Pushpin Computing System Overview: a Platform for Distributed, Embedded, Ubiquitous Sensor Networks

Joshua Lifton[1], Deva Seetharam[2], Michael Broxton[1], and Joseph Paradiso[1]

[1] MIT Media Lab, Responsive Environments Group, 1 Cambridge Center 5FL, Cambridge, MA 02142 USA
{lifton, mbroxton, joep}@media.mit.edu
http://www.media.mit.edu/resenv/
[2] MIT Media Lab, Physics & Media Group, 20 Ames Street, Cambridge, MA 02139 USA
deva@media.mit.edu
http://www.media.mit.edu/physics/

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.

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]*

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 holding us back from creating machines equipped with as much sensory bandwidth as some animals, if not people. Rather, we are faced with a problem of our own devising – how do we communicate, coordinate, process, and react to the copious amount of sensory data now available to the machines we build? Certainly, some success in harvesting and responding to multiple data streams originating from a quantity of sensors has been demonstrated (e.g. [2]), but such examples do not scale; using traditional sensing methods, even adding one more sensor to an array of a couple dozen sensors presents a formidable challenge on both the hardware and software fronts. As the number of sensors increases to the thousands, hundreds of thousands and beyond, any tractable solution will have to rely on principles of self-organization at the level of the sensors themselves in order to guarantee the proper scaling properties. In this sense, 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.

Distributed sensor networks are immediately relevant to many real world applications; 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 areas 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 any number of fields will only make that list longer.

In this paper we introduce the Pushpin Computing platform as a general purpose hardware and software toolkit for studying, designing, prototyping, and deploying dense sensor networks. Details of the hardware and programming model are given, as well as the design considerations that lead up to the current implementation. A simple example is illustrated step by step.

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 general 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 are either tangential or very closely related to the work presented here.

The direct inspiration for this work is Butera's Paintable Computing simulation work [3]. 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, according to Butera, 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. Each 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. From these simple postulates, Paintable Computing demonstrates the utility of *algorithmic self-assembly* to build up complex global behavior across the system as a whole from simple local interactions among process fragments that migrate among the processing nodes. Pushpin Computing started out as an attempt to instantiate in hardware as closely as possible the Paintable simulations, each Pushpin corresponding to a single processing node. This will be discussed further in the coming sections.

Resnick's StarLogo programming language [4] provides an accessible but rich simulation environment for exploring decentralized emergent systems. The Pushpin programming model is influenced by StarLogo's intuitive approach.

Although there are surely many more examples of computer simulation research that have some bearing on distributed sensor networks, Berkeley's (now Intel Research Lab at Berkeley) SmartDust and its associated TinyOS software environment is the only known hardware platform developed in a spirit at all similar to that of the Pushpins. 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 [5, 6]. 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 [7]. In contrast, the Pushpin platform was built more from the top down, provides each node with a rich set of onboard peripherals, bandwidth, and software, and consumes correspondingly more energy per node.

3 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. 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 [8].

The compelling architecture articulated and demonstrated in simulation by the Paintable Computing project provides a base set of design points for the hardware, operating system, and programming environment from which it is possible to build a distributed sensor network that achieves the goal of self-organization. Where practical, the Pushpin platform follows this blueprint closely. To paraphrase [3]:

- Each Pushpin (node) has the ability to communicate locally with its spatially proximal neighbors, the neighborhood being defined by the range of the mode of communication employed.
- Each Pushpin must reliably handle the fact that the number of addressable neighbors in the communication neighborhood can vary unpredictably.
- Each Pushpin must reliably handle the fact that messages sent to its neighbors may exhibit probabilistic transit times and are not explicitly acknowledged.
- Each Pushpin must provide for a mechanism for installing, executing, and passing on to its neighbors code and data received over the communication channel.

In addition, the Pushpin platform is designed specifically for ease of prototyping a wide range of digital and analog applications, so it can readily serve as a testbed for practitioners coming from many perspectives.

4 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. The Pushpin hardware platform is designed around a balanced optimization of small physical footprint, functional modularity, expandability, generality, and computational power. To this end, each Pushpin consists of four modules that separately handle power, communication, processing, and application-specific functions. Each module is contained on a printed circuit board (PCB) measuring roughly 18mm x 18mm and stacks together with other modules vertically from bottom to top in the order listed. See Fig. 1. The total stacked height of a Pushpin varies depending on the modules used, but is typically on the order of 18mm as well. A description of each module and the connections between them follow.

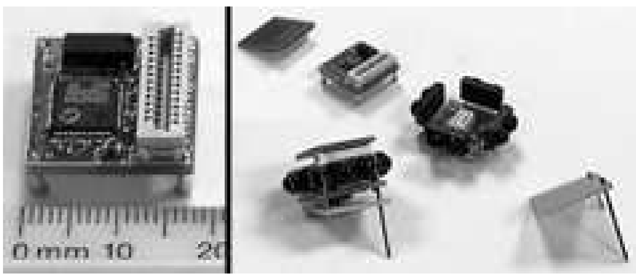

Fig. 1. Modules of a Pushpin

4.1 Power Module

The Pushpin moniker derives from the original power scheme implementation in which 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 [9]. 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) 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.

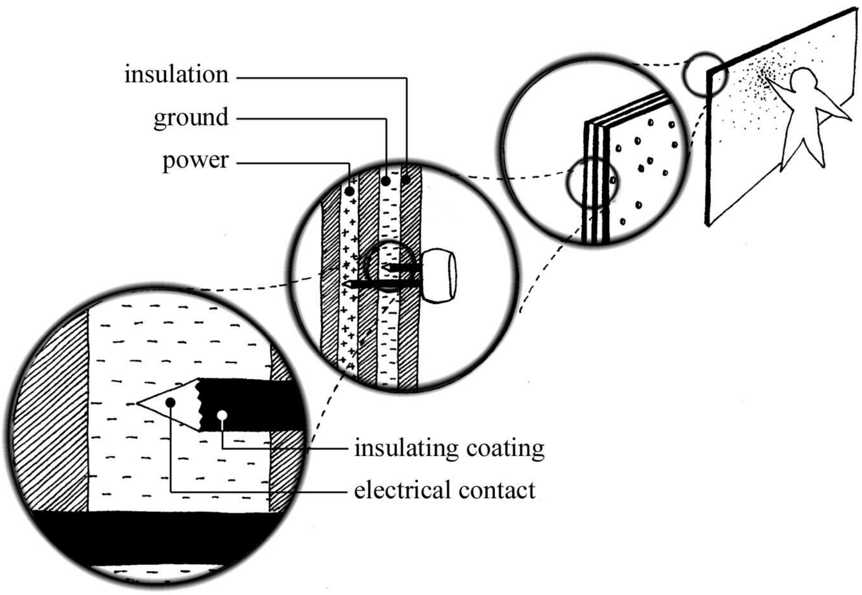

Fig. 2. Pushpin power scheme

Other power sources can easily take the place of the pins and substrate as long as they provide 2.7VDC to 3.3VDC. Two AAA batteries in series is a simple, if bulky alternative. 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. Typical current consumption of the processing module running at 22MHz with all necessary peripherals enabled is roughly 10mA, whereas the processing module running in a low-power mode off of an internal 32kHz clock requires roughly 10μA. With the clock shutdown, this falls to about 5μA. Accordingly, the lifespan of a power source can vary from hours to years depending on the particular circumstances.

4.2 Communication Module

Anything containing all the necessary hardware for effectively transmitting from and receiving to a typical hardware UART 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, several communication modules are available for Pushpins, including a capacitive coupling module and an infrared module, both of which run at up to 166kbps. See Fig. 3. A radio module is under development. There is also an interface for RS232 communication with a PC over a serial port.

Fig. 3. Pushpins equipped with IR communication modules (and white diffuser rings) drawing power from the laminate substrate

4.3 Processing Module

The Pushpins are designed around the Cygnal C8051F016 – an 8-bit, mixed-signal, up to 25 MIPS, 8051-core microprocessor [10]. The Cygnal chip is equipped with 2.25-Kbytes of RAM and 32-Kbytes of in-system programable (ISP) flash memory. 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 22.1184MHz 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 comprise the remainder of the processing module. See Fig. 4.

4.4 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, comparators, analog-to-digital converters, 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. Examples include sonar transducers, LED displays, microphones, light sensors, and supplementary microcontrollers.

5 Programming Model

The Pushpin programming model is heavily informed by the Paintable Computing programming model [3] and attempts to follow it as closely as possible. The occasional deviations from that model are due to somewhat limited computational resources and reasons of practicality. In essence, the programming model is based on algorithmic self-assembly; the idea that small algorithmic process fragments with 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 [11]; when the number of particles is large, $pV = nRT$ becomes more useful than knowing the position and momentum of each particle.

The Paintable Computing project successfully demonstrated algorithmic self-assembly in simulation. The goal of the Pushpin programming model is to create a suitable tool for exploring algorithmic self-assembly as it relates to sensory data extracted from the real world. To this end, an operating system, networking protocol, and process fragment integrated development environment (IDE) have been implemented.

5.1 Process Fragments

A process fragment is the atomic algorithmic unit in algorithmic self-assembly. Carrying the thermodynamics analogy further, a process fragment corresponds to a single gas particle. A process fragment ('pfrag') is defined as the coupling of state information ('state') and executable code ('code'). A pfrag's code acts on or according to the pfrag's state and has the ability to modify it. A process fragment is entirely contained and executed within a single Pushpin, but may transfer or

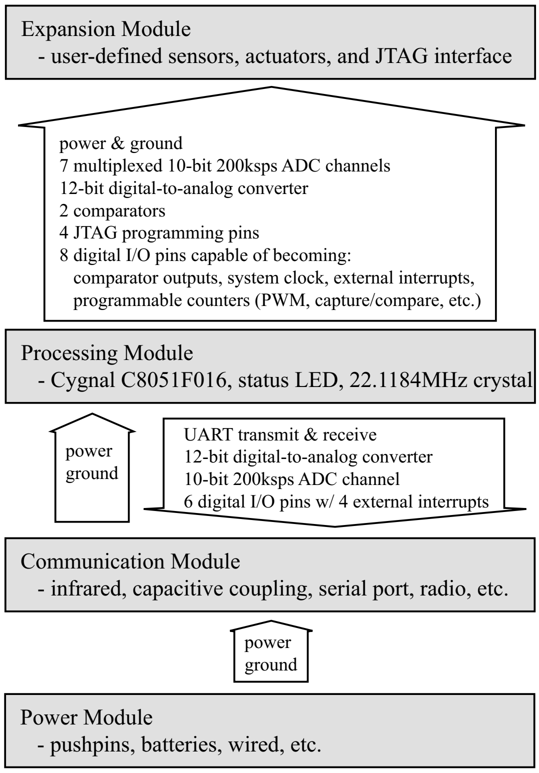

Fig. 4. The Pushpin hardware specification. The shaded boxes represent different hardware modules. The arrows represent resources that the module at the tail of the arrow provides to the module at the head of the arrow

copy itself to neighboring Pushpins and begin execution there. In order to ensure interoperation between process fragments and the Pushpin operating system (Bertha), process fragments must conform to the following constraints:

- Implement an *install* function to be called by Bertha when the process fragment is first executed in a given Pushpin.
- Implement a *deinstall* function to be called by Bertha when the process fragment is to be removed from a given Pushpin.
- Implement an *update* method to be repeatedly called by Bertha as long as the process fragment resides within a Pushpin. There is no guarantee how often the update function will be called, only that it will be called. This is where most of the functionality of a process fragment resides.
- Total process fragment code size limit of 2-Kbytes.
- Total process fragment state size limit of 445-bytes.

Aside from the required functions, process fragments may also contain as much private code as the 2-Kbyte limit allows.

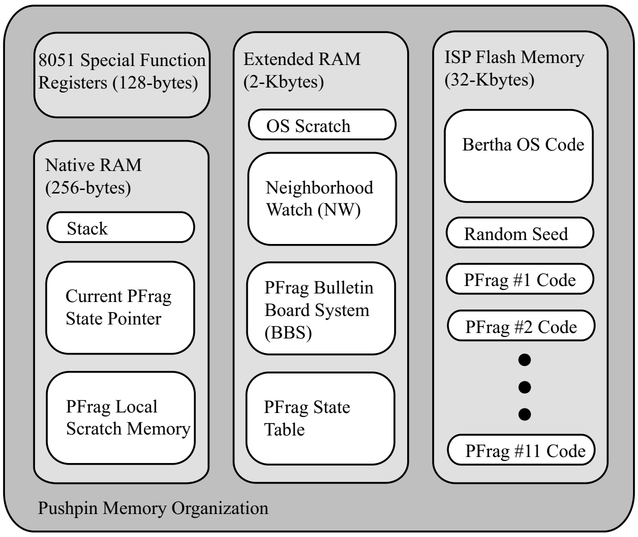

Fig. 5. A Pushpin's memory, carefully divided between process fragments and the operating system

5.2 Bertha: The Pushpin OS

Underlying system operation is handled by Bertha – a small, lightweight operating system developed especially for the Pushpins. Each Pushpin has its own instance of Bertha to manage processor startup, memory, access to hardware peripherals and system services, communication with neighboring Pushpins, and, its primary charge, resident process fragments.

Bertha can accommodate up to 11 process fragments at any given time. Process fragments enter a Pushpin through the communication port either wirelessly via a neighboring Pushpin or from a device pretending to be a Pushpin. The process fragment is written to memory (code to flash memory and state to RAM), checked for errors by means of a simple checksum, added to the list of resident process fragments (assuming the checksum passes), and initialized by calling its *install* function. Bertha executes the *update* function of resident process fragments using a simple round-robin scheme. Each process fragment is allowed to run its *update* function to completion each time it is called. Bertha provides various utility system functions to process fragments, such as those that return the current system time or a pseudo-random number.

Bertha also negotiates all communication on the behalf of process fragments. Specifically, it provides for communication between process fragments in the same Pushpin by means of a bulletin board system (BBS). By making system calls to Bertha, process fragments can post arbitrary messages of limited size to the BBS and read messages posted by other process fragments. A Pushpin's BBS can be posted to and read from only by process fragments within that Pushpin. Bertha does, however, maintain a Neighborhood Watch – a list of neighboring Pushpins (those within communication range) and brief synopses of their BBSs. The information contained in each neighbor synopsis is culled from that neighbor's own BBS. Due to memory constraints, it is not possible to mirror the entirety of all neighboring Pushpins' BBSs. Instead, whenever a process fragment posts to the local BBS, it has the option of marking that post to be included in the synopsis sent out to neighboring Pushpins. Bertha is responsible for arbitrating which of these posts get included in the synopsis in the case of the synopsis filling up. Currently, Bertha gives priority to newer posts, although this does not have to be the case and process fragments should not assume any particular method for choosing what is included in the synopsis.

Process fragments can make a request of Bertha to transfer them to one of the Pushpins listed in the Neighborhood Watch. When such a request is made, Bertha adds the request to the queue, waits until all resident process fragments have been updated, and then negotiates each transfer request with the appropriate neighbor. No guarantee is made that the transfer will be granted.

At a low level, Bertha manages the Pushpin's half-duplex communication channel with its neighbors using a simple exponential back-off protocol for collision avoidance. Bertha attempts to detect collisions with a simple checksum. To help alleviate the hidden node problem, Bertha is able (when using the capacitive coupling module, but not the IR module) to listen for transmissions from neighbors at a variable threshold. When enabled, Bertha listens at a very low threshold before transmitting and a very high threshold when receiving.

An analog-to-digital converter (ADC) channel in conjunction with a simple voltage divider allows the Pushpin operating system to detect which communication and expansion modules make up the Pushpin (as each type of module produces a characteristic voltage read by the ADC), making for plug-and-play functionality. Once Bertha knows what kind of hardware it is dealing with, it provides mediated access of those resources to resident process fragments. Thus, a process fragment can request to be informed during its next update cycle of a given interrupt being triggered or of a certain condition occurring. Process fragments can also take control of certain hardware peripherals, such as general purpose I/O pins, comparators, and analog-to-digital converter channels.

Since even some of the simplest algorithms already mentioned (e.g. exponential back-off) require randomness, Bertha maintains a 1024-bit seed for use in a pseudo-random number generator. (The size of this seed is unnecessarily large due to an artifact of the hardware organization of the flash memory). This seed can be changed during runtime.

See Fig. 5 for a schematic view of the memory layout of a Pushpin and its operating system.

5.3 Pushpin IDE

Users can create custom process fragments using the Pushpin integrated development environment (IDE). The Pushpin IDE is a Java program that runs on a desktop PC. Process fragment source code is authored within the IDE using 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 §5.1.

Currently, the Pushpin IDE calls upon a free evaluation version of the Keil C51 compiler and Keil BL51 linker [12] 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 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.

5.4 Security

One of the first observations that can be made about the Pushpin programming model is that it is incredibly insecure by almost any definition of insecure – Bertha runs any well-formed process fragment as raw machine code without any supervision. The only attempt at security is locking the flash memory containing the Bertha code so that it can't be overwritten by a process fragment. Everything else is fair game. Furthermore, there is no built-in protection against rogue process fragments with malicious intent. While security is certainly a valid concern for any system deployed in the world outside of a testbed running in a research lab, it is assumed for now that everyone authoring process fragments received the "plays well with others" stamp of approval. Although security for sensor networks is essentially ignored here, some work has been done on the subject [13]. That said, the Pushpin platform could be used in its own right to explore security issues.

6 Example: Network Gradient

To clarify the idea of process fragments and Pushpin platform operation, we present here a very simple example. The following code fragment simply copies itself to all its neighbors, keeping track of how many hops away it is from its Pushpin of origin. Its *install* routine does almost all the work. Its *update* routine copies the process fragment to neighboring Pushpins. All other required routines are implemented with default routines provided by the Pushpin IDE. The Pushpin IDE also registers this process fragment as GRADIENT with a local process fragment registry it keeps. What follows is the process fragment source code as it would appear in the IDE.

```
state {unsigned char hopsFromOrigin; unsigned char origin;}

globalID {GRADIENT;}

// Continually attempt to migrate to neighboring Pushpins.
//
unsigned int update(unsigned int eventCode, unsigned int eventValue) {
  return requestTransfer(TO_ALL_NEIGHBORS);
}

// Upon waking up in a Pushpin, check to see if there are any
// copies of this PFrag.  If so, compare hops from origin,
// keep lowest hop count, and delete yourself.  If not, check
// if you are the seed of the gradient and set hop count
// accordingly.
//
unsigned int install() {
  BBSPost post;
  getBBSPost(GRADIENT, &post);
  if (isValidBBSPost(&post)) {
    if (post.localID != getLocalID()) {
      if ((post.message[0] > state.hopsFromOrigin + 1)
          && (post.message[1] == state.origin)) {
        post.message[0] = state.hopsFromOrigin + 1;
        updateBBSPost(&post);
      }
      die();
    }
  }
  else {
    post.message[0] = state.hopsFromOrigin + 1;
    if (!isValidMessage(getNeighborMessagePostedBy(GRADIENT))) {
      post.message[1] = getPushpinID();
    }
    else {
      post.message[1] = state.origin;
    }
    postToBBS(&post, 2);
  }
  return 1;
}
```

Note that, for the sake of brevity, this process fragment is implemented in quite an inefficient manner in terms of bandwidth usage and could be improved upon with some effort.

7 Conclusions & Future Work

This paper describes the basic elements of the Pushpin Computing platform, the first hardware instantiation of an environment specifically designed to support algorithmic self-assembly for use in dense sensor networks. In particular, we have introduced the underlying Pushpin hardware and Bertha, a fully functional embedded operating system that supports mobile process fragments.

The work presented is more of a look at things to come than a culmination or conclusion of things that were. In the immediate future, there are plans to implement a Logo virtual machine on the Pushpins, improve error correction and detection, and build several complete networking and sensing applications using on the order of 100 Pushpin nodes. Longer term goals include exploring the potential of Pushpins as a tangible interface, characterizing basic algorithmic elements vital to algorithmic self-assembly in the context of dense sensor networks, and providing a theoretical foundation to describe self-assembly as a general phenomenon.

Detailed information about the Pushpin Computing project can be found at `http://www.media.mit.edu/~lifton/Pushpin/`.

Acknowledgments

The authors would like to give thanks to the sponsors of the MIT Media Lab for their generous support, and in particular to Steelcase, Inc. for donating the power substrate used in the Pushpin project. Special thanks goes to Dr. Bill Butera for his unflagging encouragement, openness, patience, and effort, without which this project would not be where it is today.

References

1. Morris, E.: *Fast, Cheap & Out of Control*, Sony Pictures Classics, 1997.
2. Paradiso, J.; Hsiao K.;, Strickon J.; Lifton, J.; Adler A.: *Sensor Systems for Interactive Surfaces*, IBM Systems Journal, Volume 39, Nos. 3 & 4, pp. 892-914, October 2000.
3. Butera, W.: *Programming a Paintable Computer*, MIT Media Laboratory, doctoral dissertation, 2002.
4. Resnick, M.: *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, The MIT Press, 1994.
5. Culler, D; Hill, J; Buonadonna, P.; Szewczyk, R.; Woo, A.: *A Network-Centric Approach to Embedded Software for Tiny Devices*, to appear in DARPA workshop on Embedded Software.
6. Hill, J; Szewczyk, R; Woo, A.; Hollar, S.; Culler, D. Pister, K: *System Architecture Directions for Networked Sensors*, 27 April 2000.
7. Woo, A.; Culler, D.: *A Transmission Control Scheme for Media Access in Sensor Networks*, Mobicom 2001.
8. Dowling, J.: *Neurons and Networks: An Introduction to Neuroscience*, Chapter 14, Harvard University Press, 1992.
9. Dipline power panel. Donated by Steelcase, Inc. `http://www.lightandmotion.vienna.at/eng-dipline.html`
10. Cygnal C8051016 microprocessor. `http://www.cygnal.com/products/C8051F016.htm`.
11. Reif, F.: *Fundamentals of Statistical and Thermal Physics*, McGraw-Hill, 1965.
12. Keil Software, Inc. `http://www.keil.com/demo/`
13. Perrig, A.; Szewczyk, R.; Wen, V.; Culler, D.; Tygar, J.: *SPINS: Security Protocols for Sensor Networks*, Mobicom 2001.