfPost[l+2] = 0.0; }}}

    return;
  }


}
```

Appendix D

Bertha OS Source Code

The following listings present the various source code components comprising the Bertha operating system.

D.1 Primary Header File

```
/*****************************************************************
 * Bertha.h
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 07MAR2002 by lifton.
 *
 * This file defines the memory layout for the Bertha operating
 * system as implemented on the Cygnal C8051F016 microprocessor.
 *****************************************************************/

/*
 * Enumerations used throughout the operating system.
 */

enum packetTypes {
  BERTHAMESSAGE, NWMESSAGE, PFRAGMESSAGE, PFRAGSTATEMESSAGE,
  RANDOMSEEDMESSAGE
};

enum BerthaMessageCodes {
  ERASEALLPFRAGS, ERASEPFRAG, ERASERANDOMSEED, TEMPNEIGHBORID,
  IDVETO, SETTIMESCALE
};

enum serialStates {
  DISABLED, IDLE, SCANNING, SENDINGTOADDRESS, RECEIVINGHEADER,
  RECEIVINGCONTENT, IGNORINGCONTENT, SENDINGHEADER, SENDINGCONTENT,
  PACKETRECEIVED, PACKETSENT, STASIS
};

enum commModes {
  DEFAULT=1, RS232=1, FOURWAYIR    // RS232 is the default mode.
};

enum QueuePacketReturnCodes {
  PACKETNOTQUEUED,
  PACKETQUEUED,
  PACKETTRANSMITTING
};

enum LEDcolors {
  RED1, RED2, AMBER1, YELLOW1, GREEN1
};

/*
 * Constants and definitions specific to the microprocessor
 * hardware.  Either 8051-specific or C8051F016-specific.
 */
#define NUMBEROFADCCHANNELS 9 /* Temp sensor + 8 pinned out.        */
#define COMMADCCHANNEL 0      /* ADC channel on comm module.        */
#define IRADCTHRESHOLD 0x0200 /* ADC value for which IR detected.   */
#define XdataSize 2048        /* 2Kbytes external memory (RAM).     */
#define FlashPageSize 512     /* Minimum flash memory erase size.   */
#define ErasedPFragChar 0xff  /* Erased flash overwrite character. */
typedef unsigned char DataAddress;        /* 256 bytes.             */
typedef unsigned char IdataAddress;       /* 256 bytes.             */
typedef unsigned int XdataAddress;        /* 2048 bytes.            */
typedef unsigned int CodeAddress;         /* 32 Kbytes.             */
typedef unsigned char ReservedMemoryArea; /* Smallest memory unit. */
typedef union AddressableLong {           /* Utility data type.     */
   unsigned long Long;   /* Access as a single unsigned long.      */
   unsigned int Int[2];  /* Access as individual unsigned integers.*/
   unsigned char Char[4];/* Access as individual unsigned bytes.   */
} AddressableLong;
typedef union AddressableInt {            /* Utility data type.     */
   unsigned int Int;     /* Access as individual unsigned integers.*/
   unsigned char Char[2];/* Access as individual unsigned bytes.   */
} AddressableInt;

/*
 * Timing constants.  Determined in large part by the bandwidth
 * available for communication, 92160 baud, which translates to
 * approximately 4600 bytes/second for the 4-way IR communication
 * module.  The largest possible PFrag and associated PFragState
 * is approximately 2500 bytes.
 */
#define ONESECOND 1843318
#define ONEMILLISECOND 1844
#define DELAYTIME 0xFFFF
#define minIDVetoWaitTime ((unsigned long) 3 * (unsigned long) \
  timeScale * (unsigned long) DELAYTIME)
#define minNeighborhoodBuildTime 0xFFFF
#define DEFAULTTIMESCALE 30
#define NEIGHBORTIMEOUT (5 * timeScale * DELAYTIME)

#define BerthaMessageSize 10                /* Minimum of 10 bytes. */
#define BerthaBBSPostSize BerthaMessageSize
#define maxNumberOfPFrags 1  // meme : just testing.
//#define maxNumberOfPFrags 9  /* Based on 2Kbytes/PFrag*/
#define maxPFragSize 2048   /* A pfrag's code is stored in one
                               2Kbyte sector of flash. */


typedef unsigned char PFragLocalID;
typedef unsigned int PFragUID;
typedef unsigned char PFragFunctionID;
typedef unsigned char BBSPostID;
typedef unsigned char PushpinLocalID;
/*
 * A list element is comprised of the number of bytes contained in
 * that element.  A pointer to a list element should be cast to the
 * appropriate data structure before use.
 */
typedef XdataAddress ListElement;
/*
 * A pointer to the first element in the list.
 */
typedef ListElement xdata * LinkedList;
typedef ListElement PFragStateSize;
typedef struct ProcessFragmentState {
  PFragStateSize size;  // Total size of state.
  PFragLocalID id;   // This state belongs to this process fragment.
  ReservedMemoryArea state; // First byte of actual state variables.
} PFragState;
```

```
  PFragSize size;     // Total size of this PFrag in bytes.

typedef ListElement PFragSize;
typedef struct ProcessFragment {



PFragUID

uid
;

//

A

UID

for

this

PFrag
'
s

code
.
  unsigned char crc;  // 8-bit check sum.
    PFragStateSize stateSize; // Size of this PFrag's state in bytes.
  ReservedMemoryArea codeFrag[maxPFragSize -
    (sizeof(PFragSize) + sizeof(PFragUID) + sizeof(unsigned char) +
    sizeof(PFragStateSize))];   // Remaining space for code.
} PFrag;
typedef ListElement BBSPostSize;
typedef struct BulletinBoardSystemPost {
  BBSPostSize size;
  PFragLocalID localID;
  PFragUID uid;
  BBSPostID postID;
  ReservedMemoryArea message[1];
} BBSPost;
typedef ListElement NeighborSize;
typedef struct PushpinNeighbor {
  NeighborSize size;        // Number of bytes used for this neighbor.
  PushpinLocalID id;        // This neighbor's local network ID.
  unsigned char bitFlags;   // Housekeeping bit flags used by Bertha.
  ListElement synopsis;     // First element of neighbor's synopsis.
} Neighbor;
typedef struct BerthaPacket {
  PushpinLocalID toAddress; // Intended recipient's neighborhood ID.
  PushpinLocalID fromAddress; // Sender's neighborhood ID.
  unsigned char packetType; // Classification of this packet.
  XdataAddress contentSize; // The number of bytes of content.
  unsigned char CRC;      // Cyclic redundancy check sum.
  unsigned char xdata * contentPtr; // A pointer to the content.
} Packet;
#define PACKETHEADERSIZE 6
#define GLOBALADDRESS 0

/* Internal RAM Layout (256 bytes)
 *
 *  -------------------------------------------------
 *  |  Stack                   Addresses 192 - 255   |
 *  |   (automatically allocated by the compiler)    |
 *  |------------------------------------------------|
 *  |  Active PFrag State      Addresses 190 - 191   |
 *  |     Pointer                                    |
 *  |------------------------------------------------|
 *  |  Active PFrag Local      Addresses 32 - 189    |
 *  |     Variables                                  |
 *  |------------------------------------------------|
 *  |  Registers               Addresses 0 - 31      |
 *  -------------------------------------------------
 */
/* A two-byte external memory pointer. */
#define ActivePFragStatePointerAddress 190


/* External RAM Layout (2Kbytes)
 *  -------------------------------------------------
 *  |  Reentrant Stack         Addresses 1998 - 2047  |
 *  |------------------------------------------------|
 *  |  Neighborhood Watch      Addresses 1152 - 1997  |
 *  |------------------------------------------------|
 *  |  Bulletin Board System   Addresses 576 - 1151   |
 *  |------------------------------------------------|
 *  |  PFrag State Table       Addresses 128 - 575    |
 *  |------------------------------------------------|
 *  |  OS Local Variables      Addresses 0 - 127      |
 *  -------------------------------------------------
 */
/* A pfrag's state is stored within the 2Kbytes of external RAM. */
#define maxPFragStateTableSize 448
/* Limit size specification to one byte. */
#define maxPFragStateSize 256
/* Amount of RAM available for pfrags to exchange information. */
#define maxBBSSize 576
/* One large message could fill the BBS. */
#define maxBBSPostSize maxBBSSize
/* All neighbor information must fit in here. */
#define maxNWSize 846
/* Indicates end of the state table. */
#define NoMorePFrags 0xFF
/* A PFrag with this UID is not valid. */
#define NullPFragUID 0xFFFF
/* No linked list can be located at '0'!*/
/* Location of the table containing all PFrag states. */
#define PFragStateTableAddress 0x80
/* PFragStateTableAddress + PFragStateTableSize */
#define BBSAddress 0x240
/* BBSAddress + maxBBSSize */
#define NWAddress 0x480

/* Program Memory Layout (32Kbytes Flash)
 *
 *  -------------------------------------------------
 *  |  Random Seed             Addresses 30768 - 32889 |
 *  |------------------------------------------------|
 *  |  Reserved                Addresses 32256 - 32767 |
 *  |     (Security Lock)                            |
 *  |------------------------------------------------|
 *  |  Lookup Tables          Addresses 31232 - 32255  |
 *  |    (codecs, CRC, etc.)                           |
 *  |--------------------------------------------------|
 *  |  System Call Table      Addresses 30720 - 31231  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #9      Addresses 28672 - 30719   |
 *  |--------------------------------------------------|
 *  |  Code Fragment #8       Addresses 26624 - 28671  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #7       Addresses 24576 - 26623  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #6       Addresses 22528 - 24575  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #5       Addresses 20480 - 22527  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #4       Addresses 18432 - 20479  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #3       Addresses 16384 - 18431  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #2       Addresses 14336 - 16383  |
 *  |--------------------------------------------------|
 *  |  Code Fragment #1       Addresses 12288 - 14335  |
 *  |--------------------------------------------------|
 *  |  Operating System       Addresses 0000 - 12287   |
 *  ----------------------------------------------------
 */

#define RandomSeedAddress 0x8000
#define RandomSeedSize 64 // Two bytes per int implies 128 byte seed.

// Bit twiddling macros for Neighbor.bitFlags.
#define DEAD (0x01)       // This neighbor should be removed if set.

#define isNeighborDead(n) \
  ((((Neighbor xdata *) n) -> bitFlags) & DEAD)
#define setNeighborDead(n) \
  ((((Neighbor xdata *) n) -> bitFlags) |= DEAD)
#define clearNeighborDead(n) \
  ((((Neighbor xdata *) n) -> bitFlags) &= ~DEAD)

// Bit twiddling macros for PFragState.bitFlags.
// This is a valid PFrag.
#define VALID (0x01)
// This contains a completely received, but unvalidated PFrag.
#define PFRAG (0x02)
// A new PFrag is coming in here over the comm channel.
#define INTRANSIT (0x04)
// This PFrag has been installed.
#define INSTALLED (0x08)
// This PFrag should be deleted.
#define READYTOREMOVE (0x10)
// This PFrag came with a valid state.
#define STATEINITIALIZED (0x20)

#define isPFragValid(id) (PFragStatus[id] & VALID)
#define setPFragValid(id) (PFragStatus[id] |= VALID)
#define clearPFragValid(id) (PFragStatus[id] &= ~VALID)

#define isPFrag(id) (PFragStatus[id] & PFRAG)
#define setPFrag(id) (PFragStatus[id] |= PFRAG)
#define clearPFrag(id) (PFragStatus[id] &= ~PFRAG)
```

```
#define isPFragInTransit(id) (PFragStatus[id] & INTRANSIT)
#define setPFragInTransit(id) (PFragStatus[id] |= INTRANSIT)
#define clearPFragInTransit(id) (PFragStatus[id] &= ~INTRANSIT)

#define isPFragInstalled(id) (PFragStatus[id] & INSTALLED)
#define setPFragInstalled(id) (PFragStatus[id] |= INSTALLED)
#define clearPFragInstalled(id) (PFragStatus[id] &= ~INSTALLED)

#define isPFragRemoveReady(id) (PFragStatus[id] & READYTOREMOVE)
#define setPFragRemoveReady(id) (PFragStatus[id] |= READYTOREMOVE)
#define clearPFragRemoveReady(id) (PFragStatus[id] &= ~READYTOREMOVE)

#define isPFragStateInitialized(id) \
  (PFragStatus[id] & STATEINITIALIZED)
#define setPFragStateInitialized(id) \
  (PFragStatus[id] |= STATEINITIALIZED)
#define clearPFragStateInitialized(id) \
  (PFragStatus[id] &= ~STATEINITIALIZED)


// SFR extra bits and redefinitions.
sfr16 timer3reload = 0x92;
sfr16 timer3 = 0x94;
#define isBuildingNeighborhood() (PCON & 0x04)
#define setBuildingNeighborhood() (PCON |= 0x04)
#define clearBuildingNeighborhood() (PCON &= ~(0x04))
#define isPacketInQueue() (PCON & 0x08)
#define setPacketInQueue() (PCON |= 0x08)
#define clearPacketInQueue() (PCON &= ~(0x08))
#define isChannelClear() (PCON & 0x10)
#define setChannelClear() (PCON |= 0x10)
#define clearChannelClear() (PCON &= ~(0x10))
#define isHighAlarmArmed() (PCON & 0x20)
#define setHighAlarmArmed() (PCON |= 0x20)
#define clearHighAlarmArmed() (PCON &= ~(0x20))
#define isNWBusy() (PCON & 0x40)
#define setNWBusy() (PCON |= 0x40)
#define clearNWBusy() (PCON &= ~(0x40))

// Programmable Counter Array functions.
#define enableAllPCAInterrupts() EIE1 |= 0x08; PCA0MD |= 0x01
#define disableAllPCAInterrupts() EIE1 &= ~(0x08); PCA0MD &= ~(0x01)
#define clearPCAInterruptFlag() CF = 0;
#define clearPCA4InterruptFlag() CCF4 = 0
#define enablePCA4Interrupts() PCA0CPM4 |= 0x01
#define disablePCA4Interrupts() PCA0CPM4 &= ~(0x01)
#define enablePCA4MatchInterrupt() PCA0CPM4 |= 0x08
#define disablePCA4MatchInterrupt() PCA0CPM4 &= ~(0x08)

// Flash memory functions.
#define enableFlashErase() PSCTL = 0x03
#define disableFlashErase() PSCTL = 0x00
#define enableFlashWrite() PSCTL = 0x01
#define disableFlashWrite() PSCTL = 0x00

// Timer 0 functions.
#define enableTimer0() TR0 = 1
#define disableTimer0() TR0 = 0
#define enableTimer0Interrupt() ET0 = 1
#define disableTimer0Interrupt() ET0 = 0
```

```
#define clearTimer0InterruptFlag() TF0 = 0
#define timer0FromSystemClock() CKCON |= 0x08
#define timer0FromSystemClockDiv12() CKCON &= ~(0x08)

// Timer 1 functions.
#define enableTimer1() TR1 = 1
#define disableTimer1() TR1 = 0
#define enableTimer1Interrupt() ET1 = 1
#define disableTimer1Interrupt() ET1 = 0
#define clearTimer1InterruptFlag() TF1 = 0

// Timer 2 functions.
#define enableTimer2() TR2 = 1
#define disableTimer2() TR2 = 0
#define enableTimer2Interrupt() ET2 = 1
#define disableTimer2Interrupt() ET2 = 0
#define clearTimer2InterruptFlag() TF2 = 0

// Timer 3 functions.
#define enableTimer3() TMR3CN |= 0x04
#define disableTimer3() TMR3CN &= ~(0x04)
#define enableTimer3Interrupt() EIE2 |= 0x01
#define disableTimer3Interrupt() EIE2 &= ~(0x01)
#define clearTimer3InterruptFlag() TMR3CN &= ~(0x80)
#define timer3FromSystemClock() TMR3CN |= 0x02
#define timer3FromSystemClockDiv12() TMR3CN &= ~(0x02)

// General interrupt functions.
#define enableGlobalInterrupts() EA = 1
#define disableGlobalInterrupts() EA = 0

// Serial UART functions.
#define enableSerialInterrupts() ES = 1
#define disableSerialInterrupts() ES = 0
#define enableSerialReceive() REN = 1
#define disableSerialReceive() REN = 0
#define setTransmitInterruptFlag() TI = 1
#define clearTransmitInterruptFlag() TI = 0
#define clearReceiveInterruptFlag() RI = 0
#define listenToAddressBytes() SM2 = 1
#define listenToAllBytes() SM2 = 0
#define setUARTMode0() SM0 = 0; SM1 = 0
#define setUARTMode1() SM0 = 0; SM1 = 1
#define setUARTMode2() SM0 = 1; SM1 = 0
#define setUARTMode3() SM0 = 1; SM1 = 1
#define sendAddressBytes() TB8 = 1
#define sendDataBytes() TB8 = 0

// meme : may not need these.
// Serial Peripheral Interface functions.
#define disableSPI() SPIEN = 0
#define enableSPIInterrupt() EIE1 |= 0x01
#define disableSPIInterrupt() EIE1 &= ~(0x01)
#define setSPIInterruptFlag() SPIF = 1
#define clearSPIInterruptFlag() SPIF = 0
```

D.2 Processor-specific Header File

```
/*---------------------------------------------------------------------------
; Copyright (C) 2001 CYGNAL INTEGRATED PRODUCTS, INC.
;   All rights reserved.
;
;
; FILE NAME   : C8051F000.h
; TARGET MCUs : C8051F000, 'F001, 'F002, 'F010, 'F011, 'F012,
;               'F005, 'F006, 'F007, 'F015, 'F016 and 'F017
; DESCRIPTION : Register/bit definitions for the C8051Fxxx family.
;
; REVISION 1.9
;---------------------------------------------------------------------------*/

/*  BYTE Registers  */
sfr P0       = 0x80; /* PORT 0                                          */
sfr SP       = 0x81; /* STACK POINTER                                   */
sfr DPL      = 0x82; /* DATA POINTER - LOW BYTE                         */
sfr DPH      = 0x83; /* DATA POINTER - HIGH BYTE                        */
sfr PCON     = 0x87; /* POWER CONTROL                                   */
sfr TCON     = 0x88; /* TIMER CONTROL                                   */
sfr TMOD     = 0x89; /* TIMER MODE                                      */
sfr TL0      = 0x8A; /* TIMER 0 - LOW BYTE                              */
sfr TL1      = 0x8B; /* TIMER 1 - LOW BYTE                              */
sfr TH0      = 0x8C; /* TIMER 0 - HIGH BYTE                             */
sfr TH1      = 0x8D; /* TIMER 1 - HIGH BYTE                             */
sfr CKCON    = 0x8E; /* CLOCK CONTROL                                   */
sfr PSCTL    = 0x8F; /* PROGRAM STORE R/W CONTROL                       */
sfr P1       = 0x90; /* PORT 1                                          */
sfr TMR3CN   = 0x91; /* TIMER 3 CONTROL                                 */
sfr TMR3RLL  = 0x92; /* TIMER 3 RELOAD REGISTER - LOW BYTE              */
sfr TMR3RLH  = 0x93; /* TIMER 3 RELOAD REGISTER - HIGH BYTE             */
sfr TMR3L    = 0x94; /* TIMER 3 - LOW BYTE                              */
sfr TMR3H    = 0x95; /* TIMER 3 - HIGH BYTE                             */
sfr SCON     = 0x98; /* SERIAL PORT CONTROL                             */
sfr SBUF     = 0x99; /* SERIAL PORT BUFFER                              */
sfr SPI0CFG  = 0x9A; /* SERIAL PERIPHERAL INTERFACE 0
                          CONFIGURATION                                 */
sfr SPI0DAT  = 0x9B; /* SERIAL PERIPHERAL INTERFACE 0 DATA              */
sfr SPI0CKR  = 0x9D; /* SERIAL PERIPHERAL INTERFACE 0 CLOCK RATE
                          CONTROL                                       */
sfr CPT0CN   = 0x9E; /* COMPARATOR 0 CONTROL                            */
sfr CPT1CN   = 0x9F; /* COMPARATOR 1 CONTROL                            */
sfr P2       = 0xA0; /* PORT 2                                          */
sfr PRT0CF   = 0xA4; /* PORT 0 CONFIGURATION                            */
sfr PRT1CF   = 0xA5; /* PORT 1 CONFIGURATION                            */
sfr PRT2CF   = 0xA6; /* PORT 2 CONFIGURATION                            */
sfr PRT3CF   = 0xA7; /* PORT 3 CONFIGURATION                            */
sfr IE       = 0xA8; /* INTERRUPT ENABLE                                */
sfr PRT1IF   = 0xAD; /* PORT 1 EXTERNAL INTERRUPT FLAGS                 */
sfr EMI0CN   = 0xAF; /* EXTERNAL MEMORY INTERFACE CONTROL               */
sfr P3       = 0xB0; /* PORT 3                                          */
sfr OSCXCN   = 0xB1; /* EXTERNAL OSCILLATOR CONTROL                     */
sfr OSCICN   = 0xB2; /* INTERNAL OSCILLATOR CONTROL                     */
sfr FLSCL    = 0xB6; /* FLASH MEMORY TIMING PRESCALER                   */
sfr FLACL    = 0xB7; /* FLASH ACESS LIMIT                               */
sfr IP       = 0xB8; /* INTERRUPT PRIORITY                              */
sfr AMX0CF   = 0xBA; /* ADC 0 MUX CONFIGURATION                         */
sfr AMX0SL   = 0xBB; /* ADC 0 MUX CHANNEL SELECTION                     */
sfr ADC0CF   = 0xBC; /* ADC 0 CONFIGURATION                             */
sfr ADC0L    = 0xBE; /* ADC 0 DATA - LOW BYTE                           */
sfr ADC0H    = 0xBF; /* ADC 0 DATA - HIGH BYTE                          */
sfr SMB0CN   = 0xC0; /* SMBUS 0 CONTROL                                 */
sfr SMB0STA  = 0xC1; /* SMBUS 0 STATUS                                  */
sfr SMB0DAT  = 0xC2; /* SMBUS 0 DATA                                    */
sfr SMB0ADR  = 0xC3; /* SMBUS 0 SLAVE ADDRESS                           */
sfr ADC0GTL  = 0xC4; /* ADC 0 GREATER-THAN REGISTER - LOW BYTE          */
sfr ADC0GTH  = 0xC5; /* ADC 0 GREATER-THAN REGISTER - HIGH BYTE         */
sfr ADC0LTL  = 0xC6; /* ADC 0 LESS-THAN REGISTER - LOW BYTE             */
sfr ADC0LTH  = 0xC7; /* ADC 0 LESS-THAN REGISTER - HIGH BYTE            */
sfr T2CON    = 0xC8; /* TIMER 2 CONTROL                                 */
sfr RCAP2L   = 0xCA; /* TIMER 2 CAPTURE REGISTER - LOW BYTE             */
sfr RCAP2H   = 0xCB; /* TIMER 2 CAPTURE REGISTER - HIGH BYTE            */
sfr TL2      = 0xCC; /* TIMER 2 - LOW BYTE                              */
sfr TH2      = 0xCD; /* TIMER 2 - HIGH BYTE                             */
sfr SMB0CR   = 0xCF; /* SMBUS 0 CLOCK RATE                              */
sfr PSW      = 0xD0; /* PROGRAM STATUS WORD                             */
sfr REF0CN   = 0xD1; /* VOLTAGE REFERENCE 0 CONTROL                     */
sfr DAC0L    = 0xD2; /* DAC 0 REGISTER - LOW BYTE                       */
sfr DAC0H    = 0xD3; /* DAC 0 REGISTER - HIGH BYTE                      */
sfr DAC0CN   = 0xD4; /* DAC 0 CONTROL                                   */
sfr DAC1L    = 0xD5; /* DAC 1 REGISTER - LOW BYTE                       */
sfr DAC1H    = 0xD6; /* DAC 1 REGISTER - HIGH BYTE                      */
sfr DAC1CN   = 0xD7; /* DAC 1 CONTROL                                   */
sfr PCA0CN   = 0xD8; /* PCA 0 COUNTER CONTROL                           */
sfr PCA0MD   = 0xD9; /* PCA 0 COUNTER MODE                              */
sfr PCA0CPM0 = 0xDA; /* CONTROL REGISTER FOR PCA 0 MODULE 0             */
sfr PCA0CPM1 = 0xDB; /* CONTROL REGISTER FOR PCA 0 MODULE 1             */
sfr PCA0CPM2 = 0xDC; /* CONTROL REGISTER FOR PCA 0 MODULE 2             */
sfr PCA0CPM3 = 0xDD; /* CONTROL REGISTER FOR PCA 0 MODULE 3             */
sfr PCA0CPM4 = 0xDE; /* CONTROL REGISTER FOR PCA 0 MODULE 4             */
sfr ACC      = 0xE0; /* ACCUMULATOR                                     */
sfr XBR0     = 0xE1; /* DIGITAL CROSSBAR CONFIGURATION REGISTER 0       */
sfr XBR1     = 0xE2; /* DIGITAL CROSSBAR CONFIGURATION REGISTER 1       */
sfr XBR2     = 0xE3; /* DIGITAL CROSSBAR CONFIGURATION REGISTER 2       */
sfr EIE1     = 0xE6; /* EXTERNAL INTERRUPT ENABLE 1                     */
sfr EIE2     = 0xE7; /* EXTERNAL INTERRUPT ENABLE 2                     */
sfr ADC0CN   = 0xE8; /* ADC 0 CONTROL                                   */
sfr PCA0L    = 0xE9; /* PCA 0 TIMER - LOW BYTE                          */
sfr PCA0CPL0 = 0xEA; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 0 - LOW BYTE                           */
sfr PCA0CPL1 = 0xEB; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 1 - LOW BYTE                           */
sfr PCA0CPL2 = 0xEC; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 2 - LOW BYTE                           */
sfr PCA0CPL3 = 0xED; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 3 - LOW BYTE                           */
sfr PCA0CPL4 = 0xEE; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 4 - LOW BYTE                           */
sfr RSTSRC   = 0xEF; /* RESET SOURCE                                    */
sfr B        = 0xF0; /* B REGISTER                                      */
sfr EIP1     = 0xF6; /* EXTERNAL INTERRUPT PRIORITY REGISTER 1          */
sfr EIP2     = 0xF7; /* EXTERNAL INTERRUPT PRIORITY REGISTER 2          */
sfr SPI0CN   = 0xF8; /* SERIAL PERIPHERAL INTERFACE 0 CONTROL           */
sfr PCA0H    = 0xF9; /* PCA 0 TIMER - HIGH BYTE                         */
sfr PCA0CPH0 = 0xFA; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 0 - HIGH BYTE                          */
sfr PCA0CPH1 = 0xFB; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                          MODULE 1 - HIGH BYTE                          */
sfr PCA0CPH2 = 0xFC; /* CAPTURE/COMPARE REGISTER FOR PCA 0
sfr PCA0CPH3 =  0xFD; /* CAPTURE/COMPARE REGISTER FOR PCA 0

                           MODULE 2 - HIGH BYTE                    */



MODULE
 3 -
HIGH

BYTE


*/
sfr PCA0CPH4 =  0xFE; /* CAPTURE/COMPARE REGISTER FOR PCA 0
                           MODULE 4 - HIGH BYTE                    */
sfr WDTCN    =  0xFF; /* WATCHDOG TIMER CONTROL                    */


/*  BIT Registers  */

/*  TCON  0x88 */
sbit TF1   = TCON ^ 7; /* TIMER 1 OVERFLOW FLAG                    */
sbit TR1   = TCON ^ 6; /* TIMER 1 ON/OFF CONTROL                   */
sbit TF0   = TCON ^ 5; /* TIMER 0 OVERFLOW FLAG                    */
sbit TR0   = TCON ^ 4; /* TIMER 0 ON/OFF CONTROL                   */
sbit IE1   = TCON ^ 3; /* EXT. INTERRUPT 1 EDGE FLAG               */
sbit IT1   = TCON ^ 2; /* EXT. INTERRUPT 1 TYPE                    */
sbit IE0   = TCON ^ 1; /* EXT. INTERRUPT 0 EDGE FLAG               */
sbit IT0   = TCON ^ 0; /* EXT. INTERRUPT 0 TYPE                    */

/*  SCON  0x98 */
sbit SM0   = SCON ^ 7; /* SERIAL MODE CONTROL BIT 0                */
sbit SM1   = SCON ^ 6; /* SERIAL MODE CONTROL BIT 1                */
sbit SM2   = SCON ^ 5; /* MULTIPROCESSOR COMMUNICATION ENABLE      */
sbit REN   = SCON ^ 4; /* RECEIVE ENABLE                           */
sbit TB8   = SCON ^ 3; /* TRANSMIT BIT 8                           */
sbit RB8   = SCON ^ 2; /* RECEIVE BIT 8                            */
sbit TI    = SCON ^ 1; /* TRANSMIT INTERRUPT FLAG                  */
sbit RI    = SCON ^ 0; /* RECEIVE INTERRUPT FLAG                   */

/*  IE  0xA8 */
sbit EA    = IE ^ 7;   /* GLOBAL INTERRUPT ENABLE                  */
sbit ET2   = IE ^ 5;   /* TIMER 2 INTERRUPT ENABLE                 */
sbit ES    = IE ^ 4;   /* SERIAL PORT INTERRUPT ENABLE             */
sbit ET1   = IE ^ 3;   /* TIMER 1 INTERRUPT ENABLE                 */
sbit EX1   = IE ^ 2;   /* EXTERNAL INTERRUPT 1 ENABLE              */
sbit ET0   = IE ^ 1;   /* TIMER 0 INTERRUPT ENABLE                 */
sbit EX0   = IE ^ 0;   /* EXTERNAL INTERRUPT 0 ENABLE              */

/*  IP  0xB8 */
sbit PT2   = IP ^ 5;   /* TIMER 2 PRIORITY                         */
sbit PS    = IP ^ 4;   /* SERIAL PORT PRIORITY                     */
sbit PT1   = IP ^ 3;   /* TIMER 1 PRIORITY                         */
sbit PX1   = IP ^ 2;   /* EXTERNAL INTERRUPT 1 PRIORITY            */
sbit PT0   = IP ^ 1;   /* TIMER 0 PRIORITY                         */
sbit PX0   = IP ^ 0;   /* EXTERNAL INTERRUPT 0 PRIORITY            */

/* SMB0CN 0xC0 */
sbit BUSY     =  SMB0CN ^ 7;  /* SMBUS 0 BUSY                      */
sbit ENSMB    =  SMB0CN ^ 6;  /* SMBUS 0 ENABLE                    */
sbit STA      =  SMB0CN ^ 5;  /* SMBUS 0 START FLAG                */
sbit STO      =  SMB0CN ^ 4;  /* SMBUS 0 STOP FLAG                 */
sbit SI       =  SMB0CN ^ 3;  /* SMBUS 0 INTERRUPT PENDING FLAG    */
sbit AA       =  SMB0CN ^ 2;  /* SMBUS 0 ASSERT/ACKNOWLEDGE FLAG   */
sbit SMBFTE   =  SMB0CN ^ 1;  /* SMBUS 0 FREE TIMER ENABLE         */
sbit SMBTOE   =  SMB0CN ^ 0;  /* SMBUS 0 TIMEOUT ENABLE            */

/*  T2CON  0xC8 */
sbit TF2   = T2CON ^ 7;       /* TIMER 2 OVERFLOW FLAG             */
sbit EXF2  = T2CON ^ 6;       /* EXTERNAL FLAG                     */
sbit RCLK  = T2CON ^ 5;       /* RECEIVE CLOCK FLAG                */
sbit TCLK  = T2CON ^ 4;       /* TRANSMIT CLOCK FLAG               */
sbit EXEN2 = T2CON ^ 3;       /* TIMER 2 EXTERNAL ENABLE FLAG      */
sbit TR2   = T2CON ^ 2;       /* TIMER 2 ON/OFF CONTROL            */
sbit CT2   = T2CON ^ 1;       /* TIMER OR COUNTER SELECT           */
sbit CPRL2 = T2CON ^ 0;       /* CAPTURE OR RELOAD SELECT          */

/*  PSW  */
sbit CY    = PSW ^ 7;  /* CARRY FLAG                               */
sbit AC    = PSW ^ 6;  /* AUXILIARY CARRY FLAG                     */
sbit F0    = PSW ^ 5;  /* USER FLAG 0                              */
sbit RS1   = PSW ^ 4;  /* REGISTER BANK SELECT 1                   */
sbit RS0   = PSW ^ 3;  /* REGISTER BANK SELECT 0                   */
sbit OV    = PSW ^ 2;  /* OVERFLOW FLAG                            */
sbit F1    = PSW ^ 1;  /* USER FLAG 1                              */
sbit P     = PSW ^ 0;  /* ACCUMULATOR PARITY FLAG                  */

/* PCA0CN D8H */
sbit CF    =  PCA0CN ^ 7;     /* PCA 0 COUNTER OVERFLOW FLAG       */
sbit CR    =  PCA0CN ^ 6;     /* PCA 0 COUNTER RUN CONTROL BIT     */
sbit CCF4  =  PCA0CN ^ 4;     /* PCA 0 MODULE 4 INTERRUPT FLAG     */
sbit CCF3  =  PCA0CN ^ 3;     /* PCA 0 MODULE 3 INTERRUPT FLAG     */
sbit CCF2  =  PCA0CN ^ 2;     /* PCA 0 MODULE 2 INTERRUPT FLAG     */
sbit CCF1  =  PCA0CN ^ 1;     /* PCA 0 MODULE 1 INTERRUPT FLAG     */
sbit CCF0  =  PCA0CN ^ 0;     /* PCA 0 MODULE 0 INTERRUPT FLAG     */

/* ADC0CN E8H */
sbit ADCEN    =  ADC0CN ^ 7;  /* ADC 0 ENABLE                      */
sbit ADCTM    =  ADC0CN ^ 6;  /* ADC 0 TRACK MODE                  */
sbit ADCINT   =  ADC0CN ^ 5;  /* ADC 0 CONVERISION COMPLETE
                                   INTERRUPT FLAG                  */
sbit ADBUSY   =  ADC0CN ^ 4;  /* ADC 0 BUSY FLAG                   */
sbit ADSTM1   =  ADC0CN ^ 3;  /* ADC 0 START OF CONVERSION MODE
                                   BIT 1                           */
sbit ADSTM0   =  ADC0CN ^ 2;  /* ADC 0 START OF CONVERSION MODE
                                   BIT 0                           */
sbit ADWINT   =  ADC0CN ^ 1;  /* ADC 0 WINDOW COMPARE INTERRUPT
                                   FLAG                            */
sbit ADLJST   =  ADC0CN ^ 0;  /* ADC 0 RIGHT JUSTIFY DATA BIT      */

/* SPI0CN F8H */
sbit SPIF     =  SPI0CN ^ 7;  /* SPI 0 INTERRUPT FLAG              */
sbit WCOL     =  SPI0CN ^ 6;  /* SPI 0 WRITE COLLISION FLAG        */
sbit MODF     =  SPI0CN ^ 5;  /* SPI 0 MODE FAULT FLAG             */
sbit RXOVRN   =  SPI0CN ^ 4;  /* SPI 0 RX OVERRUN FLAG             */
sbit TXBSY    =  SPI0CN ^ 3;  /* SPI 0 TX BUSY FLAG                */
sbit SLVSEL   =  SPI0CN ^ 2;  /* SPI 0 SLAVE SELECT                */
sbit MSTEN    =  SPI0CN ^ 1;  /* SPI 0 MASTER ENABLE               */
sbit SPIEN    =  SPI0CN ^ 0;  /* SPI 0 SPI ENABLE                  */
```

D.3 Pseudo-random Number Generator Seed

```
#include <Bertha.h>

signed int code RandomSeed[RandomSeedSize] = {
  0x4158,0x4024,0x0958,0x4520,0x4948,0x5726,0x7740,0x1103,0x4594,
  0x8572,0x0456,0x9468,0x4750,0x0928,0x4389,0x3932,0x0209,0x3475,
  0x0847,0x6274,0x1243,0x6598,0x7457,0x3506,0x9872,0x9292,0x9294,
  0x3847,0x3810,0x1395,0x4828,0x7457,0x4529,0x8489,0x2896,0x4871,
  0x8713,0x4986,0x1571,0x1359,0x8136,0x4334,0x1032,0x8584,0x8303,
  0x0328,0x5756,0x6482,0x9821,0x0103,0x8467,0x2518,0x7305,0x0392,
  0x8772,0x6251,0x9222,0x2829,0x1874,0x3874,0x8820,0x8436,0x2001,
  0x2592
};
```

D.4 Global Variables

```
extern void code * code SystemCallTable[maxNumberOfSystemFunctions];
extern signed int code RandomSeed[RandomSeedSize];
extern void xdata * idata ActivePFragStatePointer;
extern ReservedMemoryArea xdata PFragStateTable[maxPFragStateTableSize];
extern ReservedMemoryArea xdata BBS[maxBBSSize];
extern ReservedMemoryArea xdata NW[maxNWSize];
extern PFrag code * code PFragPointers[maxNumberOfPFrags];
extern Packet xdata receivePacket;
extern Packet xdata transmitPacket;
extern unsigned char xdata serialState;
extern unsigned char xdata commMode;
extern PushpinLocalID xdata NeighborhoodID;
extern PFragLocalID xdata currentPFragID;
extern PFragLocalID xdata StateInWaitingID;
extern unsigned char xdata BerthaMessage[BerthaMessageSize];
extern unsigned char xdata PFragStatus[maxNumberOfPFrags];
extern unsigned int xdata timeScale;
extern unsigned long xdata neighborTimeoutAlarm;
extern const unsigned char code CRC8Table[256];
extern const unsigned char code encodeNybbleTable[16];
extern const unsigned char code decodeNybbleTable[256];
```

D.5 Shared Information

```
/****************************************************************
 * BerthaPFragShared.h
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 09MAR2002 by lifton.
 *
 * This file contains all information shared between both Bertha and
 * PFrags.
 ****************************************************************/

/* See BerthaGlobals.c for special linking instructions. */

/* 256 two-byte pointers. */
#define maxNumberOfSystemFunctions 256
/* Hook into system calls for process fragments. */
#define SystemCallTableAddress 0x7800
enum callIDs {
  updateID, installID, deinstallID
};
enum BerthaPostIDs {
  LDRValue, TimeStamp, EightByteTempID
};
```

D.6 Hardware Definitions

```
/****************************************************************
 * PushpinHardwareV3.h revision 0
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2001 all rights reserved.
 *
 * Last modified 30NOV2001 by lifton.
 *
 * Summary:
 * This file specifies the hardware configuration for version 3 of
 * the pushpin architecture.  The underlying processor is the Cygnal
 * C8051F016.  The hardware configuration results from the Cygnal
 * crossbar configuration and should only be changed by licensed
 * professionals.
 ****************************************************************/

/****************************************************************
Port I/O Pin Assignments:

The Pushpin hardware architecture makes use of both pinned out
8-channel I/O ports on the Cygnal C8051F016 (the C8051F0xx family
has four 8-channel I/O ports, but only two are available as actual
pins on the C8051F016).  Some of the port I/O pins are dedicated to
system operation and should or can not be manipulated by the user
and some are meant specifically to support user-added hardware such
as sensors or other microprocessors.
****************************************************************/

/*
 * Port 0.
 */
sbit txPin    = P0^0; // Transmit pin for UART.
sbit rxPin    = P0^1; // Receive pin for UART.
sbit gpio0Pin = P0^2; // General purpose I/O pin on sensor connector.
sbit gpio1Pin = P0^3; // General purpose I/O pin on sensor connector.
sbit gpio2Pin = P0^4; // General purpose I/O pin on sensor connector.
sbit gpio3Pin = P0^5; // General purpose I/O pin on sensor connector.
sbit gpio4Pin = P0^6; // General purpose I/O pin on sensor connector.
sbit gpio5Pin = P0^7; // General purpose I/O pin on sensor connector.

/*
 * Port 1.
 */
// General purpose I/O pin on sensor connector.
sbit gpio6Pin             =  P1^0;
// Shared GPIO pin between comm and sensor connectors.
sbit gpioCommSharedPin    =  P1^1;
// General purpose I/O pin on comm connector.
sbit comm0Pin             =  P1^2;
// Red LED on processor board.
sbit statusLED            =  P1^3;
// General purpose I/O pin w/ interrupt on comm connector.
sbit comm1Pin             =  P1^4;
// General purpose I/O pin w/ interrupt on comm connector.
sbit comm2Pin             =  P1^5;
// General purpose I/O pin w/ interrupt on comm connector.
sbit comm3Pin             =  P1^6;
// General purpose I/O pin w/ interrupt on comm connector.
sbit comm4Pin             =  P1^7;

/*
 * Pins used by the 4-way IR communications module.
 */
sbit IRChannelSelect0 = P1^1;
sbit IRChannelSelect1 = P1^2;
sbit IRChannel0 = P1^4;
sbit IRChannel1 = P1^5;
sbit IRChannel2 = P1^6;
sbit IRChannel3 = P1^7;
/*
 * Pins used by the LDR expansion module.
 */
sbit greenLED = P0^6;
sbit yellowLED = P0^2;
sbit amberLED = P0^3;
sbit orangeLED = P0^4;
sbit redLED = P0^5;
sbit LDR = P0^7;

/*
 * The Keil 8051 C compiler allows two consecutive 8-bit special
 * function registers to be declared and treated as a single
 * 16-bit register.
 */
sfr16 timer2 = 0xCC;
sfr16 timer2reload = 0xCA;
```

D.7 Function Prototypes

```c
/***************************************************************
 * BerthaFunctionPrototypes.h
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 07MAR2002 by lifton.
 *
 * This file contains all function prototypes needed to compile the
 * Bertha operating system. Function prototypes are listed in
 * alphabetical order by function name and grouped according to the
 * file they are defined in.
 ***************************************************************/

// BerthaBBS.c -- Functions to manage the Bulletin Board System.
void initializeBBS(void);
void updateSystemBBSPosts(void);

// BerthaFunctions.c -- Functions available only to Bertha.
void buildNeighborhood(void);
void configurePins(void);
void disableWatchdogTimer(void);
void enableADC(void);
void enableExternalClock(void);
void initializeHardware(void);
void main(void);
void notEnoughPFragStateMemory(PFragLocalID);
void processBerthaMessage(PushpinLocalID);
void setRandomSeed(int);

// BerthaPFrag.c
PFragState xdata * allocatePFragState(PFragLocalID, unsigned char)
  reentrant;
unsigned char allocatePFragLocalID(void);
unsigned char callPFrag(PFragLocalID, PFragFunctionID);
PFragState xdata * getPFragState(PFragLocalID) reentrant;
void initializePFrags(void);
void validatePFrag(PFragLocalID);
void removePFrag(PFragLocalID);
unsigned char removePFragPosts(PFragLocalID);
unsigned char transferPFrag(PFragLocalID, PushpinLocalID);

// BerthaNW.c
void broadcastSynopsis(void);
void initializeNW(void);
void generateNeighborhoodID(void);
void addNeighbor(PushpinLocalID);
Neighbor xdata * getNeighbor(PushpinLocalID) reentrant;
Neighbor xdata * getNthNeighbor(PushpinLocalID);
void removeNeighbor(PushpinLocalID);
unsigned char getNthSynopsisPost(unsigned char, Neighbor xdata *,
  unsigned int, unsigned char data *);
unsigned char getSynopsisPost(BBSPostID, Neighbor xdata *,
  unsigned int, unsigned char data *);
unsigned char isNeighborhoodIDValid(void);
void updateNW(void);

// LinkedList.c
ListElement xdata * addListElement(ListElement, LinkedList,
  XdataAddress) reentrant;
ListElement xdata * getLastListElement(LinkedList) reentrant;
XdataAddress getListSize(LinkedList) reentrant;
ListElement xdata * getNextListElement(ListElement xdata *,
  LinkedList);
unsigned char getNumberOfElements(LinkedList);
unsigned char removeListElement(ListElement xdata *, LinkedList)
  reentrant;
void xdataMemMove(unsigned char xdata *, unsigned char xdata *,
  unsigned int) reentrant;

// SystemFunctions.c -- System calls available to PFrags.
void delay(unsigned int);
void die(void);
void flashLDRLED(unsigned char, unsigned int, unsigned int);
void flashLED(unsigned int, unsigned int) reentrant;
unsigned int getADCValue(unsigned char);
unsigned char getBBSPost(BBSPostID, unsigned int,
  unsigned char data *) reentrant;
unsigned char getBBSPostCount(void);
unsigned char getNeighborCount(void);
unsigned char getNthBBSPost(unsigned char, unsigned int,
  unsigned char data *);
PushpinLocalID getNthNeighborID(PushpinLocalID);
unsigned char getNthPostFromMthSynopsis(unsigned char,
  PushpinLocalID, unsigned char data *);
unsigned char getNthPostFromSynopsis(unsigned char,
  PushpinLocalID, unsigned char data *);
unsigned char getNthSynopsisPostCount(PushpinLocalID);
PFragUID getPFragUID(void);
unsigned char getPostFromMthSynopsis(BBSPostID, PushpinLocalID,
  unsigned char data *);
unsigned char getPostFromSynopsis(BBSPostID, PushpinLocalID,
  unsigned char data *);
unsigned char getSynopsisPostCount(PushpinLocalID);
unsigned char isStateReplicated(void);
unsigned char postToBBS(BBSPostID, unsigned int,
  unsigned char data *);
unsigned int random(void);
unsigned char removeAllPosts(void);
unsigned char removePost(BBSPostID) reentrant;
void setLEDIntensity(unsigned char, unsigned char);
unsigned char transfer(PushpinLocalID);

// BerthaCommunication.c -- Functions to handle communication.
void disableCommunication(void);
void enableCommunication(void);
unsigned char getNextPacketByte(void) reentrant;
unsigned char xdata *  getContentPtr(void);
unsigned char packetCRC8(Packet xdata *) reentrant;
void packetReceived(void);
unsigned char queuePacketForTransmission(PushpinLocalID,
  unsigned char, XdataAddress, unsigned char xdata *) reentrant;
void receivedByte(unsigned char);
void serialInterrupt(void);
void setCommMode(void);
void setNextPacketByte(unsigned char);
void timer2Interrupt(void);

unsigned char attemptToSendPacket(void);


// BerthaTiming.c -- Management of real-time clock and timers.
void ringAlarm(void);
void startRealTimeClock(void);
void setAlarm(unsigned long);
unsigned long getTime(void);
void setTime(unsigned long);
void timer0Interrupt(void);
void timer3Interrupt(void);

// ExpansionModules.c -- Configuration and use of expansion modules.
void enableLDRModule(void);
```

D.8 Top-level OS Loop

```
/****************************************************************
 * BerthaOS.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 30MAY2002 by lifton.
 *
 * This is Bertha, the operating system for Pushpins.
 ****************************************************************/

/****************************************************************
The Life of a Process Fragment -- by Josh Lifton

In the beginning, there is the Pushpin IDE and the user.  The user
may use the Pushpin IDE's text area to compose a new process
fragment from a template or modify a previously saved process
fragment file (.c file).  The user may use the Pushpin IDE to
compile the process fragment into a binary file (.BIN file).  The
user may then use the Pushpin IDE to upload a binary process
fragment file through the serial port of the computer running the
Pushpin IDE to a Pushpin running Bertha.  Bertha's communication
code will detect the incoming process fragment, allocate flash
memory to it, mark that piece of memory as containing a process
fragment in transit, and store it byte by byte as it arrives.
After the entire process fragment has arrived, Bertha will,
assuming the process fragment passes validation, allocate external
RAM to the process fragment's state, remove the process fragment
in transit mark, and mark the process fragment as existing.
Bertha's main loop checks for process fragments marked as existing
and not in transit and attempts to validate those process fragments.
Validation consists checking that the process fragment is stored
within a certain range of flash memory and that the actual code
stored there passes an eight-bit cyclic redundancy check (CRC)
generated originally by the Pushpin IDE during compilation of the
process fragment.  If the process fragment is not valid, Bertha
marks it for deletion.  If the process fragment in question is
valid, its install function is called if it hasn't been called
already.  If the install function has already been called, the
process fragment's update function is called instead.  If the
process fragment is marked for deletion, Bertha then deallocates
both the process fragment and the process fragment's state.  The
process fragment can mark itself for deletion during install or
update.

****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>

extern void code * code SystemCallTable[maxNumberOfSystemFunctions];
void xdata * idata ActivePFragStatePointer
  _at_ ActivePFragStatePointerAddress;
ReservedMemoryArea xdata PFragStateTable[maxPFragStateTableSize]
  _at_ PFragStateTableAddress;
ReservedMemoryArea xdata BBS[maxBBSSize] _at_ BBSAddress;
ReservedMemoryArea xdata NW[maxNWSize] _at_ NWAddress;
PFrag code * code PFragPointers[maxNumberOfPFrags] = {
  12288,    // 0x3000
//  14336,    // 0x3800 meme : just a test
//  16384,    // 0x4000
//  18432,    // 0x4800
//  20480,    // 0x5000
//  22528,    // 0x5800
//  24576,    // 0x6000
//  26624,    // 0x6800
//  28672,    // 0x7000
};
Packet xdata receivePacket;
Packet xdata transmitPacket;
unsigned char xdata serialState = DISABLED;
unsigned char xdata commMode;
PushpinLocalID xdata NeighborhoodID = GLOBALADDRESS;
PFragLocalID xdata currentPFragID = NoMorePFrags;
PFragLocalID xdata StateInWaitingID = NoMorePFrags;
unsigned char xdata BerthaMessage[BerthaMessageSize] = {
  0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff
};
unsigned char xdata PFragStatus[maxNumberOfPFrags];
unsigned int xdata timeScale = DEFAULTTIMESCALE;
unsigned long xdata neighborTimeoutAlarm;

void main(void) {
  /*
   * Local variables used only for the main loop can reside in
   * the data memory space only if they are used outside of
   * PFrag execution.
   */
  // meme : unsigned char data BerthaBBSPost[BerthaBBSPostSize];
  unsigned int data waitTime;

  /*
   * Initialization.
   */
  initializeHardware();
  setRandomSeed(*((unsigned int code *) RandomSeedAddress));
  enableGlobalInterrupts();
  startRealTimeClock();
  enableLDRModule();
  setCommMode();
  neighborTimeoutAlarm = 2*NEIGHBORTIMEOUT;
  /*
   * Indicate on-board initialization complete.
   */
  flashLED(commMode,0xFFFF);
  initializeBBS();
  initializePFrags();
  buildNeighborhood();
  while(!isNeighborhoodIDValid()) {
    generateNeighborhoodID();
  }
  /*
```

```
    */

    * Indicate network initialization complete.



flashLED
(5,35000);

  /* Control loop. */
  while (1) {
    /*
     * Indicate this Pushpin is still alive.
     */
    flashLED(1,0x3FFF);

    /*
     * Update or install and possibly delete each PFrag, starting with
     * the first.  Assumes each incoming PFrag is automatically
     * allocated a PFragState.  Then perform housekeeping for the NW
     * and communication system.
     */
    disableCommunication(); // meme : LinkedList kludge.
    for (currentPFragID = 0;
        currentPFragID < maxNumberOfPFrags;
        currentPFragID++) {
      if (isPFrag(currentPFragID) &&
          !isPFragInTransit(currentPFragID)) {
        validatePFrag(currentPFragID);
        if (isPFragValid(currentPFragID)) {
          if (isPFragInstalled(currentPFragID)) {
            callPFrag(currentPFragID, updateID);
          }
          else {
            callPFrag(currentPFragID, installID);
            setPFragInstalled(currentPFragID);
          }
        }
        else {
          setPFragRemoveReady(currentPFragID);
        }
      }
      if (isPFragRemoveReady(currentPFragID)) {
        callPFrag(currentPFragID, deinstallID);
        removePFrag(currentPFragID);
      }
      if (isPacketInQueue()) {
        attemptToSendPacket();
      }
    }

    if (isPacketInQueue()) {   //
      enableCommunication();   //
      attemptToSendPacket();   // meme : LinkedList kludge.
    }                          //
    else {                     //
      enableCommunication();   //
    }                          //

    /*
     * Make system posts to the BBS.
     */
    updateSystemBBSPosts();

    /*
     * Perform housekeeping tasks to maintain the Neighborhood Watch.
     */
    updateNW();

    /*
     * Do nothing for a random, but minimum amount of time.
     */
    for (waitTime = 0; waitTime < timeScale; waitTime++) {
      delay(DELAYTIME);
    }
    delay(random());

    /*
     * Transmit to all neighbors a summary of the BBS.
     */
    broadcastSynopsis();
  }
}
```

D.9 PFrag Management

```
/****************************************************************
 * BerthaPFrag.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 13MAY2002 by lifton.
 *
 * Functions pertaining to Bertha's maintenance of PFrags.  Much of
 * this maintenance is carried out through manipulation of the
 * 'bitFlags' data held in each PFragState.
 ****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>


/*
 * Should be called after a reset.  Checks flash for valid process
 * fragments and re-installs them.  Erases invalid process fragments.
 */
void initializePFrags(void) {
  XdataAddress xdata index;
  for (index=0; index<maxPFragStateTableSize; index++) {
    PFragStateTable[index] = 0;
  }
  for (index=0; index<maxNumberOfPFrags; index++) {
    /*
     * Clear all status flags and check for valid PFrags.
     */
    PFragStatus[index] = 0;
    if (!allocatePFragState((PFragLocalID) index, 1)) {
      setPFragRemoveReady((PFragLocalID) index);
    }
    if (isPFragValid((PFragLocalID) index)) {
      setPFrag((PFragLocalID) index);
    }
  }
}

/*
 * Allocates an entry of the appropriate size in the PFragStateTable
 * for the state of the PFrag in question.  Returns a pointer to
 * that state.  If the returned pointer is null, the memory was not
 * successfully allocated.
 */
PFragState xdata * allocatePFragState(PFragLocalID id,
    unsigned char isNew) reentrant {
  PFragState xdata * xdata statePtr = 0;
  if (isNew) {
    validatePFrag(id);  // PFrags don't come pre-validated.
    if (isPFragValid(id)) {
      clearPFragStateInitialized(id);
      statePtr = (PFragState xdata *) addListElement(((ListElement)
        (PFragPointers[id] -> stateSize)) +
        sizeof(PFragState) - 1, (LinkedList) PFragStateTable,
        maxPFragStateTableSize);
    }
  }
  else if (id != NoMorePFrags) {
    setPFragStateInitialized(id);
    statePtr = (PFragState xdata *) addListElement((ListElement)
      receivePacket.contentSize,
      (LinkedList) PFragStateTable,
      maxPFragStateTableSize);
  }
  if (statePtr) {
    (statePtr -> id) = id;
  }
  return statePtr;
}


/*
 * This is Bertha's entry point for calling functions defined by
 * PFrags.  This function takes a pointer to a PFrag and a function
 * call ID code as arguments.  If both the PFrag and call ID are
 * valid, that PFrag's function is called and 1 is returned.
 * Otherwise 0 is returned.
 */
unsigned char callPFrag(PFragLocalID pfragID,
    PFragFunctionID callID) {
  if (isPFragValid(pfragID)) {
    ActivePFragStatePointer = &((*getPFragState(pfragID)).state);
    if (ActivePFragStatePointer) {
      (((unsigned int (*)(unsigned char,unsigned int,unsigned int))
        (&((*(PFragPointers[pfragID])).codeFrag))) (callID));
      return 1;
    }
  }
  else {
    return 0;
  }
}

/*
 * Currently, just checks to make sure the PFrag isn't null and that
 * it passes its checksum.  It is unfortunate that a PFrag consisting
 * of 0x00 followed by all 0xFF will actually pass the CRC.  Hence,
 * the added requirement that the PFrag's UID not be equal to the
 * NullPFragUID.
 */
void validatePFrag(PFragLocalID id) {
  unsigned char crc = 0;
  unsigned int i;
  if ((id < maxNumberOfPFrags) &&
      (id >= 0) &&
      (PFragPointers[id] -> size) &&
      (PFragPointers[id] -> size < maxPFragSize) &&
      (PFragPointers[id] -> uid != NullPFragUID)) {
    crc = CRC8Table[crc ^ (0xFF &
      ((*(PFragPointers[id])).size >> 8))];
      (*(PFragPointers[id])).size)];

    crc = CRC8Table[crc ^ (0xFF &



crc
 =
CRC8Table
[
crc
 ^ (0
xFF
 &
      ((*(PFragPointers[id])).uid >> 8))];
    crc = CRC8Table[crc ^ (0xFF &
      (*(PFragPointers[id])).uid)];
    crc = CRC8Table[crc ^ (0xFF &
      ((*(PFragPointers[id])).stateSize >> 8))];
    crc = CRC8Table[crc ^ (0xFF &
      (*(PFragPointers[id])).stateSize)];
    for (i= (((unsigned char xdata *)
        &((*(PFragPointers[id])).codeFrag)) -
        ((unsigned char xdata *) PFragPointers[id]));
        i < (*(PFragPointers[id])).size; i++) {
      crc = CRC8Table[crc ^
        *(((unsigned char code *) PFragPointers[id]) + i)];
    }
    if ((*(PFragPointers[id])).crc == crc) {
      setPFragValid(id);
    }
  }
  else {
    clearPFragValid(id);
  }
}

/*
 * Removes all information associated with a given PFrag and, if
 * need be, clears the code memory associated with that PFrag.
 * This function should only be called from Bertha's main loop
 * through the use of semaphores, if need be.
 */
void removePFrag(PFragLocalID id) {
  unsigned char xdata * xdata codePtr;
  removePFragPosts(id);
  removeListElement((ListElement xdata *) getPFragState(id),
    (LinkedList) &PFragStateTable);
  if (id < maxNumberOfPFrags) {
    /*
     * Wait for communication to cease and then disable further
     * communication and start removing the PFrag.  Reenable
     * communication afterward.  meme : communication may not
     * be enabled initially.
     */
    if (serialState != DISABLED) {
      while ((serialState != IDLE) && (serialState != SCANNING));
      disableCommunication();
      serialState = STASIS;
    }
    PFragStatus[id] = 0;
```

```
    for (codePtr = ((unsigned char xdata *) PFragPointers[id]);
       codePtr < ((unsigned char xdata *) PFragPointers[id]) +
        maxPFragSize;
       codePtr++) {
      if (*((unsigned char code *) codePtr) != ErasedPFragChar) {
        /*
         * Point to xdata so as to ensure using MOVX instruction to
         * erase flash.  First, must disable interrupts to avoid
         * extra MOVX executions.
         */
        disableGlobalInterrupts();
        enableFlashErase();
        *codePtr = 0x00;
        disableFlashErase();
        enableGlobalInterrupts();
      }
    }
    if (serialState == STASIS) {
      enableCommunication();
    }
  }
}

/*
 * Removes all posts created by the PFrag with a local id
 * matching the function argument.  Returns the number of
 * posts removed.
 */
unsigned char removePFragPosts(PFragLocalID id) {
  unsigned char xdata numOfPostsRemoved = 0;
  BBSPost xdata * xdata postPtr =
    (BBSPost xdata *) getNextListElement(0, (LinkedList) &BBS);
  while (*((ListElement xdata *) postPtr)) {
    if ((postPtr -> localID) == id) {
      if (removeListElement((ListElement xdata *) postPtr,
                            (LinkedList) &BBS)) {
        numOfPostsRemoved++;
      }
    }
    postPtr = (BBSPost xdata *) getNextListElement(
                                  (ListElement xdata *) postPtr,
                                  (LinkedList) &BBS);
  }
  return numOfPostsRemoved;
}

/*
 * Returns a pointer to the state of the process fragment identified
 * with PFragID. Returns 0 if PFragID is not a valid PFragID or if
 * there is no state corresponding to the process fragment ID in
 * question.
 */
PFragState xdata * getPFragState(PFragLocalID id) reentrant {
  ListElement xdata * xdata statePtr = getNextListElement(0,
    (LinkedList) &PFragStateTable);
  while (*statePtr) {
    if ((*((PFragState xdata *) statePtr)).id == id) {
      break;
    }
    else {
      statePtr = getNextListElement((ListElement xdata *) statePtr,
        (LinkedList) &PFragStateTable);
    }
  }
  if (*statePtr == 0) {
    return 0;   // No PFragState with given ID.
  }
  return statePtr;
}

/*
 * Allocates a random process fragment ID guaranteed not to be in use
 * by another process fragment.  If no more IDs are available,
 * returns NoMorePFrags.
 */
PFragLocalID allocatePFragLocalID(void) {
  PFragLocalID count;
  PFragLocalID id = ((PFragLocalID) random())%maxNumberOfPFrags;
  /*
   * Start at a random ID and return the first unused ID greater
   * than or equal to the start ID, unless a wrap around occurs,
   * in which case there are no more IDs to allocate.
   */
  for (count = 0; count < maxNumberOfPFrags; count++) {
    if (!isPFrag(id) && !isPFragInTransit(id)) {
      return id;
    }
    else {
      id++;
      if (id >= maxNumberOfPFrags) {
        id = 0;
      }
    }
  }
  return NoMorePFrags;
}

/*
 * Queues up a process fragment for transfer to a neighbor.  All
 * PFrags have a PFragState even if that PFrag never uses it.
 * The PFragState is sent first, followed immediately by the PFrag.
 */
unsigned char transferPFrag(PFragLocalID id,
    PushpinLocalID neighbor) {
  if (isPFragValid(id)) {
    return queuePacketForTransmission(neighbor, PFRAGSTATEMESSAGE,
      getPFragState(id) -> size,
      (unsigned char xdata *) getPFragState(id));
  }
  return 0;
}
```

D.10 BBS Management

```
/****************************************************************
 * BerthaBBS.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 18MAY2002 by lifton.
 *
 * These functions manage the Bulletin Board System.
 ****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>

/*
 * Should be called after a reset.  Clears the BBS.
 */
void initializeBBS(void) {
    XdataAddress xdata i;
    for (i=0; i<maxBBSSize; i++) {
        BBS[i] = 0;
    }
}

/*
 * This function adds posts system posts to the BBS.  A system post
 * is post added by Bertha.  All other posts should be added only by
 * PFrags.  All system posts have a localID of NoMorePFrags.  The
 * uid of system posts should be ignored.  Because this function
 * makes use of a portion of the data memory space (the memory space
 * PFrags use for local variables), it should not be called when a
 * PFrag is in the process of being called.  Specifically, it should
 * only be called in Bertha's main loop and not by any interrupt
 * service routines.
 */
void updateSystemBBSPosts(void) {
    unsigned char data BerthaBBSPost[BerthaBBSPostSize];
    /*
     * Use the NoMorePFrags ID to identify the the post as coming
     * from Bertha.
     */
    currentPFragID = NoMorePFrags;
    /*
     * Update timestamp post.
     */
    *((unsigned long data *) BerthaBBSPost) = getTime();
    removePost(TimeStamp);
    postToBBS(TimeStamp, sizeof(unsigned long), BerthaBBSPost);
    /*
     * Update LDR post.
     */
    removePost(LDRValue);
    *((unsigned int data *) BerthaBBSPost) = getADCValue(1);
    postToBBS(LDRValue, sizeof(unsigned int), BerthaBBSPost);
}
```

D.11 NW Management

```
/***************************************************************
 * BerthaNW.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 07MAR2002 by lifton.
 *
 * This file contains those functions most closely pertaining to
 * Bertha's maintenance of the Neighborhood Watch (NW). A NW
 * represents the sum of all the knowledge Bertha has about other
 * Pushpins within its communication radius. The NW is a LinkedList
 * wherein each element of the list is a Neighbor. Each Neighbor
 * represents one of the neighboring Pushpins. Another Pushpin is
 * considered neighboring if and only if it is within direct
 * communication range. The Neighbor data structure contains an
 * 8-bit local ID uniquely identifying it within the NW. Each
 * Pushpin uses this same 8-bit ID to identify itself to all its
 * neighbors. A network of Pushpins conforming to this
 * identification scheme is equivalent to a solution to the graph
 * theoretic coloring problem. The Neighbor data structure also
 * contains a synopsis of that neighbor's BBS in the form of a
 * LinkedList. The synopsis is simply a subset of all the posts of
 * the BBS it corresponds to. Which subset that is exactly depends
 * on two factors. First, it contains those posts the neighboring
 * Pushpin deems worthy of including. This is discussed in more
 * detail in the section pertaining to the BBS. Second, the subset
 * is constrained by the amount of space allotted to the Neighbor by
 * Bertha. The amount of space allotted to each Neighbor varies
 * according to how much space is requested and how much space is
 * available. Finally, the Neighbor data structure includes some
 * house keeping information, such as its size and whether or not the
 * Pushpin it corresponds to has been active within the last window
 * of time, as specified by Bertha. If it has not been active, the
 * neighbor is assumed to have been removed from the network and is
 * removed from the NW accordingly.
 ***************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>

void updateNW(void) {
  Neighbor xdata * xdata neighborPtr;
  /*
   * Wait for the NW to be released by the comm subsystem.
   */
  while (isNWBusy()); // meme : while (serialState != SCANNING);
  /*
   * Check the alarm, taking into consideration clock overflow.
   * neighborTimeoutAlarm == NEIGHBORTIMEOUT only when an
   * overflow is expected.
   */
  if (((neighborTimeoutAlarm == NEIGHBORTIMEOUT) &&
      (getTime() < neighborTimeoutAlarm)) ||
      ((neighborTimeoutAlarm != NEIGHBORTIMEOUT) &&
      (getTime() > neighborTimeoutAlarm))) {
    /*
     * The alarm has gone off. Check for dead neighbors and
     * remove them.
     */
    neighborPtr = (Neighbor xdata *)
      getNextListElement(0, (LinkedList) NW);
    while (*((ListElement xdata *) neighborPtr)) {
      if (isNeighborDead(neighborPtr)) {
        /*
         * Remove this dead neighbor. neighborPtr will now point
         * to the next neighbor automatically.
         */
        removeListElement((ListElement xdata *) neighborPtr,
          (LinkedList) NW);
      }
      else {
        /*
         * Mark this neighbor as dead and get the next neighbor.
         */
        setNeighborDead(neighborPtr);
        neighborPtr = (Neighbor xdata *)
          getNextListElement((ListElement xdata *) neighborPtr,
                             (LinkedList) NW);
      }
    }
    /*
     * Reset the alarm, avoiding wrap arounds.
     */
    if (neighborTimeoutAlarm + 2*NEIGHBORTIMEOUT >
        neighborTimeoutAlarm) {
      neighborTimeoutAlarm += NEIGHBORTIMEOUT;
    }
    else {
      /*
       * The next alarm will occur after the clock overflows.
       */
      neighborTimeoutAlarm = NEIGHBORTIMEOUT;
    }
  }
}

/*
 * Sends out over the communication channel a synopsis of the local
 * BBS. This function blocks until the entire synopsis is finished
 * sending so that the BBS isn't modified while its being sent.
 * Note that the null element indicating the end of the synopsis is
 * not sent with the synopsis and must be added by the receiver.
 * Currently, the BBS synopsis consists of the entire BBS.
 */
void broadcastSynopsis(void) {
  /*
   * Must be calculated separately so as not to invoke the library
   * functions. This is a serious bug that should be fixed.
   * Ideally, this calculation would take place in the argument
   * list of queuePacketForTransmission().
   */
```

```
    getListSize((LinkedList) BBS) - sizeof(ListElement);

  XdataAddress xdata size =



/*
   * First, wait for any other packets to finish transmitting and
   * then put the BBS synopsis in the queue.
   */
  while(queuePacketForTransmission(GLOBALADDRESS, NWMESSAGE,
              size, BBS));
  /*
   * Then, send the packet as soon as any incoming packets finish
   * arriving.
   */
  while(isPacketInQueue() && (!((serialState==PACKETSENT) ||
                                (serialState==SENDINGHEADER) ||
                                (serialState==SENDINGCONTENT) ||
                                (serialState==SENDINGTOADDRESS)))) {
    attemptToSendPacket();
  }
  /*
   * Finally, wait for the BBS synopsis packet to finish sending.
   */
  while(isPacketInQueue());
}


/*
 * This function coordinates the construction of the local
 * neighborhood network of Pushpins. It is a blocking function.
 * It should be called at reset, but used carefully otherwise. The
 * local neighborhood is constructed by listening to all network
 * traffic for a certain amount of time (a minimum amount of time
 * plus some random amount). A list of all to and from addresses
 * of all packets passing their CRC is kept in the Neighborhood
 * Watch, each address corresponding to a neighbor with a blank
 * synopsis. Note that this list of addresses may include those
 * belonging to Pushpins outside immediate communication radius.
 */
void buildNeighborhood(void) {
  unsigned long xdata i;

  disableCommunication();

  for (i=0; i<maxNWSize; i++) {
    NW[i] = 0;    // Erase existing Neighborhood watch.
  }

  i = getTime() + minNeighborhoodBuildTime + random();

  setBuildingNeighborhood();    //| Listen to all traffic.
  enableCommunication();        //|

  // Wait and listen for speaking neighbors.
  if (getTime() < i) {
    while(getTime() < i);
  }
  else {
    while(i < getTime());
  }

  // Listen only to traffic addressed to us.
  clearBuildingNeighborhood();
}

unsigned char isNeighborhoodIDValid(void) {
  unsigned char data BerthaBBSPost[BerthaBBSPostSize];
  unsigned long xdata i;
  if (NeighborhoodID != GLOBALADDRESS) {
    /*
     * Compose a message to all neighbors informing them of a
     * proposed neighborhood ID for this Pushpin.
     */
    BerthaMessage[0] = TEMPNEIGHBORID;
    BerthaMessage[1] = NeighborhoodID;
    BerthaMessage[2] = (unsigned char) random();
    BerthaMessage[3] = (unsigned char) random();
    BerthaMessage[4] = (unsigned char) random();
    BerthaMessage[5] = (unsigned char) random();
    BerthaMessage[6] = (unsigned char) random();
    BerthaMessage[7] = (unsigned char) random();
    BerthaMessage[8] = (unsigned char) random();
    BerthaMessage[9] = (unsigned char) random();
    /*
     * Queue the message for transmission only when it is sure to be
     * transmitted.
     */
    while(serialState != SCANNING);
    queuePacketForTransmission(GLOBALADDRESS, BERTHAMESSAGE, 10,
      &BerthaMessage);
    /*
     * Use the NoMorePFrags ID to identify the the post as coming
     * from Bertha.
     */
    currentPFragID = NoMorePFrags;
    /*
     * Copy the temporary 8-byte ID to an array in data memory so
     * it can be posted.
     */
    BerthaBBSPost[0] = BerthaMessage[2];
    BerthaBBSPost[1] = BerthaMessage[3];
    BerthaBBSPost[2] = BerthaMessage[4];
    BerthaBBSPost[3] = BerthaMessage[5];
    BerthaBBSPost[4] = BerthaMessage[6];
    BerthaBBSPost[5] = BerthaMessage[7];
    BerthaBBSPost[6] = BerthaMessage[8];
    BerthaBBSPost[7] = BerthaMessage[9];
    postToBBS(EightByteTempID, 8, BerthaBBSPost);
    /*
     * Give neighbors time to veto the new ID.
     */
    i = getTime() + minIDVetoWaitTime + random();
    if (getTime() < i) {
      while(getTime() < i);
    }
    else {
      while(i < getTime());
    }
    /*
     * If a neighbor vetoes the proposed ID, the post will be erased.
     * The ID is considered validated if no neighbors veto it.
     */
    if(getBBSPost(EightByteTempID, 8, BerthaBBSPost)) {
      removePost(EightByteTempID);
      return 1;
    }
  }
  /*
   * The global address is not a valid neighborhood ID for a single
   * Pushpin, or the proposed ID was vetoed.
   */
  return 0;
}

/*
 * Assigns an 8-bit ID unique relative to the local IDs of this
 * Pushpin's neighbors, as listed in the Neighborhood Watch, and
 * not equal to the global address.
 */
void generateNeighborhoodID(void) {
  NeighborhoodID = (unsigned char) random();
  disableCommunication();
  while (getNeighbor(NeighborhoodID) ||
         (NeighborhoodID == GLOBALADDRESS)) {
    NeighborhoodID = (unsigned char) random();
  }
  enableCommunication();
}

/*
 * Copies up to 'length' number of bytes from the 'n'th post in
 * 'neighborID's synopsis to the address 'post'.  Returns 0 if
 * the neighbor or the post does not exist, 1 otherwise.
 */
unsigned char getNthSynopsisPost(unsigned char n,
    Neighbor xdata * neighborPtr, unsigned int length,
    unsigned char data * post) {
  BBSPost xdata * xdata postPtr;
  while(isNWBusy());    // Guarantee the NW isn't being updated.
  if (neighborPtr && n) {
    postPtr = (BBSPost xdata *) getNextListElement(0,
      (LinkedList) &(neighborPtr -> synopsis));
    while (*((ListElement xdata *) postPtr)) {
      /*
       * Search for the nth post.
       */
      if (!--n) {
        /*
         * Compare size of origin to size of destination and
         * adjust accordingly.
         */
        if (length > (postPtr -> size) + sizeof(BBSPost) -
            sizeof(ReservedMemoryArea)) {
          length = (postPtr -> size) + sizeof(BBSPost) -
            sizeof(ReservedMemoryArea);
        }
        /*
         * Copy from origin to destination.
         */
        while(length--) {
          *(post++) = *(((unsigned char xdata *) postPtr)++);
```

```
        }
        return 1;
      }
      /*
       * Get the next list element.
       */
      postPtr = (BBSPost xdata *) getNextListElement(
        (ListElement xdata *) postPtr,
        (LinkedList) &(neighborPtr -> synopsis));
    }
  }
  return 0;
}

/*
 * Copies up to 'length' number of bytes from the post in
 * 'neighborID's synopsis with post ID equal to 'postID' to the
 * address 'post'.  Returns 0 if the neighbor or the post does not
 * exist, 1 otherwise.
 */
unsigned char getSynopsisPost(BBSPostID postID,
    Neighbor xdata * neighborPtr, unsigned int length,
    unsigned char data * post) {
  BBSPost xdata * xdata postPtr;
  while(isNWBusy());    // Guarantee the NW isn't being updated.
  if (neighborPtr) {
    postPtr = (BBSPost xdata *) getNextListElement(0, (LinkedList)
      &(neighborPtr -> synopsis));
    while (*((ListElement xdata *) postPtr)) {
      /*
       * Search for first instance of postID.
       */
      if ((postPtr -> postID) == postID) {
        /*
         * Compare size of origin to size of destination and adjust
         * accordingly.
         */
        if (length > (postPtr -> size) + sizeof(BBSPost) -
            sizeof(ReservedMemoryArea)) {
          length = (postPtr -> size) + sizeof(BBSPost) -
            sizeof(ReservedMemoryArea);
        }
        /*
         * Copy from origin to destination.
         */
        while(length--) {
          *(post++) = *(((unsigned char xdata *) postPtr)++);
        }
        return 1;
      }
      /*
       * Get the next list element.
       */
      postPtr = (BBSPost xdata *) getNextListElement(
        (ListElement xdata *) postPtr,
        (LinkedList) &(neighborPtr -> synopsis));
    }
  }
  return 0;
}
```

```
/*
 * Try adding a new neighbor with a local ID as given in the
 * argument.  Don't do anything if there isn't enough memory or if
 * the Neighbor exists already.  Also, don't add a neighbor with a
 * local ID equal to the global address.
 */
void addNeighbor(PushpinLocalID id) {
  Neighbor xdata * neighborPtr;
  if (!getNeighbor(id) && id != GLOBALADDRESS) {
    neighborPtr = (Neighbor xdata *) addListElement(
      (ListElement) sizeof(Neighbor), (LinkedList) &NW, maxNWSize);
    if (neighborPtr){
      (neighborPtr -> id) = id;
      (neighborPtr -> synopsis) = 0;
    }
  }
}

/*
 * Try removing the neighbor with a local ID as given in the argument.
 * Don't do anything if the Neighbor doesn't exist.
 */
void removeNeighbor(PushpinLocalID id) {
  removeListElement((ListElement xdata *) getNeighbor(id),
    (LinkedList) &NW);
}

/*
 * Returns a pointer to the neighbor identified with the local ID
 * passed as an argument. Returns 0 if there is no neighbor
 * corresponding local ID in question.
 */
Neighbor xdata * getNeighbor(PushpinLocalID ID) reentrant {
  Neighbor xdata * xdata neighborPtr =
    (Neighbor xdata *) getNextListElement(0, (LinkedList) &NW);
  while (*((ListElement xdata *) neighborPtr)) {
    if ((neighborPtr -> id) == ID) {
      break;
    }
    else {
      neighborPtr = (Neighbor xdata *) getNextListElement(
        (ListElement xdata *) neighborPtr, (LinkedList) &NW);
    }
  }
  if (*((ListElement xdata *) neighborPtr) == 0) {
    return 0;   // No neighbor with given ID.
  }
  return neighborPtr;
}

/*
 * Returns a pointer to the synopsis of the 'n'th neighbor. Returns
 * 0 if there is no neighbor corresponding local ID in question.
 */
Neighbor xdata * getNthNeighbor(PushpinLocalID n) {
  Neighbor xdata * xdata neighborPtr = 0;
  if (n) {
    neighborPtr = (Neighbor xdata *) getNextListElement(0,
      (LinkedList) &NW);
    while (*((ListElement xdata *) neighborPtr)) {
      if (!--n) {
        break;
      }
      else {
        neighborPtr = (Neighbor xdata *) getNextListElement(
          (ListElement xdata *) neighborPtr, (LinkedList) &NW);
      }
    }
  }
  if (*((ListElement xdata *) neighborPtr) == 0) {
    return 0;   // No neighbor with given ID.
  }
  return neighborPtr;
}
```

D.12 Communication Subsystem

```
/*****************************************************************
 * BerthaCommunication.h
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 21APR2002 by lifton.
 *
 * This file contains all functions needed to handle all levels of
 * communication.
 *
 *****************************************************************/

/*****************************************************************

All communication is packet based and interrupt driven.

A packet is composed of a header and content.  A header is composed
of the address of the packet's node of origin, the address of the
intended destination, the packet's classification, the number of
bytes contained in the packet's content, and the packet's checksum.
Content can consist of up to 64Kbytes of contiguous memory.  That
is, a packet's content must reside somewhere (anywhere) in the
Pushpin as a single block of memory no more than 64Kbytes bytes in
length.  In practice, only content up to 2Kbytes should be accepted
by other Pushpins, as this is the largest block of memory of any
interest -- a PFrag.

Care must be taken when sending or receiving packets.  In
particular, it is important to note that all routines originating
from a serial interrupt must terminate before another serial
interrupt can be handled.  Thus, it is not possible to send a
packet as a direct and immediate response to receiving a packet.
To get around this, a packet can be placed in a transmit queue
that is dealt with after all other serial interrupts are handled.

Once enabled, the communication subsystem's default state should
be listening for packets.

*****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>

/*
 * The following are the reload values for 8-bit timer baud rate
 * generation used by the UART.  Values are calculated assuming
 * a clock frequency of 22.1184MHz. Note that BAUDRATE must be
 * greater than a certain minimum value in order for the value of
 * TIMER2IRSERIALRELOAD to be valid.  The resistor and capacitor
 * values listed in the comments after each reload value indicate
 * typical values used on the 4-way IR communication module to
 * generate that baud rate.  Note that in general bit rate and
 * baud rate are not the same; baud rate is the number of signal
 * transitions per second whereas bit rate is the number of bits
 * of information transmitted per second.
 */
#define BAUD9600 0x70
#define BAUD14400 0xA0
#define BAUD19200 0xB9
#define BAUD28800 0xD0
#define BAUD38400 0xDC
#define BAUD57600 0xE8       // 10kOhm, 1800pF
#define BAUD92160 0xF1       // 10kOhm, 1085pF
#define BAUD115200 0xF4      // 10kOhm, 820pF
#define MAXBAUDRATE 0xFE     // 10kOhm, 270pF or 220pF
#define BAUDRATE TH1         // Timer 1 reload generates baud rate.

/*
 * The following values determine the overall baudrate and the
 * timing needed for the initial data stream used by the recipient
 * to lock onto the IR channel.
 */
#define OVERHEAD 250
#define OVERFLOWSPERBIT 16
#define BITSPERUARTBYTE 10
#define UARTBYTESPERDATABYTE 2
#define CYCLESPERBYTE \
   (UARTBYTESPERDATABYTE*BITSPERUARTBYTE*OVERFLOWSPERBIT* \
   (0xFF-BAUDRATE) + OVERHEAD)
#define NUMBEROFSCANBYTES 3
#define TIMER2TRANSMITRELOAD (0xFFFF - CYCLESPERBYTE)
#define TIMER2RECEIVERELOAD (0xFFFF - \
   NUMBEROFSCANBYTES*CYCLESPERBYTE)
#define RECEIVETIMEOUTRELOAD (0XFFFF - (NUMBEROFSCANBYTES + 1) \
   *CYCLESPERBYTE)
#define BROADCASTADDRESSREPITITIONS 6

/*
 * These functions are defined as macros for the sake of execution
 * speed and clarity.
 */
#define renewReceiveTimeout() timer2 = RECEIVETIMEOUTRELOAD

/*
 * Global state needed throughout the communciation process.
 */
unsigned int xdata currentByte = 0;
unsigned char xdata bytesRemaining = 0;
unsigned char xdata byteBuffer = 0;
unsigned char xdata tempCRC = 0;

/*
 * Initiates the process of transmitting a packet.  Only one packet
 * at a time (the transmitPacket) can be held in the queue.
 */
unsigned char queuePacketForTransmission(
                     PushpinLocalID toAddress,
                     unsigned char packetType,
                     XdataAddress contentSize,
                     unsigned char xdata * contentPtr) reentrant {
   if (!isPacketInQueue()) {
        transmitPacket.fromAddress = NeighborhoodID;

        transmitPacket.toAddress = toAddress;



transmitPacket
.
packetType
 =
packetType
;
        transmitPacket.contentSize = contentSize;
        transmitPacket.contentPtr = contentPtr;
        transmitPacket.CRC = packetCRC8(&transmitPacket);
        setPacketInQueue();
        if (attemptToSendPacket()) {
            return PACKETTRANSMITTING;
        }
        return PACKETQUEUED;
    }
    return PACKETNOTQUEUED;
}

/*
 * Starts transmitting a packet if the channel is clear.  Otherwise,
 * waits a random amount of time and checks again.  If the channel
 * is still busy, returns without sending the packet.  This process
 * is only initiated if there is indeed a packet queued to be sent.
 * The communication channel is defined to be busy if data is being
 * received.
 */
unsigned char attemptToSendPacket(void) {
    /*
     * Check for a packet to send that is not already being sent.
     */
    if (!((serialState==PACKETSENT)||(serialState==SENDINGHEADER)||
      (serialState==SENDINGCONTENT)||(serialState==SENDINGTOADDRESS)
      )) {
        /*
         * Check if channel is open.
         */
        if (serialState != SCANNING) {
            delay(random());
            if (serialState != SCANNING) {
                return 0;
            }
        }
        disableSerialReceive();
        clearTransmitInterruptFlag();
        clearReceiveInterruptFlag();
        enableSerialInterrupts();
        switch (commMode) {
            case (RS232) :
                serialState = SENDINGHEADER;
                currentByte = 1;
                SBUF = transmitPacket.toAddress;
                return 1;
                break;
            case (FOURWAYIR) :
                serialState = SENDINGTOADDRESS;
```

```
                currentByte = 1;
                disableTimer2();
                byteBuffer = transmitPacket.toAddress;
                SBUF = encodeNybbleTable[0x0F & byteBuffer];
                byteBuffer >>= 4;
                bytesRemaining = 1;
                return 1;
                break;
            default :
                disableCommunication();
                enableCommunication();
                break;
        }
    }
    return 0;
}

/*
 * This function is called once a complete packet has been received.
 * This function manages the validation of the received packet and
 * passes the content on to appropriate handlers.
 */
void packetReceived(void) {
    PFragLocalID xdata pfragIndex;
    ListElement xdata * xdata elementPtr;
    disableTimer2();        // Don't want to timeout.
    if (isBuildingNeighborhood()) {
        if (receivePacket.CRC == tempCRC) {
            addNeighbor(receivePacket.fromAddress);
            addNeighbor(receivePacket.toAddress);
        }
    }
    else if (receivePacket.CRC == packetCRC8(&receivePacket)) {
        switch (receivePacket.packetType) {
            case BERTHAMESSAGE :
                processBerthaMessage(receivePacket.fromAddress);
                break;
            case NWMESSAGE :
                /*
                 * Change the null synopsis at the end of the list
                 * to a real synopsis.  Size of the synopsis
                 * includes the null element at the end to be
                 * added shortly.
                 */
                elementPtr = getLastListElement((LinkedList) NW);
                *elementPtr = receivePacket.contentSize +
                   sizeof(Neighbor);
                /*
                 * Assign the new synopsis to the neighbor from
                 * whence it came.
                 */
                (((Neighbor xdata *) elementPtr) -> id) =
                   receivePacket.fromAddress;
                /*
                 * The neighbor just received is not DOA.
                 */
                clearNeighborDead(elementPtr);
                /*
                 * Cap the end of the NW with a null element.
                 */
                elementPtr = (ListElement xdata *)
```

```
               (((unsigned char xdata *) elementPtr) +
               *elementPtr);
            *elementPtr = 0;
            /*
             * Cap the end of the synopsis just added with a
             * null element.
             */
            elementPtr++;
            *elementPtr = 0;
            /*
             * Check if the synopsis just added should replace
             * an older version of itself.
             */
            elementPtr--;
            elementPtr = (ListElement xdata *)
               (((unsigned char xdata *) elementPtr) -
               receivePacket.contentSize - sizeof(Neighbor));
            if (getNeighbor(receivePacket.fromAddress) !=
               (Neighbor xdata *) elementPtr) {
                 removeNeighbor(receivePacket.fromAddress);
            }
            /*
             * Indicate to the PFrags that the NW is now free.
             */
            clearNWBusy();
            break;
        case PFRAGMESSAGE :
            /*
             * Add a blank state entry to for the new PFrag.
             */
            for (pfragIndex = 0;
                 pfragIndex < maxNumberOfPFrags;
                 pfragIndex++) {
                /*
                 * It's not possible for more than one PFrag to
                 * be in transit at any given time.
                 */
                if (isPFragInTransit(pfragIndex)) {
                    clearPFragInTransit(pfragIndex);
                    /*
                     * Having a state is a condition of being a
                     * PFrag.  Either the state was received
                     * before the PFrag itself was received, or
                     * the PFrag must be allocated fresh state
                     * memory.
                     */
                    if (pfragIndex == StateInWaitingID) {
                        StateInWaitingID = NoMorePFrags;
                        validatePFrag(pfragIndex);
                        if (isPFragValid(pfragIndex) &&
                          ((getPFragState(pfragIndex) -> size) -
                            sizeof(PFragState) +
                            sizeof(ReservedMemoryArea) ==
                            PFragPointers[pfragIndex]->stateSize))
                        {
                            setPFrag(pfragIndex);
                        }
                        else {
                            setPFragRemoveReady(pfragIndex);
                        }
                    }
                    else {
                        if (allocatePFragState(pfragIndex, 1)) {
                            setPFrag(pfragIndex);
                        }
                        else {
                            setPFragRemoveReady(pfragIndex);
                        }
                    }
                    break;
                }
            }
            break;
        case PFRAGSTATEMESSAGE :
            /*
             * The local ID shipped with the PFragState is only
             * applicable to the Pushpin from which it came.  A
             * new ID valid for this Pushpin must be assigned.
             */
         (((PFragState xdata *) receivePacket.contentPtr) -> id)
              = StateInWaitingID;
         break;
        case RANDOMSEEDMESSAGE :
         break;
        default :
         break;
    }
}
else {
    switch (receivePacket.packetType) {
        case BERTHAMESSAGE :
            /*
             * Don't do anything... ignore the message.
             */
            break;
        case NWMESSAGE :
            /*
             * Free the NW.  No other changes are necessary.
             */
            clearNWBusy();
            break;
        case PFRAGMESSAGE :
            /*
             * Throw away the incomplete PFrag.
             */
            for (pfragIndex = 0;
                 pfragIndex < maxNumberOfPFrags;
                 pfragIndex++) {
                if (isPFragInTransit(pfragIndex)) {
                    setPFragRemoveReady(pfragIndex);
                    break;
                }
            }
            break;
        case PFRAGSTATEMESSAGE :
            /*
             * Undo changes to the PFragStateTable.
             */
            break;
        case RANDOMSEEDMESSAGE :
            /*
             * Don't need to do anything.
```

```
                 */
                break;
            default :
                break;
        }
    }
    /*
     * This should be the very last action performed.
     */
    serialState = SCANNING;
}

/*
 * Sets the medium over which communication takes place.
 */
void setCommMode(void) {
    setUARTMode1();          // 8-bit serial communication.
    T2CON = 0x01;            // Configure timer 2 (no auto-reload).
    CKCON |= 0x20;           // Set timer 2 to run off system clock.
    TMOD |= 0x20;            // Set timer1 as 8-bit auto reload.
    PCON |= 0x80;            // Double baud rate; ie, SMOD = 1.

    /*
     * Determine the mode of communication.
     */
    enableADC();
    if (getADCValue(COMMADCCHANNEL) > IRADCTHRESHOLD) {
        commMode = FOURWAYIR;
    }
    else {
        commMode = RS232;
    }

    switch (commMode) {
        case RS232 :
            setUARTMode1();          // 8-bit serial communication.
            BAUDRATE = BAUD19200;    // Set timer1 reload.
            break;
        case FOURWAYIR :
            BAUDRATE = BAUD92160;
            PRT1CF &= 0x0F;          //|
            IRChannel0 = 1;          //|
            IRChannel1 = 1;          //| Set pins P1.[4-7] as inputs.
            IRChannel2 = 1;          //|
            IRChannel3 = 1;          //|
            break;
        default :
            break;
    }
}

/*
 * Initiates an interrupt-driven scan and reception of a packet.
 */
void enableCommunication(void) {
    currentByte = 0;
    bytesRemaining = 0;
    enableTimer1();
    clearTimer2InterruptFlag();
    enableTimer2Interrupt();
    enableSerialReceive();
```

```
    clearTransmitInterruptFlag();
    clearReceiveInterruptFlag();
    clearPacketInQueue();
    clearNWBusy();
    enableSerialInterrupts();
    serialState = SCANNING;
}

/*
 * Shuts down communication.
 */
void disableCommunication(void) {
    serialState = DISABLED;
    disableTimer1();
    disableTimer2();
    disableTimer2Interrupt();
    disableSerialInterrupts();
    disableSerialReceive();
    clearTimer2InterruptFlag();
    clearTransmitInterruptFlag();
    clearReceiveInterruptFlag();
    clearPacketInQueue();
    clearNWBusy();
}

/*
 * Handles received bytes by building up a packet one byte at a
 * time.  This function is independent of the physical layer of
 * communication used.
 */
void setNextPacketByte(unsigned char c) {
    switch (serialState) {
        case SCANNING :
            if (c == NeighborhoodID || c == GLOBALADDRESS
                || isBuildingNeighborhood()) {
                listenToAllBytes();
                receivePacket.toAddress = 0;
                receivePacket.fromAddress = 0;
                receivePacket.packetType = 0;
                receivePacket.contentSize = 0;
                receivePacket.CRC = 0;
                currentByte = 0;
                /*
                 * Fall through to next case.
                 */
                serialState = RECEIVINGHEADER;
            }
            else {
                /*
                 * Ignore all all bytes not addressed to this Pushpin.
                 * This includes those bytes from incomplete packets.
                 */
                break;
            }
        case RECEIVINGHEADER :
            /*
             * Wait until end of toAddress broadcast.
             */
            if ((c==receivePacket.toAddress) && (currentByte==1)) {
                break;
            }
            *(((unsigned char xdata *) &receivePacket)
              + currentByte++) = c;
            /*
             * Finished receiving the header.  Decide what to do
             * with content.
             */
            if (currentByte >= PACKETHEADERSIZE) {
                if (isBuildingNeighborhood()) {
                    currentByte = 0;
                    tempCRC = 0;
                    tempCRC = CRC8Table[tempCRC ^
                      receivePacket.toAddress];
                    tempCRC = CRC8Table[tempCRC ^
                      receivePacket.fromAddress];
                    tempCRC = CRC8Table[tempCRC ^
                      receivePacket.packetType];
                    tempCRC = CRC8Table[tempCRC ^
                      *((unsigned char xdata *)
                        &(receivePacket.contentSize))];
                    tempCRC = CRC8Table[tempCRC ^
                      *(((unsigned char xdata *)
                        &(receivePacket.contentSize)) + 1)];
                    if (receivePacket.contentSize) {
                        serialState = IGNORINGCONTENT;
                    }
                    else {
                        serialState = PACKETRECEIVED;
                    }
                }
                else if (receivePacket.contentSize != 0) {
                    currentByte = 0;
                    receivePacket.contentPtr = getContentPtr();
                    if (receivePacket.contentPtr) {
                        serialState = RECEIVINGCONTENT;
                    }
                    else {
                        serialState = IGNORINGCONTENT;
                    }
                }
                else {
                    serialState = PACKETRECEIVED;
                }
            }
            break;
        case RECEIVINGCONTENT :
            /*
             * contentPtr is an instance of (unsigned char xdata *)
             * but when flash write is enabled, the MOVX
             * instruction is interpretted as a write to the flash
             * memory.
             */
            switch (receivePacket.packetType) {
                case PFRAGMESSAGE :
                    /*
                     * Fall through to next case.
                     */
                case RANDOMSEEDMESSAGE :
                    receivePacket.contentPtr += currentByte;
                    /*
                     * Must disable interrupts to avoid extra MOVX
                     * executions.
                     */
                    disableGlobalInterrupts();
                    /*
                     * The next MOVX instruction will effect flash.
                     */
                    enableFlashWrite();
                    *(receivePacket.contentPtr) = c;
                    disableFlashWrite();
                    enableGlobalInterrupts();
                    receivePacket.contentPtr -= currentByte;
                    currentByte++;
                    break;
                default :
                    *(receivePacket.contentPtr + currentByte++) = c;
                    break;
            }
            if (currentByte >= receivePacket.contentSize) {
                serialState = PACKETRECEIVED;
            }
            break;
        case IGNORINGCONTENT :
            currentByte++;
            if (isBuildingNeighborhood()) {
                tempCRC = CRC8Table[tempCRC ^ c];
            }
            if (currentByte >= receivePacket.contentSize) {
                serialState = PACKETRECEIVED;
            }
            break;
        default :
            break;
    }
}

/*
 * Returns the next byte to send out over the communication link.
 */
unsigned char getNextPacketByte() reentrant {
  unsigned char xdata next;
  switch (serialState) {
    case SENDINGHEADER :
      next = *(((unsigned char *) &transmitPacket)
        + (currentByte++));
      if (currentByte >= PACKETHEADERSIZE) {
        currentByte = 0;
        if (transmitPacket.contentSize > 0) {
          serialState = SENDINGCONTENT;
        }
        else {
          serialState = PACKETSENT;
        }
      }
      break;
    case SENDINGCONTENT :
      switch (transmitPacket.packetType) {
        case PFRAGMESSAGE :
          /*
           * Fall through to next case.
           */
        case RANDOMSEEDMESSAGE :
          next = *(((unsigned char code *)
            (transmitPacket.contentPtr)) + (currentByte++));
          break;
        default :
          next = *(transmitPacket.contentPtr + (currentByte++));
          break;
      }
      if (currentByte >= transmitPacket.contentSize) {
        serialState = PACKETSENT;
      }
      break;
    default :
      break;
  }
  return next;
}

/*
 * Determines where to put the data contained in the incoming
 * packet based on information contained in the header.  Returns a
 * null pointer if the request cannot be accommodated for any
 * reason (space limitations, invalid header, etc.).
 */
unsigned char xdata *  getContentPtr(void) {
  PFragLocalID xdata incomingPFrag;
  switch (serialState) {
    case RECEIVINGHEADER :
      if ((StateInWaitingID != NoMorePFrags) &&
          (receivePacket.packetType != PFRAGMESSAGE)) {
        /*
         * The PFragState that was previously received was not
         * followed immediately by a PFrag and should therefore
         * be deleted.
         */
        setPFragRemoveReady(StateInWaitingID);
        StateInWaitingID = NoMorePFrags;
      }
      switch (receivePacket.packetType) {
        case NWMESSAGE :
          /*
           * Warn PFrags that the NW will soon become unusable.
           * In the interest of time efficiency, but at the risk of
           * violating data synchrony, this could be placed in
           * packetReceived().
           */
          setNWBusy();
          /*
           * Limit the size of the incoming synopsis if there is
           * not enough room for it.  Recall that the null element
           * indicating the end of the synopsis must be added and
           * is not included in the calculation of
           * receivePacket.contentSize.
           */
          if (maxNWSize > getListSize((LinkedList) NW) +
              sizeof(Neighbor) + sizeof(ListElement)) {
            if (receivePacket.contentSize >
                (maxNWSize - getListSize((LinkedList) NW) -
                sizeof(Neighbor) - sizeof(ListElement))) {
              receivePacket.contentSize =
                maxNWSize - getListSize((LinkedList) NW) -
                sizeof(Neighbor) - sizeof(ListElement);
            }
            /*
             * Pretend the last null element is no longer null and
             * return a pointer to its synopsis as the place to put
             * the incoming synopsis.  Only after the synopsis is
             * copied should the null element be changed to a
             * non-null value.
             */
            return ((unsigned char xdata *)
              getLastListElement((LinkedList) NW)) +
              (sizeof(Neighbor) - sizeof(ListElement));
          }
          break;
        case PFRAGMESSAGE :
          if (StateInWaitingID != NoMorePFrags) {
            /*
             * The incoming PFrag's state has already arrived.
             */
            incomingPFrag = StateInWaitingID;
          }
          else {
            /*
             * The incoming PFrag did not come with an associated
             * state.
             */
            incomingPFrag = allocatePFragLocalID();
          }
          if (incomingPFrag != NoMorePFrags &&
              receivePacket.contentSize <= maxPFragSize) {
            setPFragInTransit(incomingPFrag);
            return ((unsigned char xdata *)
              PFragPointers[incomingPFrag]);
          }
          break;
        case PFRAGSTATEMESSAGE :
          StateInWaitingID = allocatePFragLocalID();
          if (StateInWaitingID != NoMorePFrags) {
            setPFragInTransit(StateInWaitingID);
            return (unsigned char xdata * )
              allocatePFragState(StateInWaitingID, 0);
          }
          break;
        case RANDOMSEEDMESSAGE :
          if (receivePacket.contentSize <= 2*RandomSeedSize) {
            return ((unsigned char xdata *) RandomSeedAddress);
          }
          break;
        default :
          if (receivePacket.contentSize <= BerthaMessageSize) {
            return &BerthaMessage;
          }
          break;
      }
      break;
    default :
      break;
  }
  return 0;
}


/*
```

```
 * Note on the UART:
 * Interrupt number 4 is the serial interrupt.  A serial interrupt
 * is called any time RI or TI is set to 1.  This can be done in
 * either software or hardware.  RI is set to 1 by the UART hardware
 * whenever the UART has completed receiving a byte.  The byte just
 * received resides in SBUF.  TI is set to 1 by the UART hardware
 * whenever the UART has completed transmitting a byte.  Loading a
 * byte into SBUF initiates the transmission of that byte by the
 * UART.  Thus, the statement SBUF=SBUF instructs the UART to
 * transmit the byte it last received.  Both TI and RI are only
 * cleared to 0 in software; the hardware does not clear them at all.
 *
 * This interrupt service routine checks the state of the
 * transmission or reception processes and acts accordingly; it
 * either transmits the next byte or it logs the byte just received
 * and waits for the next byte.  The byteHalfSent variable keeps tabs
 * on which half of the data byte (high nybble or low nybble) is
 * being dealt with.  All other global state regarding serial
 * communication is held in the serialState variable.
 */
void serialInterrupt (void) interrupt 4 {
  /*
   * The receive interrupt (RI) indicates that the UART has completed
   * receiving a byte.
   */
  if (RI) {
    clearReceiveInterruptFlag();
    renewReceiveTimeout();
    enableTimer2();
    switch (commMode) {
      case RS232 :          // Each network byte is a data byte.
        setNextPacketByte(SBUF);
        break;
      case FOURWAYIR :      // Two network bytes for each data byte.
        if (!bytesRemaining) {
          byteBuffer = decodeNybbleTable[SBUF];
          bytesRemaining = 1;
        }
        else {
          byteBuffer |= (decodeNybbleTable[SBUF] << 4);
          bytesRemaining = 0;
          setNextPacketByte(byteBuffer);
        }
        break;
      default :             // This should never happen.
        break;
    }
    if (serialState == PACKETRECEIVED) {
      listenToAddressBytes();
      packetReceived();
    }
  }
  if (TI) {
    clearTransmitInterruptFlag();
    switch (commMode) {
      case RS232 :
        if (serialState != PACKETSENT) {
          SBUF = getNextPacketByte();
        }
        else if (serialState == PACKETSENT) {
          clearPacketInQueue();
          enableSerialReceive();
          serialState = SCANNING;
          if (transmitPacket.packetType == PFRAGSTATEMESSAGE) {
            /*
             * A PFragState was just sent.  The accompanying
             * PFrag must be sent immediately.
             */
            transmitPacket.packetType = PFRAGMESSAGE;
            transmitPacket.contentPtr = (unsigned char xdata *)
              PFragPointers[((PFragState xdata *)
              (transmitPacket.contentPtr)) -> id];
            transmitPacket.contentSize = ((PFrag code *)
              (transmitPacket.contentPtr)) -> size;
            transmitPacket.CRC = packetCRC8(&transmitPacket);
            setPacketInQueue();
            attemptToSendPacket();
          }
        }
        break;
      case FOURWAYIR :
        if (bytesRemaining) {
          /*
           * Send second half of the byte being sent.
           */
          SBUF = encodeNybbleTable[0x0F & byteBuffer];
          bytesRemaining = 0;
        }
        else if (serialState == SENDINGTOADDRESS) {
          /*
           * Start countdown to sending next byte.
           */
          timer2 = TIMER2TRANSMITRELOAD;
          clearTimer2InterruptFlag();
          enableTimer2();
        }
        else if (serialState != PACKETSENT) {
          /*
           * Send the next byte in the packet.
           */
          byteBuffer = getNextPacketByte();
          SBUF = encodeNybbleTable[0x0F & byteBuffer];
          byteBuffer >>= 4;
          bytesRemaining = 1;
        }
        else if (serialState == PACKETSENT) {
          clearPacketInQueue();
          enableSerialReceive();
          serialState = SCANNING;
          if (transmitPacket.packetType == PFRAGSTATEMESSAGE) {
            /*
             * A PFragState was just sent.  The accompanying
             * PFrag must be sent immediately.
             */
            transmitPacket.packetType = PFRAGMESSAGE;
            transmitPacket.contentPtr = (unsigned char xdata *)
              PFragPointers[((PFragState xdata *)
              (transmitPacket.contentPtr)) -> id];
            transmitPacket.contentSize = ((PFrag code *)
              (transmitPacket.contentPtr)) -> size;
            transmitPacket.CRC = packetCRC8(&transmitPacket);
            setPacketInQueue();
            attemptToSendPacket();
          }
        }
        break;
      default :   // Should never get here.
        break;
    }
  }
}

void timer2Interrupt(void) interrupt 5 {
  switch (commMode) {
    case FOURWAYIR :
      /*
       * Nothing interesting on current channel, so switch to the
       * next channel.
       */
      if (serialState == SCANNING) {
        IRChannelSelect0 = ~IRChannelSelect0;
        if (!IRChannelSelect0) {
          IRChannelSelect1 = ~IRChannelSelect1;
        }
        timer2 = TIMER2RECEIVERELOAD;
      }
      /*
       * Send the packet recipient's address several times separated
       * by a byte interval.
       */
      else if (serialState == SENDINGTOADDRESS) {
        if (currentByte++ < BROADCASTADDRESSREPITITIONS) {
          byteBuffer = transmitPacket.toAddress;
          SBUF = encodeNybbleTable[0x0F & byteBuffer];
          byteBuffer >>= 4;
          bytesRemaining = 1;
        }
        else {
          serialState = SENDINGHEADER;
          currentByte = 1;
          byteBuffer = getNextPacketByte();
          SBUF = encodeNybbleTable[0x0F & byteBuffer];
          byteBuffer >>= 4;
          bytesRemaining = 1;
        }
        disableTimer2();
      }
      /*
       * The rest of the packet didn't arrive and the receive timed
       * out.  Stop receiving.
       */
      else if (REN) {
        disableCommunication();     //| Reset communication state.
        enableCommunication();      //|
      }
      break;
    default :
      if (REN) {
        disableCommunication();     //| Reset communication state.
        enableCommunication();      //|
      }
      break;
  }
```

```
  clearTimer2InterruptFlag();
}

/*
 * Adapted from: http://cell-relay.indiana.edu/mhonarc/cell-relay/
 *                   1999-Jan/msg00074.html
 * 8 bit CRC Generator, MSB shifted first
 * Polynom: x^8 + x^2 + x^1 + 1
 *
 * Calculates an 8-bit cyclic redundancy check sum for a packet.
 * This function takes care not to include the packet's check sum
 * in calculating the check sum.  Assumes the CRC is the last byte
 * of the packet header.  Also takes care to look in code space
 * (instead of xdata space) when dealing with a PFrag.
 */
unsigned char packetCRC8(Packet xdata * packet) reentrant {
  unsigned int xdata i;
  unsigned char xdata crc = 0;
  /*
   * Don't include the CRC itself when calculating the CRC.
   */
  for (i=0; i < (PACKETHEADERSIZE - 1); i++) {
    crc=CRC8Table[crc ^ (*(((unsigned char xdata *) packet) + i))];
  }
  switch ((*packet).packetType) {
    case PFRAGMESSAGE :     // Fall through to next case.
    case RANDOMSEEDMESSAGE :
      for (i=0; i < (*packet).contentSize; i++) {
        crc = CRC8Table[crc ^ (*(((unsigned char code *)
          ((*packet).contentPtr)) + i))];
      }
      break;
    default :
      for (i=0; i < (*packet).contentSize; i++) {
        crc = CRC8Table[crc ^ (*(((*packet).contentPtr) + i))];
      }
      break;
  }
  return crc;
}
```

D.13 Expansion Module Support

```
/****************************************************************
 * ExpansionModules.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 07MAY2002 by lifton.
 *
 * Functions pertaining to Bertha's control of various expansion
 * modules.
 ****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>

void enableLDRModule(void) {
  /*
   * Reconfigure crossbar.
   */
  XBR2 &= ~(0x40);     // Disable crossbar.
  XBR0 |= 0x28;       //| Output all PCA modules to port pins.
  XBR0 &= ~(0x10);    //|
  XBR2 |= 0x40;       // Enable crossbar.
  /*
   * Configure ADC.
   */
  enableADC();
  /*
   * Enable LDR and wait for it to settle.
   */
  LDR = 1;
  delay(0xffff);
  /*
   * Configure PCA.
   */
  PCA0MD = 0;         // Select system clock /12 for counter.
  PCA0CPM0 = 0x02;     // Set all PCA modules to PWM.
  PCA0CPM1 = 0x02;
  PCA0CPM2 = 0x02;
  PCA0CPM3 = 0x02;
  PCA0CPM4 = 0x02;
  PCA0CPH0 = 0xFF;     // 0% duty cycle on all PWMs.
  PCA0CPH1 = 0xFF;
  PCA0CPH2 = 0xFF;
  PCA0CPH3 = 0xFF;
  PCA0CPH4 = 0xFF;
  CR = 1;             // Enable PCA counter timer.
}
```

D.14 Timing Functions

```
/*****************************************************************
 * BerthaTiming.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 08APR2002 by lifton.
 *
 * Implementation of Bertha's timing functions, including the
 * real-time clock.  Assumes the Pushpin is running off a
 * 22.1184MHz crystal.  A 32-bit real-time clock is formed by
 * conjoining timer 0 running in 16-bit clock mode with a 16-bit
 * unsigned integer incremented every time timer 0 overflows.  Timer
 * 0 is configured increment once for every 12 system clocks.  Given
 * this arrangement, the 32-bit real-time clock should overflow once
 * every 38.84 minutes.
 *
 * Software alarms can be set such that an interrupt is generated
 * after a specified amount of time, up to the limit of the real-time
 * clock.  Timer 3 triggers the actual alarm.  Overhead between
 * calling the setAlarm function and actually setting the alarm
 * accounts for approximately 32 microseconds.  This overhead time is
 * taken into account when setting the alarm.
 *
 *****************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaFunctionPrototypes.h>

#define ALARMOVERHEAD 30

unsigned int xdata highTime;

void startRealTimeClock(void) {
    /*
     * Timer 0 is the lower 16 bits of the 32 bit real-time clock.
     */
    disableTimer0();
    enableTimer0Interrupt();
    clearTimer0InterruptFlag();
  timer0FromSystemClockDiv12();
    TMOD &= ~(0x0F);     // Clear timer 0 control registers.
    TMOD |= 0x01;        // Set timer 0 as 16-bit timer.
  /*
   * Timer 3 acts keeps track of the alarm.
   */
  disableTimer3();
  enableTimer3Interrupt();
  clearTimer3InterruptFlag();
  timer3FromSystemClockDiv12();

  highTime = 0;
  clearHighAlarmArmed();
    setTime(0);
    enableTimer0();
}

/*
 * The argument to this function specifies how many microseconds
 * (1.085 microseconds actually) to delay before setting off the
 * timer 3 interrupt. The minimum delay is the overhead this
 * function requires for setting the alarm.  The overhead is
 * empirically calculated.
 */
void setAlarm(unsigned long fromNow) {
  AddressableLong xdata time;
  disableTimer0();
  time.Int[0] = highTime;
    time.Char[2] = TH0;
    time.Char[3] = TL0;
  enableTimer0();
  if (fromNow > ALARMOVERHEAD) {
    fromNow -= ALARMOVERHEAD;
  }
  else {
    fromNow = 0;
  }
  if (fromNow > 0xFFFF) {
      fromNow += time.Long;
    timer3reload = *((unsigned int *) &fromNow);
    timer3 = 0xFFFF - ((unsigned int) fromNow);
    setHighAlarmArmed();
  }
  else {
    timer3 = 0xFFFF - ((unsigned int) fromNow);
    enableTimer3();
  }
}

unsigned long getTime(void) {
    AddressableLong xdata time;
  disableTimer0();
  time.Int[0] = highTime;
    time.Char[2] = TH0;
    time.Char[3] = TL0;
  enableTimer0();
    return time.Long;
}

void setTime(unsigned long time) {
    TL0 = *(((unsigned char *) &time) + 3);
    TH0 = *(((unsigned char *) &time) + 2);
  highTime = *((unsigned int *) &time);
}

void timer0Interrupt(void) interrupt 1 {
    clearTimer0InterruptFlag();
  highTime++;
  if (isHighAlarmArmed() && (highTime == timer3reload)) {
    clearHighAlarmArmed();
        enableTimer3();
  }

}
```

```
void timer3Interrupt(void) interrupt 14 {




disableTimer3
();
  clearTimer3InterruptFlag();
  // meme : add alarm state and switch statement here.
    setAlarm(ONESECOND);
  statusLED = ~statusLED;
}
```

D.15 Miscellaneous Functions

```
/******************************************************************
 * BerthaFunctions.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 07MAR2002 by lifton.
 *
 * These functions are for Bertha's internal use only.
 ******************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <BerthaGlobals.h>
#include <stdlib.h>

void configurePins(void) {
  PRT0CF = 0xFF; // Set all of port 0 to push-pull outputs.
  PRT1CF = 0xFF; // Set all of port 1 to push-pull outputs.
  PRT2CF = 0xFF; // Set all of port 2 (not pinned out) to push-pull.
  XBR0 |= 0x04;  // UART TX and RX on P0.0 and P0.1.
  XBR2 |= 0x40;  // Enable crossbar.
  XBR2 |= 0x80;  // Disable weak pull-ups.
}

void disableWatchdogTimer(void) {
  WDTCN = 0xde;        // This register must be set first.
  WDTCN = 0xad;        // This register must be set second.
}

void enableExternalClock(void) {
  CKCON |= 0x10;  // Don't divide by 12 for base clock; ie, T1M=1.
  OSCXCN = 0x67;  // Start external oscillator > 6.74MHz.
  delay(0xffff);  // Wait more than a millisecond.
  while (!(OSCXCN & 0x80)) {} // Wait for oscillator to stablize.
  OSCICN |= 0x80;        // Check for missing clock.
  OSCICN |= 0x08;        // Use external crystal.
  FLSCL = ((FLSCL & 0xF0) | 0x09);  // Set flash scalar.
}

void initializeHardware(void) {
  disableWatchdogTimer();
  enableExternalClock();
  configurePins();
  statusLED = 0;
}

/*
 * A wrapper for the srand() function provided in stdlib.h.
 */
void setRandomSeed(int seed) {
  srand(seed);
}

/*
 * Handles an OS message delivered from a neighboring Pushpin.
 */
void processBerthaMessage(PushpinLocalID from) {
  /*
   * This array is only to be used when no PFrags are in the middle
   * of execution.
   */
  BBSPost data BerthaBBSPost;
  switch (BerthaMessage[0]) {
    case ERASEALLPFRAGS :
      /*
       * If there are any resident PFrags, then erase them
       * and pass on the message to neighbors.
       */
      for (from = 0; from < maxNumberOfPFrags; from++) {
        if (isPFrag(from)) {
          setPFragRemoveReady(from);
          /*
           * This packet is actually only sent once since
           * this routine is called by the serial interrupt routine.
           */
          queuePacketForTransmission(GLOBALADDRESS, BERTHAMESSAGE,
            receivePacket.contentSize, &BerthaMessage);
        }
      }
      break;
    case ERASEPFRAG :
      /*
       * If there are any resident PFrags of the specified UID, then
       * erase them and pass on the message to neighbors.
       */
      for (from = 0; from < maxNumberOfPFrags; from++) {
        if (isPFrag(from)) {
          if ((PFragPointers[from] -> uid) ==
              *((PFragUID xdata *) &(BerthaMessage[2]))) {
            queuePacketForTransmission(GLOBALADDRESS, BERTHAMESSAGE,
              receivePacket.contentSize, &BerthaMessage);
            setPFragRemoveReady(from);
          }
        }
      }
      break;
    case ERASERANDOMSEED :
      /*
       * Erase the 128-byte random seed located in flash.  Must
       * disable interrupts to avoid extra MOVX executions.
       * Unlike removePFrag (the only other instance of erasing
       * flash memory), erasing the random seed is carried out
       * as part of the serial interrupt routine and therefore
       * doesn't need to worry about disrupting communication.
       */
      disableGlobalInterrupts();
      enableFlashErase();
      *((unsigned char xdata *) RandomSeedAddress) = 0x00;
      disableFlashErase();
      enableGlobalInterrupts();
      break;
    case TEMPNEIGHBORID :
     * Veto this ID if it exists already.  Note that if another

    /*



*
packet

is

already

queued

for

transmission
,
the

veto

won
'
t
     * be sent.  This should happen only very rarely.  Note that
     * the veto must be sent to the global address in case the
     * id collision is with this Pushpin, in which case both the
     * to address and from address would be the same, a condition
     * not permitted in the four-way IR communication protocol.
     */
    if (getNeighbor(receivePacket.fromAddress) ||
        (NeighborhoodID == receivePacket.fromAddress)) {
      BerthaMessage[0] = IDVETO;
      // meme flashLED(1,0xFFFF);
      queuePacketForTransmission(GLOBALADDRESS, BERTHAMESSAGE,
        10, BerthaMessage);
    }
    break;
  case IDVETO :
    /*
     * A neighbor has vetoed a proposed ID.  Save the
     * currentPFragID value, check if it is this Pushpin's ID that
     * is vetoed, and then restore the the currentPFragID value.
     * See isNeighborhoodIDValid() for details.  The value of
     * currentPFragID must be changed to allow Bertha to
     * manipulate the BBS.
     */
    BerthaMessage[1] = currentPFragID;
    currentPFragID = NoMorePFrags;
    if(getBBSPost(EightByteTempID, sizeof(BBSPost)+8-1,
        (unsigned char data *) &BerthaBBSPost)) {
      if (BerthaMessage[2] == BerthaBBSPost.message[0] &&
          BerthaMessage[3] == BerthaBBSPost.message[1] &&
          BerthaMessage[4] == BerthaBBSPost.message[2] &&
          BerthaMessage[5] == BerthaBBSPost.message[3] &&
          BerthaMessage[6] == BerthaBBSPost.message[4] &&
          BerthaMessage[7] == BerthaBBSPost.message[5] &&
          BerthaMessage[8] == BerthaBBSPost.message[6] &&
          BerthaMessage[9] == BerthaBBSPost.message[7]) {
```

```
          removePost(EightByteTempID);
        }
        /*
         * Reset random seed just in case.
         */
        setRandomSeed(*((unsigned int code *) (RandomSeedAddress +
          (((unsigned char) random())%RandomSeedSize))));
      }
      currentPFragID = BerthaMessage[1];
      break;
    case SETTIMESCALE :
      /*
       * Sets the number of times the delay function is called
       * in Bertha's main loop.  Allows for changing the time
       * scale at which the system operates.  Sends the message
       * on to all neighbors if the time scale is set to a
       * new value.
       */
      if (timeScale!=*((unsigned int xdata *) &(BerthaMessage[1]))){
        timeScale = *((unsigned int xdata *) &(BerthaMessage[1]));
        queuePacketForTransmission(GLOBALADDRESS, BERTHAMESSAGE,
                                   receivePacket.contentSize,
                                   BerthaMessage);
      }
      break;
    default :
      break;
  }
}

/*
 * Configure and enable analog to digital converter.  Select
 * channel 0 as default.
 */
void enableADC(void) {
  REF0CN |= 0x03;    // Enable VREF and ADC/DAC bias.
  AMX0CF = 0x00;     // Single-ended inputs (not differential).
  AMX0SL = 0x00;     // Select Analog Input 0 (AIN0).
  ADC0CF &= ~(0x07); // Gain 1.
  ADC0CF |= 0xE0;    // Conversion clock to system clock / 16.
  ADLJST = 0;        // Right justification.
  ADCTM = 1;         // Tracking is software controlled
                     // necessary for software control through ADBUSY
  ADCEN = 1;         // Enable ADC.
  getADCValue(0);
}
```

D.16 Linked List Functions

```
/****************************************************************
 * LinkedList.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 11APR2002 by lifton.
 *
 * A common data structure used by Bertha.  Assumes the list is
 * stored in external memory (xdata).  Each ListElement is composed
 * simply of one piece of information -- the number of bytes
 * required by that element.  The LinkedList structure is
 * essentially a pointer to the first ListElement.  The next
 * ListElement is arrived at by adding the value of the ListElement
 * to the address of the ListElement.  The memory space in between
 * is used to hold the data itself.  The advantage of each
 * ListElement being the size as opposed to a pointer to the next
 * element is that ListElements are easily transferred from one list
 * to another.  A ListElement with value (size) '0' indicates the
 * end of the LinkedList, whereas a ListElement pointer pointing to
 * '0' indicates a request for the first element in the list.  As a
 * result, no list should begin at memory element '0'.
 ****************************************************************/

#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaGlobals.h>
#include <BerthaFunctionPrototypes.h>

/*
 * Removes the ListElement passed as a pointer from the associated
 * LinkedList.  Note that this function does not check whether the
 * ListElement is an element of the list.
 */
unsigned char removeListElement(ListElement xdata * elementPtr,
    LinkedList list) reentrant {
  ListElement xdata * xdata nextElementPtr;
  if (elementPtr && ((XdataAddress) elementPtr >=
      (XdataAddress) list)) { //Don't remove the end-of-list element.
    nextElementPtr = getNextListElement(elementPtr, list);
    xdataMemMove((unsigned char *) elementPtr,
      (unsigned char *) nextElementPtr,
      (XdataAddress) getLastListElement(list) -
        (XdataAddress) nextElementPtr + sizeof(ListElement));
    return 1; // Succeeded in removing list element.
  }
  return 0;   // Failed to remove list element.
}

/*
 * Takes a ListElement and LinkedList as arguments and attempts to
 * add the element to the list, according to the space limitations
 * of the Xdata memory.  Returns a pointer to the new element if
 * successful, returns '0' otherwise.
 */
ListElement xdata * addListElement(ListElement newElement,
    LinkedList list, XdataAddress maxListSize) reentrant {
  ListElement xdata * xdata elementPtr;
  elementPtr = getLastListElement(list);
  if ((XdataAddress) elementPtr + (XdataAddress) newElement +
      sizeof(ListElement) < maxListSize + (XdataAddress) list) {
    *elementPtr = newElement;
    elementPtr = getNextListElement(elementPtr, list);
    *elementPtr = 0;    // Indicate end of list.
    // Success.  Return pointer to new element.
    return (XdataAddress) elementPtr - (XdataAddress) newElement;
  }
  return 0;          // Failed to add the ListElement.
}

/*
 * Returns the number of bytes used by a linked list, including the
 * null list element marking the end of the list of size
 * sizeof(ListElement).
 */
XdataAddress getListSize(LinkedList list) reentrant {
  return ((XdataAddress) getLastListElement(list) -
    (XdataAddress) list + sizeof(ListElement));
}

/* Returns the number of elements containted in a linked list.
 */
unsigned char getNumberOfElements(LinkedList list) {
  unsigned char xdata n = 0;
  ListElement xdata * xdata elementPtr = (ListElement xdata *) list;
  while (*elementPtr) {
    n++;
    elementPtr = getNextListElement(elementPtr, list);
  }
  return n;
}

/*
 * Given a LinkedList, returns a pointer to the last element of that
 * list.  Note the last element is a null element.
 */
ListElement xdata * getLastListElement(LinkedList list) reentrant {
  ListElement xdata * xdata elementPtr = (ListElement xdata *) list;
  while (*elementPtr) {
    elementPtr = getNextListElement(elementPtr, list);
  }
  return elementPtr;
}

/*
 * Takes a pointer to a list element and the list that element
 * belongs to and returns the next element in the list.  If the
 * pointer to the element is null, the head of the list is returned.
 * If the element itself is null (indicating it is the last element
 * in the list), the same element is returned.
 */
ListElement xdata * getNextListElement(
    ListElement xdata * elementPtr, LinkedList list) {
  if (elementPtr == 0) {    // Request for beginning of list.
    elementPtr = (ListElement *) list;
  }
  }

  else if (*elementPtr == 0) {      // Already at end of list.



else
 {
    ((unsigned char xdata *) elementPtr) += *elementPtr;
  }
  return elementPtr;
}

/*
 * This function moves a contiguous block of memory from one place
 * to another within the additional RAM (addressed as external
 * memory) of the Cygnal processor.  The function guarantees that
 * no portion of the original block of memory is overwritten
 * until it has been copied.
 */
void xdataMemMove(unsigned char xdata * to,
    unsigned char xdata * from, unsigned int size) reentrant {
  unsigned int xdata i;
  if ((to + size < XdataSize) && (from + size < XdataSize)) {
    if (to > from) {
      for (i=size; i>0; i--){
        *(to + i) = *(from + i);
      }
    }
    if (to < from) {
      for (i=0; i<size; i++){
        *(to + i) = *(from + i);
      }
    }
  }
}
```

D.17 System Call Table

```
/*****************************************************************
 * SystemCallTable.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2001 all rights reserved.
 *
 * Last modified 08MAY2002 by lifton.
 *
 * This table serves as the primary interface between process
 * fragments and the operating system.  The table is included as
 * part of the operating system, but is used primarily by process
 * fragments in order to call system functions provided by the
 * operating system.  The table holds up to 256 (to allow one-byte
 * addressing) pointers to system functions.  Each pointer is a
 * memory-specific pointer (as opposed to a generic pointer), which
 * takes up two bytes of memory.  The resulting table is therefore
 * 512 bytes in size.
 *
 * In order to use this table, each process fragment must know both
 * the location of the table within memory and the organization of
 * the function pointers within the table.  Both of these pieces of
 * information are included with the process fragments when they are
 * compiled in the form of an absolute memory address and an
 * enumeration of available system functions.
 *
 * Two issues concerning this table arise when linking the operating
 * system.  First, the table must be placed in a particular place in
 * memory.  This is accomplished by adding the
 * CODE(?CO?SystemCallTable(3584)) option to the command line call to
 * the BL51 linker locator.  This option simply instructs the linker
 * to place the ?CO?SystemCallTable segment at address 3584 in code
 * memory.  Thus, the system call table takes up the 512 bytes
 * leading up to address 4096.  The second issue arises from the fact
 * that the linker automatically discards segments which are never
 * called in the call tree.  Since the operating system never calls
 * the system call table, the linker will discard the system call
 * table segment unless specifically instructed not to.  That is,
 * the function call tree built up by the linker must be modified to
 * include a call from the main function to the system call table
 * segment.  Using the OVERLAY(main ! ?CO?SystemCallTable) command
 * line option when calling the BL51 linker locator does just that.
 *****************************************************************/

#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>

void code * code SystemCallTable[maxNumberOfSystemFunctions] = {
  delay,
  die,
  flashLDRLED,
  flashLED,
  getADCValue,
  getBBSPost,
  getBBSPostCount,
  getNeighborCount,
  getNthBBSPost,
  getNthNeighborID,
  getNthPostFromMthSynopsis,
  getNthPostFromSynopsis,
  getNthSynopsisPostCount,
  getPFragUID,
  getPostFromMthSynopsis,
  getPostFromSynopsis,
  getSynopsisPostCount,
  isStateReplicated,
  postToBBS,
  random,
  removeAllPosts,
  removePost,
  setLEDIntensity,
  transfer
};
```

D.18 System Functions

```
/******************************************************************
 * SystemFunctions.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 26MAR2002 by lifton.
 *
 * These functions are available to both Bertha and PFrags.
 ******************************************************************/

#include <c8051F000.h>
#include <PushpinHardwareV3.h>
#include <Bertha.h>
#include <BerthaPFragShared.h>
#include <BerthaFunctionPrototypes.h>
#include <stdlib.h>
#include <BerthaGlobals.h>

/*************
 * NW
 *************/

/*
 * Returns the total number of neighbors.
 */
unsigned char getNeighborCount(void) {
  return getNumberOfElements((LinkedList) &NW);
}

/*
 * Returns the total number of posts in the synopsis belonging to
 * the neighbor corresponding to 'neighborID'.  Returns 0 if the
 * neighbor does not exist.
 */
unsigned char getSynopsisPostCount(PushpinLocalID neighborID) {
  Neighbor xdata * neighborPtr = getNeighbor(neighborID);
  if ((ListElement xdata *) neighborPtr) {
    return getNumberOfElements((LinkedList) &(neighborPtr->synopsis));
  }
  return 0;
}

/*
 * Returns the total number of posts in the 'n'th synopsis.
 * Returns 0 if the neighbor does not exist.
 */
unsigned char getNthSynopsisPostCount(PushpinLocalID n) {
  Neighbor xdata * neighborPtr = getNthNeighbor(n);
  if ((ListElement xdata *) neighborPtr) {
    return getNumberOfElements((LinkedList) &(neighborPtr->synopsis));
  }
  return 0;
}
```

```
/*
 * The following four functions correspond to the four permutations
 * of retrieving a post (identified either by order in the list or
 * post ID) from a synopsis (identified either by order in the list
 * or neighbor ID).
 */
unsigned char getNthPostFromSynopsis(unsigned char n,
                                     PushpinLocalID neighborID,
                                     unsigned int length,
                                     unsigned char data * post) {
  return getNthSynopsisPost(n, getNeighbor(neighborID), length, post);
}

unsigned char getPostFromSynopsis(BBSPostID postID,
                                  PushpinLocalID neighborID,
                                  unsigned int length,
                                  unsigned char data * post) {
  return getSynopsisPost(postID,getNeighbor(neighborID),length,post);
}

unsigned char getNthPostFromMthSynopsis(unsigned char n,
                                        PushpinLocalID m,
                                        unsigned int length,
                                        unsigned char data * post) {
  return getNthSynopsisPost(n, getNthNeighbor(m), length, post);
}

unsigned char getPostFromMthSynopsis(BBSPostID postID,
                                     PushpinLocalID m,
                                     unsigned int length,
                                     unsigned char data * post) {
  return getSynopsisPost(postID, getNthNeighbor(m), length, post);
}


/*************
 * BBS
 *************/

/*
 * Copies up to 'length' number of bytes from the 'n'th BBS post to
 * the address 'post'.  Returns 0 if the post does not exist, 1
 * otherwise.
 */
unsigned char getNthBBSPost(unsigned char n,
                            unsigned int length,
                            unsigned char data * post) {
  BBSPost xdata * xdata postPtr =
    (BBSPost xdata *) getNextListElement(0, (LinkedList) &BBS);
  if (n == 0) {
    return 0;
  }
  while (*((ListElement xdata *) postPtr)) {
    /*
     * Search for the nth post.
     */
    if (!n--) {
      /*
       * Compare size of origin to size of destination and adjust
       * accordingly.
       */
```

```
                      sizeof(ReservedMemoryArea)) {

      if (length > (postPtr->size) + sizeof(BBSPost) -



length
 = (
postPtr
 ->
size
) +
sizeof
(
BBSPost
) -
                    sizeof(ReservedMemoryArea);
      }
      /*
       * Copy from origin to destination.
       */
      while(length--){
        *(post++) = *(((unsigned char xdata *) postPtr)++);
      }
      return 1;
    }
    /*
     * Get the next list element.
     */
    postPtr = (BBSPost xdata *) getNextListElement(
                                    (ListElement xdata *) postPtr,
                                    (LinkedList) &BBS);
  }
  return 0;
}

/*
 * Copies up to 'length' number of bytes from the BBS post whose
 * post ID matches 'postID' to the address 'post'.  Returns 0 if
 * the post does not exist, 1 otherwise.
 */
unsigned char getBBSPost(BBSPostID postID,
                         unsigned int length,
                         unsigned char data * post) {
  BBSPost xdata * xdata postPtr =
    (BBSPost xdata *) getNextListElement(0, (LinkedList) &BBS);
  while (*((ListElement xdata *) postPtr)) {
    /*
     * Search for first instance of postID.
     */
    if ((postPtr -> postID) == postID) {
      /*
       * Compare size of origin to size of destination and adjust
       * accordingly.
       */
      if (length > (postPtr -> size) + sizeof(BBSPost) -
                      sizeof(ReservedMemoryArea)) {
        length = (postPtr -> size) + sizeof(BBSPost) -
                    sizeof(ReservedMemoryArea);
      }
      /*
       * Copy from origin to destination.
       */
      while(length--){
        *(post++) = *(((unsigned char xdata *) postPtr)++);
      }
      return 1;
    }
    /*
     * Get the next list element.
     */
    postPtr = (BBSPost xdata *) getNextListElement(
                                    (ListElement xdata *) postPtr,
                                    (LinkedList) &BBS);
  }
  return 0;
}

/*
 * This function deletes a single post from the BBS.  The first
 * post in the BBS with postID matching the argument and originally
 * posted by the active PFrag is removed.  Returns 1 if the post
 * was successfully removed, 0 otherwise.
 */
unsigned char removePost(BBSPostID postID) {
  BBSPost xdata * xdata postPtr =
    (BBSPost xdata *) getNextListElement(0, (LinkedList) &BBS);
  while (*((ListElement xdata *) postPtr)) {
    if ((postPtr -> localID) == currentPFragID) {
      if ((postPtr -> postID) == postID) {
        return removeListElement((ListElement xdata *) postPtr,
                                 (LinkedList) &BBS);
      }
    }
    postPtr = (BBSPost xdata *) getNextListElement(
                                    (ListElement xdata *) postPtr,
                                    (LinkedList) &BBS);
  }
  return 0;
}

unsigned char removeAllPosts(void) {
  unsigned char xdata numOfPostsRemoved = 0;
  BBSPost xdata * xdata postPtr =
    (BBSPost xdata *) getNextListElement(0, (LinkedList) &BBS);
  while (*((ListElement xdata *) postPtr)) {
    if ((postPtr -> localID) == currentPFragID) {
      if (removeListElement((ListElement xdata *) postPtr,
                            (LinkedList) &BBS)) {
        numOfPostsRemoved++;
      }
    }
    postPtr = (BBSPost xdata *) getNextListElement(
                                    (ListElement xdata *) postPtr,
                                    (LinkedList) &BBS);
  }
  return numOfPostsRemoved;
}


/*
 * This function writes a single post to the BBS.  The post is
 * rejected if it is larger than the BBS can accommodate.  The
 * argument ‘postID’ is the post ID assigned to the post.  The
 * argument ‘length’ is the number of bytes of content to be posted.
 * The argument ‘messagePtr’ is a pointer to the beginning of the
 * message to be posted, which is assumed to be of length ‘length’.
 * Returns 1 if the post was successfully written to the BBS, 0
 * otherwise.
 */
unsigned char postToBBS(BBSPostID postID,
                        unsigned int length,
            unsigned char data * messagePtr) {
  unsigned char xdata * xdata postContentPtr;
  BBSPost xdata * xdata postPtr = addListElement(
           (ListElement) (length + sizeof(BBSPost) - 1),
                 (LinkedList) &BBS,
                     maxBBSSize);
  if (postPtr) {
    /*
     * Fill in all fields of the BBS post.
     */
    (postPtr -> postID) = postID;
    (postPtr -> uid) = (PFragPointers[currentPFragID] -> uid);
    (postPtr -> localID) = currentPFragID;
    for (postContentPtr = &((unsigned char) (postPtr -> message));
      postContentPtr < &((unsigned char)(postPtr->message)) + length;
      postContentPtr++) {
       *postContentPtr = *(messagePtr++);
    }
    return 1;
  }
  else {
    /*
     * The post request was not fulfilled.
     */
    return 0;
  }
}

unsigned char getBBSPostCount(void) {
  return getNumberOfElements((LinkedList) &BBS);
}

/******************
 * General Purpose
 ******************/

/*
 * A blocking function that flashes the LED located on the processing
 * module a certain number of times at a certain interval, both given
 * as arguments. To be replaced by use of a timer.
 */
void flashLED(unsigned int times, unsigned int interval) reentrant {
  while(times--) {
    statusLED = 1;
    delay(interval);
    statusLED = 0;
    delay(interval);
  }
}

/*
```

```
 * Sets the duty cycle of the pulse width modulator corresponding
 * to the LED 'color', based on the value of 'intensity'.  The
 * minimum duty cycle corresponds to an 'intensity' value of 0xFF,
 * ramping up linearly to a maximum duty cycle corresponding to an
 * 'intensity' value of 0x00.
 */
void setLEDIntensity(unsigned char color, unsigned char intensity) {
  switch (color) {
    case RED1 :
      PCA0CPH3 = intensity;
      break;
    case RED2 :
      PCA0CPH2 = intensity;
      break;
    case AMBER1 :
      PCA0CPH1 = intensity;
      break;
    case YELLOW1 :
      PCA0CPH0 = intensity;
      break;
    case GREEN1 :
      PCA0CPH4 = intensity;
      break;
    default :
      break;
  }
}

/*
 * A blocking function that flashes the LED of type 'color' located
 * on the LDR expansion module a certain number of times, at a
 * certain interval, at a certain intensity, all given as arguments.
 */
void flashLDRLED(unsigned char color,
                 unsigned int times,
                 unsigned int interval,
                 unsigned char intensity) {
  switch (color) {
    case RED1 :
      while (times--) {
        PCA0CPH3 = intensity;
        delay(interval);
        PCA0CPH3 = 0xFF;
        delay(interval);
      }
      break;
    case RED2 :
      while (times--) {
        PCA0CPH2 = intensity;
        delay(interval);
        PCA0CPH2 = 0xFF;
        delay(interval);
      }
      break;
    case AMBER1 :
      while (times--) {
        PCA0CPH1 = intensity;
        delay(interval);
        PCA0CPH1 = 0xFF;
        delay(interval);
      }
      break;
    case YELLOW1 :
      while (times--){
        PCA0CPH0 = intensity;
        delay(interval);
        PCA0CPH0 = 0xFF;
        delay(interval);
      }
      break;
    case GREEN1 :
      while (times--){
        PCA0CPH4 = intensity;
        delay(interval);
        PCA0CPH4 = 0xFF;
        delay(interval);
      }
      break;
    default :
      break;
  }
}

/*
 * A blocking function that simply waits an amount of time roughly
 * proportional to the value of the argument.
 */
void delay(unsigned int interval) {
  while(interval--);
}

/*
 * Removes the current PFrag.
 */
void die(void) {
  setPFragRemoveReady(currentPFragID);
}

/*
 * A wrapper for the rand() function provided by stdlib.h.  Returns
 * a pseudo-random number between 0 and 32767 (2^15 - 1).  PFrags
 * shouldn't call rand() directly because of seed initialization
 * issues.
 */
unsigned int random(void) {
  return (unsigned int) rand();
}

/*
 * Initiates a transfer to another Pushpin.  Returns 1 if transfer
 * was successfully initiated, 0 otherwise.  Note that a successful
 * transfer initiation does not guarantee a successful transfer.
 */
unsigned char transfer(PushpinLocalID pushpin) {
  return transferPFrag(currentPFragID, pushpin);
}
```

D.19 Lookup Tables

```
/*****************************************************************
 * codecs.c
 * target: PushpinV3
 *
 * Josh Lifton
 * lifton@media.mit.edu
 * MIT Media Lab
 * Copyright 2002 all rights reserved.
 *
 * Last modified 18APR2002 by lifton.
 *
 * Look up tables.
 *****************************************************************/

/*
 * Taken from: http://cell-relay.indiana.edu/mhonarc/cell-relay/
 * 1999-Jan/msg00074.html
 * 8 bit CRC Generator, MSB shifted first
 * Polynom: x^8 + x^2 + x^1 + 1
 */
const unsigned char code CRC8Table[256] = {
    0x00,0x07,0x0E,0x09,0x1C,0x1B,0x12,0x15,
    0x38,0x3F,0x36,0x31,0x24,0x23,0x2A,0x2D,
    0x70,0x77,0x7E,0x79,0x6C,0x6B,0x62,0x65,
    0x48,0x4F,0x46,0x41,0x54,0x53,0x5A,0x5D,
    0xE0,0xE7,0xEE,0xE9,0xFC,0xFB,0xF2,0xF5,
    0xD8,0xDF,0xD6,0xD1,0xC4,0xC3,0xCA,0xCD,
    0x90,0x97,0x9E,0x99,0x8C,0x8B,0x82,0x85,
    0xA8,0xAF,0xA6,0xA1,0xB4,0xB3,0xBA,0xBD,
    0xC7,0xC0,0xC9,0xCE,0xDB,0xDC,0xD5,0xD2,
    0xFF,0xF8,0xF1,0xF6,0xE3,0xE4,0xED,0xEA,
    0xB7,0xB0,0xB9,0xBE,0xAB,0xAC,0xA5,0xA2,
    0x8F,0x88,0x81,0x86,0x93,0x94,0x9D,0x9A,
    0x27,0x20,0x29,0x2E,0x3B,0x3C,0x35,0x32,
    0x1F,0x18,0x11,0x16,0x03,0x04,0x0D,0x0A,
    0x57,0x50,0x59,0x5E,0x4B,0x4C,0x45,0x42,
    0x6F,0x68,0x61,0x66,0x73,0x74,0x7D,0x7A,
    0x89,0x8E,0x87,0x80,0x95,0x92,0x9B,0x9C,
    0xB1,0xB6,0xBF,0xB8,0xAD,0xAA,0xA3,0xA4,
    0xF9,0xFE,0xF7,0xF0,0xE5,0xE2,0xEB,0xEC,
    0xC1,0xC6,0xCF,0xC8,0xDD,0xDA,0xD3,0xD4,
    0x69,0x6E,0x67,0x60,0x75,0x72,0x7B,0x7C,
    0x51,0x56,0x5F,0x58,0x4D,0x4A,0x43,0x44,
    0x19,0x1E,0x17,0x10,0x05,0x02,0x0B,0x0C,
    0x21,0x26,0x2F,0x28,0x3D,0x3A,0x33,0x34,
    0x4E,0x49,0x40,0x47,0x52,0x55,0x5C,0x5B,
    0x76,0x71,0x78,0x7F,0x6A,0x6D,0x64,0x63,
    0x3E,0x39,0x30,0x37,0x22,0x25,0x2C,0x2B,
    0x06,0x01,0x08,0x0F,0x1A,0x1D,0x14,0x13,
    0xAE,0xA9,0xA0,0xA7,0xB2,0xB5,0xBC,0xBB,
    0x96,0x91,0x98,0x9F,0x8A,0x8D,0x84,0x83,
    0xDE,0xD9,0xD0,0xD7,0xC2,0xC5,0xCC,0xCB,
    0xE6,0xE1,0xE8,0xEF,0xFA,0xFD,0xF4,0xF3
};

const unsigned char code encodeNybbleTable[16] = {
  0x55, // 0b00000000 -> 0b01010101
  0x57, // 0b00000001 -> 0b01010111
  0x5D, // 0b00000010 -> 0b01011101
  0x5F, // 0b00000011 -> 0b01011111
  0x75, // 0b00000100 -> 0b01110101
  0x77, // 0b00000101 -> 0b01110111
  0x7D, // 0b00000110 -> 0b01111101
  0x7F, // 0b00000111 -> 0b01111111
  0xD5, // 0b00001000 -> 0b11010101
  0xD7, // 0b00001001 -> 0b11010111
  0xDD, // 0b00001010 -> 0b11011101
  0xDF, // 0b00001011 -> 0b11011111
  0xF5, // 0b00001100 -> 0b11110101
  0xF7, // 0b00001101 -> 0b11110111
  0xFD, // 0b00001110 -> 0b11111101
  0xFF  // 0b00001111 -> 0b11111111
};

// The look up table for decoding a byte into a nibble.
const unsigned char code decodeNybbleTable[256] = {
  0xFF, // 0b00000000 -> error
  0xFF, // 0b00000001 -> error
  0xFF, // 0b00000010 -> error
  0xFF, // 0b00000011 -> error
  0xFF, // 0b00000100 -> error
  0xFF, // 0b00000101 -> error
  0xFF, // 0b00000110 -> error
  0xFF, // 0b00000111 -> error
  0xFF, // 0b00001000 -> error
  0xFF, // 0b00001001 -> error
  0xFF, // 0b00001010 -> error
  0xFF, // 0b00001011 -> error
  0xFF, // 0b00001100 -> error
  0xFF, // 0b00001101 -> error
  0xFF, // 0b00001110 -> error
  0xFF, // 0b00001111 -> error
  0xFF, // 0b00010000 -> error
  0xFF, // 0b00010001 -> error
  0xFF, // 0b00010010 -> error
  0xFF, // 0b00010011 -> error
  0xFF, // 0b00010100 -> error
  0xFF, // 0b00010101 -> error
  0xFF, // 0b00010110 -> error
  0xFF, // 0b00010111 -> error
  0xFF, // 0b00011000 -> error
  0xFF, // 0b00011001 -> error
  0xFF, // 0b00011010 -> error
  0xFF, // 0b00011011 -> error
  0xFF, // 0b00011100 -> error
  0xFF, // 0b00011101 -> error
  0xFF, // 0b00011110 -> error
  0xFF, // 0b00011111 -> error
  0xFF, // 0b00100000 -> error
  0xFF, // 0b00100001 -> error
  0xFF, // 0b00100010 -> error
  0xFF, // 0b00100011 -> error
  0xFF, // 0b00100100 -> error
  0xFF, // 0b00100101 -> error
  0xFF, // 0b00100110 -> error
  0xFF, // 0b00100111 -> error
  0xFF, // 0b00101000 -> error
  0xFF, // 0b00101001 -> error
  0xFF, // 0b00101010 -> error
  0xFF, // 0b00101100 -> error

  0xFF, // 0b00101011 -> error



0
xFF
,
//
 0
b00101101
 ->
error
  0xFF, // 0b00101110 -> error
  0xFF, // 0b00101111 -> error
  0xFF, // 0b00110000 -> error
  0xFF, // 0b00110001 -> error
  0xFF, // 0b00110010 -> error
  0xFF, // 0b00110011 -> error
  0xFF, // 0b00110100 -> error
  0xFF, // 0b00110101 -> error
  0xFF, // 0b00110110 -> error
  0xFF, // 0b00110111 -> error
  0xFF, // 0b00111000 -> error
  0xFF, // 0b00111001 -> error
  0xFF, // 0b00111010 -> error
  0xFF, // 0b00111011 -> error
  0xFF, // 0b00111100 -> error
  0xFF, // 0b00111101 -> error
  0xFF, // 0b00111110 -> error
  0xFF, // 0b00111111 -> error
  0xFF, // 0b01000000 -> error
  0xFF, // 0b01000001 -> error
  0xFF, // 0b01000010 -> error
  0xFF, // 0b01000011 -> error
  0xFF, // 0b01000100 -> error
  0xFF, // 0b01000101 -> error
  0xFF, // 0b01000110 -> error
  0xFF, // 0b01000111 -> error
  0xFF, // 0b01001000 -> error
  0xFF, // 0b01001001 -> error
  0xFF, // 0b01001010 -> error
  0xFF, // 0b01001011 -> error
  0xFF, // 0b01001100 -> error
  0xFF, // 0b01001101 -> error
  0xFF, // 0b01001110 -> error
  0xFF, // 0b01001111 -> error
  0xFF, // 0b01010000 -> error
  0xFF, // 0b01010001 -> error
  0xFF, // 0b01010010 -> error
  0xFF, // 0b01010011 -> error
  0xFF, // 0b01010100 -> error
  0x00, // 0b01010101 -> 0b00000000
  0xFF, // 0b01010110 -> error
  0x01, // 0b01010111 -> 0b00000001
  0xFF, // 0b01011000 -> error
  0xFF, // 0b01011001 -> error
  0xFF, // 0b01011010 -> error
  0xFF, // 0b01011011 -> error
  0xFF, // 0b01011100 -> error
```

```
0x02, // 0b01011101 -> 0b00000010
0xFF, // 0b01011110 -> error
0x03, // 0b01011111 -> 0b00000011
0xFF, // 0b01100000 -> error
0xFF, // 0b01100001 -> error
0xFF, // 0b01100010 -> error
0xFF, // 0b01100011 -> error
0xFF, // 0b01100100 -> error
0xFF, // 0b01100101 -> error
0xFF, // 0b01100110 -> error
0xFF, // 0b01100111 -> error
0xFF, // 0b01101000 -> error
0xFF, // 0b01101001 -> error
0xFF, // 0b01101010 -> error
0xFF, // 0b01101011 -> error
0xFF, // 0b01101100 -> error
0xFF, // 0b01101101 -> error
0xFF, // 0b01101110 -> error
0xFF, // 0b01101111 -> error
0xFF, // 0b01110000 -> error
0xFF, // 0b01110001 -> error
0xFF, // 0b01110010 -> error
0xFF, // 0b01110011 -> error
0xFF, // 0b01110100 -> error
0x04, // 0b01110101 -> 0b00000100
0xFF, // 0b01110110 -> error
0x05, // 0b01110111 -> 0b00000101
0xFF, // 0b01111000 -> error
0xFF, // 0b01111001 -> error
0xFF, // 0b01111010 -> error
0xFF, // 0b01111011 -> error
0xFF, // 0b01111100 -> error
0x06, // 0b01111101 -> 0b00000110
0xFF, // 0b01111110 -> error
0x07, // 0b01111111 -> 0b00000111
0xFF, // 0b10000000 -> error
0xFF, // 0b10000001 -> error
0xFF, // 0b10000010 -> error
0xFF, // 0b10000011 -> error
0xFF, // 0b10000100 -> error
0xFF, // 0b10000101 -> error
0xFF, // 0b10000110 -> error
0xFF, // 0b10000111 -> error
0xFF, // 0b10001000 -> error
0xFF, // 0b10001001 -> error
0xFF, // 0b10001010 -> error
0xFF, // 0b10001011 -> error
0xFF, // 0b10001100 -> error
0xFF, // 0b10001101 -> error
0xFF, // 0b10001110 -> error
0xFF, // 0b10001111 -> error
0xFF, // 0b10010000 -> error
0xFF, // 0b10010001 -> error
0xFF, // 0b10010010 -> error
0xFF, // 0b10010011 -> error
0xFF, // 0b10010100 -> error
0xFF, // 0b10010101 -> error
0xFF, // 0b10010110 -> error
0xFF, // 0b10010111 -> error
0xFF, // 0b10011000 -> error
0xFF, // 0b10011001 -> error
```

```
0xFF, // 0b10011010 -> error
0xFF, // 0b10011011 -> error
0xFF, // 0b10011100 -> error
0xFF, // 0b10011101 -> error
0xFF, // 0b10011110 -> error
0xFF, // 0b10011111 -> error
0xFF, // 0b10100000 -> error
0xFF, // 0b10100001 -> error
0xFF, // 0b10100010 -> error
0xFF, // 0b10100011 -> error
0xFF, // 0b10100100 -> error
0xFF, // 0b10100101 -> error
0xFF, // 0b10100110 -> error
0xFF, // 0b10100111 -> error
0xFF, // 0b10101000 -> error
0xFF, // 0b10101001 -> error
0xFF, // 0b10101010 -> error
0xFF, // 0b10101011 -> error
0xFF, // 0b10101100 -> error
0xFF, // 0b10101101 -> error
0xFF, // 0b10101110 -> error
0xFF, // 0b10101111 -> error
0xFF, // 0b10110000 -> error
0xFF, // 0b10110001 -> error
0xFF, // 0b10110010 -> error
0xFF, // 0b10110011 -> error
0xFF, // 0b10110100 -> error
0xFF, // 0b10110101 -> error
0xFF, // 0b10110110 -> error
0xFF, // 0b10110111 -> error
0xFF, // 0b10111000 -> error
0xFF, // 0b10111001 -> error
0xFF, // 0b10111010 -> error
0xFF, // 0b10111011 -> error
0xFF, // 0b10111100 -> error
0xFF, // 0b10111101 -> error
0xFF, // 0b10111110 -> error
0xFF, // 0b10111111 -> error
0xFF, // 0b11000000 -> error
0xFF, // 0b11000001 -> error
0xFF, // 0b11000010 -> error
0xFF, // 0b11000011 -> error
0xFF, // 0b11000100 -> error
0xFF, // 0b11000101 -> error
0xFF, // 0b11000110 -> error
0xFF, // 0b11000111 -> error
0xFF, // 0b11001000 -> error
0xFF, // 0b11001001 -> error
0xFF, // 0b11001010 -> error
0xFF, // 0b11001011 -> error
0xFF, // 0b11001100 -> error
0xFF, // 0b11001101 -> error
0xFF, // 0b11001110 -> error
  0xFF, // 0b11001111 -> error
  0xFF, // 0b11010000 -> error
  0xFF, // 0b11010001 -> error
  0xFF, // 0b11010010 -> error
  0xFF, // 0b11010011 -> error
  0xFF, // 0b11010100 -> error
  0x08, // 0b11010101 -> 0x00001000
  0xFF, // 0b11010110 -> error
  0x09, // 0b11010111 -> 0x00001001
  0xFF, // 0b11011000 -> error
  0xFF, // 0b11011001 -> error
  0xFF, // 0b11011010 -> error
  0xFF, // 0b11011011 -> error
  0xFF, // 0b11011100 -> error
  0x0A, // 0b11011101 -> 0x00001010
  0xFF, // 0b11011110 -> error
  0x0B, // 0b11011111 -> 0x00001011
  0xFF, // 0b11100000 -> error
  0xFF, // 0b11100001 -> error
  0xFF, // 0b11100010 -> error
  0xFF, // 0b11100011 -> error
  0xFF, // 0b11100100 -> error
  0xFF, // 0b11100101 -> error
  0xFF, // 0b11100110 -> error
  0xFF, // 0b11100111 -> error
  0xFF, // 0b11101000 -> error
  0xFF, // 0b11101001 -> error
  0xFF, // 0b11101010 -> error
  0xFF, // 0b11101011 -> error
  0xFF, // 0b11101100 -> error
  0xFF, // 0b11101101 -> error
  0xFF, // 0b11101110 -> error
  0xFF, // 0b11101111 -> error
  0xFF, // 0b11110000 -> error
  0xFF, // 0b11110001 -> error
  0xFF, // 0b11110010 -> error
  0xFF, // 0b11110011 -> error
  0xFF, // 0b11110100 -> error
  0x0C, // 0b11110101 -> 0x00001100
  0xFF, // 0b11110110 -> error
  0x0D, // 0b11110111 -> 0x00001101
  0xFF, // 0b11111000 -> error
  0xFF, // 0b11111001 -> error
  0xFF, // 0b11111010 -> error
  0xFF, // 0b11111011 -> error
  0xFF, // 0b11111100 -> error
  0x0E, // 0b11111101 -> 0x00001110
  0xFF, // 0b11111110 -> error
  0x0F  // 0b11111111 -> 0x00001111
};
```

References

[1] Errol Morris, producer/director. *Fast, Cheap & Out of Control*. Sony Pictures Classics, 1997. 82 minutes.

[2] Joseph Paradiso, Kai yuh Hsiao, Joshua Strickon, Joshua Lifton, and Ari Adler. Sensor Systems for Interactive Surfaces. *IBM Systems Journal, Volume 39, Nos. 3 & 4*, pages 892–914, October 2000.

[3] William Joseph Butera. *Programming a Paintable Computer*. PhD thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, February 2002.

[4] Harold Abelson, Thomas F. Knight, and Gerald Jay Sussman. Amorphous Computing Manifesto. Technical report, MIT Project on Mathematics and Computation, 1996.

[5] Mitchel Resnick. *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Bradford Books/MIT Press, Cambridge, MA, 1994.

[6] Santa Fe Institute. `http://www.santafe.edu`.

[7] Martin Gardner. The fantastic combinations of john conway's new solitaire game "Life". *Scientific American*, 223(10):120–123, 1970.

[8] Cosma Rohilla Shalizi. *Causal Architecture, Complexity and Self-Organization in Time Series and Cellular Automata*. PhD thesis, Physics Department, University of Wisconsin-Madison, May 2001.

[9] James D. McLurkin. Algorithms for distributed sensor networks. Master's thesis, Berkeley Sensor and Actuator Center, University of California at Berkeley, Decemeber 1999.

[10] μAMPS CAD tools. `http://www-mtl.mit.edu/research/icsystems/uamps/cadtools/`.

[11] Amit Sinha and Anantha Chandrakasan. Energy Efficient Real-Time Scheduling. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, November 2001.

[12] Andrew Wang, Seong-Hwan Cho, Charles G. Sodini, and Anantha P. Chandrakasan. Energy Efficient Real-Time Scheduling. In *Proceedings of ISLPED 2001*, August 2001.

[13] Eugene Shih, Seong-Hwan Cho, Nathan Ickes, Rex Min, Amit Sinha, Alice Wang, and Anantha Chandrakasan. Physical Layer Driven Algorithm and Protocol Design for Energy-Efficient Wireless Sensor Networks. In *Proceedings of MOBICOM 2001*, July 2001.

[14] Devasenapathi P. Seetharamakrishnan. A Programming Language for Massively Distributed Embedded Systems. Master's thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, expected in September 2002.

[15] Amorphous Computing's 'Gunk on the Wall'. `http://www.swiss.ai.mit.edu/projects/amorphous/HC11/`.

[16] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT*, pages 114–130, 2001.

[17] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[18] Alec Woo and David E. Culler. A transmission control scheme for media access in sensor networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, Rome, Italy.* ACM, July 2001.

[19] Ya Xu, John Heidemann, and Deborah Estrin. Geography Informed Energy Conmservation for Ad Hoc Routing. In *Proceedings of the Seventh ACM/IEEE International Conference on Mobile Computing (ACM MOBICOM) and Networking*, July 2001.

[20] Jeremy Elson and Deborah Estrin. Time Synchronization for Wireless Sensor Networks. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS)*, April 2001.

[21] Wei Ye, John Heidemann, and Deborah Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, June 2002.

[22] Gabriel T. Sibley, Mohammad H. Rahimi, and Gaurav S. Sukhatme. Robomote: A Tiny Mobile Robot Platform for Large-Scale Sensor Networks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA2002)*, 2002.

[23] Andrew Wheeler. TephraNet: Wireless, Self-Organizing Network Platform for Environmental Science. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2001.

[24] Robert Poor. *Embedded Networks: Pervasive, Low-Power, Wireless Connectivity.* PhD thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, May 2001.

[25] Kwindla Kramer. Moveable Objects, Mobile Code. Master's thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, 1998.

[26] Ember Corporation. `http://www.ember.com`.

[27] Sensoria Corporation. `http://www.sensoria.com`.

[28] J.E. Dowling. *Neurons and Networks: an Introduction to Neuroscience.* The Belknap Press of Harvard University Press, Cambridge, MA., 1992.

[29] W. Keith Edwards and Rebecca E. Grinter. At Home with Ubiquitous Computing: Seven Challenges. In *Proceedings of the Ubicomp 2001: Ubiquitous Computing Conference*, pages 256–272, September 2001.

[30] Zillah Bahar. What 'Smart Dust' Could Do for You. *The Industry Standard*, June 2001. `http://www.thestandard.com/article/0,1902,27573,00.html`.

[31] ARM Ltd. ARM Product Information, ARM9 Thumb Family. `http://www.arm.com`.

[32] Wade Roush. Radio-Ready Chips, All-silicon radios could make everything wireless. *Technology Review*, pages 22–23, June 2002.

[33] Joseph Paradiso. Renewable Energy Sources for the Future of Mobile and Embedded Computing. Invited talk given at the Computing Continuum Conference, San Francisco, CA, March 16, 2000.

[34] Microchip. PIC16F84 Device. `http://www.microchip.com/1010/pline/picmicro/category/digictrl/8kbytes/devices/16f84/`.

[35] Joshua Lifton and Jay Lee. MediaMatrix: Self-organizing Distributed Physical Database. In *Proceedings of the ACM CHI 2001 Conference - Extended Abstracts*, pages 193–194, 2001.

[36] Light & Motion. DIPline power panel. `http://www.lightandmotion.vienna.at/eng-dipline.html`. Donated by Steelcase, Inc.

[37] EMC Process Company, Inc. EMC-232 Process: resin bonded PTFE finish. `http://www.emcprocess.com/coatings/xylan/emc-232.html`.

[38] Bernard Sklar. *Digital Communications, Fundamentals and Applications, Second Edition.* Prentice Hall, 2001.

[39] Cygnal Integrated Products. C8051f016 mixed-signal microcontroller. `http://www.cygnal.com/products/C8051F016.htm`.

[40] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[41] F. Reif. *Fundamentals of Statistical and Thermal Physics.* McGraw-Hill, 1965.

[42] Michael O. Albertson and Joan P. Hutchinson. *Discrete Mathematics with Algorithms.* John Wiley & Sons, Inc, 1988.

[43] V. Bhargavan, A. Demers, S. Shenker, and L. Zhang. MACAW: A Media Access Protocol for Wireless LANs. *SIGCOMM*, pages 212–225, 1994.

[44] Keil Software, Inc. Evaluation Software. `http://www.keil.com/demo/`.

[45] TinyOS. `http://webs.cs.berkeley.edu/tos/`.

[46] J. White. Telescript Technology: An Introduction to the Language. In J. Bradshaw, editor, *Software Agents.* AAAI/MIT Press, 1997.

[47] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[48] 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.

[49] Dejan S. Milojicic, Frederick Douglis, and Richard Wheeler, editors. *Mobility: Processes, Computers, and Agents.* ACM Press, 1999.

[50] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2001.

[51] Jeremy I. Silber. Cooperative Communication Protocol for Wireless Ad-hoc Networks. Master's thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2002.

[52] Berg Electronics. Conan connector, part number 91920-21125.
`http://www.berg.com`.

[53] Robert Poor. Nami - waves of color.
`http://web.media.mit.edu/ r/projects/nami/`, January 2000.

[54] MIT Media Lab, Life Long Kindergarten Group. Programmable Bricks.
`http://llk.media.mit.edu/projects/cricket/about/index.shtml`.

[55] LEGO Corporation. LEGO Mindstorms. `http://mindstorms.lego.com/`.

[56] Joseph Paradiso. Towards Sensate Media and Electronic Skins - testbeds for very high density distributed sensor networks. NSF proposal 0225492, April 2002.