NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-09-01
January 31, 2009

GOTO Considered Helpful: A BASIC Approach To Sensor Network Node Programming

J. Scott Miller **Peter A. Dinda** **Robert P. Dick**

Abstract

Sensor networks have the potential to empower domain experts from a wide range of fields. However, presently they are notoriously difficult for these domain experts to program, even though their applications are often conceptually simple. We address this problem by bringing the BASIC programming language to sensor networks. BASIC has proven highly successful in the past in allowing even children to write useful programs on home computers. Our contributions include: (1) a BASIC implementation for modern sensor networks, (2) the first-ever user study evaluating how well novice (no programming experience) and intermediate (some programming experience) users can accomplish simple sensor network tasks in our BASIC and in TinyScript (an alternative also designed for inexperienced programmers), and (3) an evaluation of power-consumption issues in interpreted languages like BASIC. 45–55% of novice users can complete simple tasks in BASIC, while only 0–17% can do so in TinyScript. Users generally succeeded best using imperative loop-oriented programming. The use of an interpreter, such as our BASIC implementation, has little impact on the power consumption of applications in which computational demands are low. Further, when in final form, BASIC can be compiled to reduce power consumption even further.

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-0721978.

Keywords: sensor networks, programming languages, BASIC

GOTO Considered Helpful: A BASIC Approach To Sensor Network Node Programming

J. Scott Miller Peter A. Dinda Robert P. Dick
{jeffrey-miller,pdinda,dickrp}@northwestern.edu
Department of Electrical Engineering and Computer Science
Northwestern University

ABSTRACT

Sensor networks have the potential to empower domain experts from a wide range of fields. However, presently they are notoriously difficult for these domain experts to program, even though their applications are often conceptually simple. We address this problem by bringing the BASIC programming language to sensor networks. BASIC has proven highly successful in the past in allowing even children to write useful programs on home computers. Our contributions include: (1) a BASIC implementation for modern sensor networks, (2) the first-ever user study evaluating how well novice (no programming experience) and intermediate (some programming experience) users can accomplish simple sensor network tasks in our BASIC and in TinyScript (an alternative also designed for inexperienced programmers), and (3) an evaluation of power-consumption issues in interpreted languages like BASIC. 45–55% of novice users can complete simple tasks in BASIC, while only 0–17% can do so in TinyScript. Users generally succeeded best using imperative loop-oriented programming. The use of an interpreter, such as our BASIC implementation, has little impact on the power consumption of applications in which computational demands are low. Further, when in final form, BASIC can be compiled to reduce power consumption even further.

1. INTRODUCTION

Wireless sensor networks (WSNs) can be viewed as general purpose distributed computing platforms defined by their spatial presence and an emphasis on environment monitoring. The most prominent applications of sensor networks have thus far included monitoring applications with a variety of requirements, although WSNs need not be limited to these tasks. While WSNs are currently of great interest to the research communities concerned with the design and implementation of efficient methods of distributed communication and computation, it is ultimately communities and users *outside* of these areas—application domain experts—that have the most to gain from the functionality that WSNs can provide. We focus on application domain experts who are programming novices and *not* experts on WSNs, and who cannot afford such experts. Our goal is to make the development of WSN *applications* by such individuals and groups tractable and, ideally, straightforward.

To deliver the power of wireless sensor networks into the hands of such application domain experts, the barrier to entry must be modest. In terms of raw hardware, this point has already mostly been reached, provided custom hardware is not needed. However, through our interaction with a civil engineering group that is designing, implementing, and deploying an autonomous crack monitoring application [20, 10, 11, 2], we have become convinced that sensor network programming languages and systems have not yet reached this point. Current languages require knowledge of either very low-level systems development (including the details of sensor hardware and embedded system design), or high-level programming concepts and abstractions that are not obvious to most application domain experts, who generally have little programming experience almost all of which is with with simple single-threaded imperative programming models. Regardless of the language, the developer must keep in mind many of the details that motivate continued WSN research such as the reliability of communication and power consumption.

It cannot be assumed that an application domain expert who stands to benefit from a WSN possesses a background in embedded systems development, can devote time to a programming curriculum, or has the funds to hire an embedded systems expert. Even if such an expert is available, the capabilities of a sensor network are tightly coupled to its hardware and software design, making any disconnect between the application domain expert and embedded systems expert a barrier to achieving the domain expert's goals.

It is vital that application domain experts not be confused with traditional application developers. A Unix, Windows, or web developer may be able to stretch his capabilities to write a WSN application. For example, someone familiar with writing Microsoft Windows applications in C++ or C# already has some of the conceptual framework needed to grasp event-driven programming in a C-like language on a sensor network node. We believe that application domain experts generally start much closer to zero than application programmers. For this reason, it is unlikely that programming languages and concepts that have found strong adoption and demonstrated productivity gains in the WSN or general software development communities will give application domain experts similar results. We can only assume that the application domain expert will remain a perpetual novice[1], or, at best, an intermediate programmer.

The sensor network research community has made several efforts to simplify the development of WSN applications by creating a range of languages and programming systems designed specifically for the platform (see Section 2 for more). These languages span a number of programming paradigms and each targets a different type of developer. However, as far as we are aware, there do not exist any studies of the efficacy of these languages and systems when used by application domain experts. Indeed, we are also unaware of proposed metrics for such assessment.

We are addressing this shortcoming by bringing novices, including application domain experts as previously described, into the core of the language design and implementation "loop" via rigorous user studies. We run user studies to

[1]We use this term in the sense meant by Dineh Davis [8], who argues that instead of searching for ways to make users experts in the use of computers and technology, we should seek ways to make them better novices.

- evaluate how our target users respond to different languages and systems,
- determine how quickly and correctly they can complete tasks using each language and how power-efficient the solutions are, and
- inform future language, system, and interface design targeting these users.

As far as we are aware, we are the first to evaluate a sensor network programming language and system in this fashion.

This paper focuses on our first iteration in applying user studies to the problem of making WSNs easy to program by application domain experts. We focus on the problem of programming individual nodes, including sensing, sending information back to a centralized aggregator, and node-based actuation. Although this problem is more limited in scope than general WSN programming, and the kinds of programming supported by other languages/programming systems, it is nonetheless an interesting problem. As we describe in Section 2.3, a study of extant sensor network applications suggests that node-programming of this kind has very broad applicability.

Extrapolating from the undeniable success that the BASIC programming language had in the late 1970s and early 1980s in engaging extreme novices—children—in programming, and the similarities between the home computer platforms of that era and the sensor network nodes of today, we consider the use of BASIC as a WSN node programming language. Our specific contributions are as follows.

- We ported a small BASIC interpreter to a modern sensor network node and operating system, extending the language and implementation with simple features for communication, power management, sensing, and actuation.
- We created a simple integrated development environment (IDE) for our port, as well as a tutorial for our extended BASIC, both targeting the kinds of users described earlier.
- We evaluated our extended BASIC, IDE, and tutorial by conducting a rigorous user study involving novices.
- We also evaluated TinyScript, a high-level, event-driven node-level programming language for sensor networks using a set of exercises identical to those used to evaluate BASIC.
- We measured the computational and power costs involved in using our interpreter.

It is important to point out that the user studies noted above are, as far as we are aware, the first ever done to evaluate sensor network programming languages. Beyond letting us evaluate the utility of our BASIC, they also provide a useful characterization of user reaction to TinyScript, and prototypes for future studies.[2]

Our evaluation found the following.

- Novice users are able to use our system to implement simple sensor network programs on MicaZ motes that include data acquisition, communication, and actuation tasks.
- While results depend on the nature of the task, 45-55% of novice users are likely to complete simple tasks in BASIC, while only 0-17% are likely to do so in TinyScript.
- Participants with programming experience had similar rates of success using BASIC and TinyScript.
- Many participants struggled with developing applications using an event-driven programming model.
- While our system incurs a significant computational overhead (the interpreted code is, not surprisingly, much slower than compiled C), for common application patterns in which the hardware spends significant time asleep, this overhead and its concomitant power costs are negligible. A "sense-and-send" task with a one second period, for example, consumes only 1.5% more power when written in BASIC.

This experience underlines the value of using user studies to evaluate languages and programming systems targeting application domain experts. Our work shows that there is value in using BASIC-like languages in the sensor network domain and more broadly identified some of the language features most appropriate for enabling novice programmers.

2. RELATED WORK

The architectural visions of Hill et al [18]; Polastre, Szewczyk, and Culler [36]; as well as Cerpa and Estrin [5] have had great impact on sensor network research and design. Our work is more specifically related to work on sensor network programming languages, measures of software engineering productivity, and existing applications.

2.1 Sensor network programming languages

There are a number of programming languages, support libraries, and operating systems for sensor network nodes [15, 27, 1, 25, 6, 24, 13]. They provide support for modular programming and the use of hardware modules, reaction to events, and some degree of network abstraction. Some languages focus on permitting specification of network-wide behavior instead of specifying the behavior of individual components [16, 33]. Bonivento, Carloni, and Sangiovanni-Vincentelli propose a platform-based design methodology for wireless sensor networks [4]. Recent improvements to sensor network programming environments have been substantial. However, these advances primarily benefit embedded system design and programming experts, not application domain experts.

Existing sensor network programming languages are explicitly designed to ease the development and deployment of sensor applications. The languages borrow their semantics from well-known programming paradigms, including structured query, functional and event-driven styles. The languages differ in the abstraction they provide for the underlying sensor network, treating the network as either a single logical machine or a collection of communicating entities.

The Regiment [32], TinyDB [29], and Tables [19] languages are examples of macro-programming languages in which the developer writes code that targets the entire sensor network. Heterogeneous execution emerges based on local conditions. Regiment follows a functional programming design that treats each sensor as a stream of data. Regiment allows programmers to partition streams into logical neighborhoods based on network proximity, allowing event detection that spans multiple sensors. In TinyDB, the programmer writes queries to a logical database table representing sensor values across the network. TinyDB's SQL-like syntax is assumed to be familiar to application developers.

[2]All study materials will be made available for this purpose.

Tables, a framework for programming sensor networks that uses a spreadsheet model (specifically, pivot tables) to describe tasks, takes a similar approach.

In contrast to such network-level languages, NesC [18], TinyScript [26], and embedded C languages are node-level programming languages, targeting individual sensors. In practice, however, the model is SPMD (Single Program Multiple Data)—the same code generally runs on all the nodes in the network with the possible exception of a base-station that acts as an accumulator of sensor data. All of these languages provide an imperative syntax. In both NesC and TinyScript, high-level program flow is controlled through events that are triggered by communication, timers or are user-defined.

Aside from their scope, these node-level languages target different kinds of programmers. The C-like syntax of NesC is more appropriate for programmers with strong C backgrounds. NesC also uses a form of event-driven programming that seasoned developers might be accustomed to but is unfamiliar to a large class of novice programmers. TinyScript adopts a more simplified set of semantics in order to make the NesC model more approachable for novices.

In addition to these node-level languages, higher level application programming languages have also made inroads on the WSN space. The Micro .NET Framework [30] and Java Sun SPOTS [31] platforms leverage the C# and Java languages, respectively, which should be familiar to a range of experienced application developers.

The present work focuses on node-level programming languages and systems for *novice users*, including application domain experts. For the most part, the network-level and node-level languages just described have the goal of making *expert developers* more efficient. The exceptions are TinyScript (node-level), on which we elaborate below, and Tables. Tables specifically targets naive users. In contrast with our work, however, Tables is a network-level programming system, and, to the best of our knowledge, has not yet been evaluated in a user study.

TinyScript. The goals of the present paper most closely resemble those of TinyScript [26], a high level programming language that is compiled to the byte-code of the Maté virtual machine platform for sensor networks [25]. The creators of TinyScript were early to expose the interesting problem that we are now working on: how can one design a language to make sensor network programming more accessible? Their answer, TinyScript, is a dynamically typed imperative language with an event-driven programming model. TinyScript applications result in relatively few high-level Maté instructions, allowing for straightforward application distribution and updates within a sensor network.

Our work differs from TinyScript in several ways. First, BASIC is a simple imperative language with no event model. We have added minimal extensions to support node-level programing. Our implementation is a simple interpreter (which leverages the uBASIC codebase of Adam Dunkels [12]) with no underlying byte-code virtual machine.

A second difference is that we have focused, both in the language and in the presentation of the language via the IDE, on reducing complexity for shorter programs. A program in our system is represented as a single source code file, displayed (and hidden) in a custom IDE. All control flow is in this file and is immediately visible to the programmer. In contrast, in TinyScript, the programmer creates a separate code block for each handled event, with code in one handler being able to interact with that in another. The TinyScript IDE further separates each event handler by allowing the programmer to view and modify only one handler at a time. There is no notion of scope in our BASIC—all variables are at global scope. In contrast, variables in TinyScript are either locally scoped to each event handler or globally scoped across all handlers. All variables in our BASIC are the same type (integer). In contrast, in TinyScript, data is represented by several types and the application developer must at times explicitly convert among types.

The extreme simplicity of BASIC makes it unsuitable for the development of large software projects, and event-driven control flow, scoping, and typing should be minor issues for an experienced application developer. However, WSN nodes are very resource constrained, so large software is physically impossible, and our target user is the novice, for whom events, scoping, and typing are challenges.

We show here how these differences affect application development for novice and intermediate users by carrying out user studies comparing BASIC and TinyScript. We are unaware of a study of the effectiveness of TinyScript from the user perspective. Evaluating our system with novice and intermediate users is a critical component of our work. We have intentionally sought out novices both to understand how well our design succeeds at enabling sensor network development by them, and to inform future language improvements.

2.2 Productivity measures

Although a range of software engineering metrics exist [14, 21], we are unaware of any proposed metric or benchmark for sensor network application programming by novices. Perhaps closest is work in developing metrics for evaluating students in introductory programming courses [9], but this doesn't consider the power concerns and environmental coupling of sensor network application programming.

All of the sensor network programming languages and systems discussed earlier include abstractions whose aim is to simplify the process of writing code. Across this varied landscape of languages, there is little quantification of how well each language suits the needs of different user communities, particularly application domain experts acting as novice programmers. As far as we are aware, there is no agreed upon set of benchmarks to assess the strengths or weaknesses of each language. This degree of choice is common among languages targeting expert programmers, but probably overwhelming for novices. Our work includes the rudiments of an evaluation strategy that could provide solid data for ranking languages/systems in terms of their utility for novices.

2.3 Existing applications

Node-oriented programming languages, such as our BASIC implementation, are suitable for a set of applications in the WSN design space. This set of applications is usually based on homogeneous systems in which each node has the same functionality. Many existing WSN applications fall in to this category For example, structural health monitoring applications [10, 23], environmental monitoring applications [37, 38, 17], and animal tracking applications [28].

With proper abstractions to hide communication details, it is likely that node-oriented programming complexity can be acceptable to novice programmers.

3. WHY BASIC?

As its name implies, the Beginners All-purpose Symbolic Instruction Code (BASIC) is specifically designed to provide programmers with a language that is complete, simple and easy to understand [7]. A grammar for early versions of BASIC confirms this: the language does not contain features now considered necessary for the creation of large, maintainable applications such encapsulation, user-defined types or in some cases even local variables.

Why then should we give users a programming language that is primitive when compared to its more modern peers? Many programming constructs have come about to promote code maintainability as an application grows in size. For sensor network applications characterized by relatively simple high-level behavior, such features are not necessary. Maintainability of tiny code bases is often very easy regardless of language.

The simple syntax offered by BASIC minimizes the number of concepts that need to be understood by the novice programmer. The application domain expert views programming sensor nodes as a means of conducting a single aspect of her research. Given this, she has minimal time to spend in learning programming concepts, particularly if they are not critical for the typically small programs she writes. Furthermore, if the frequency at which she programs is sufficiently low, it is not unreasonable that the domain expert will quickly forget programming concepts. Thus it is vital that she be able to quickly "context switch" into programming, even if she hasn't done it for a while. BASIC's extreme simplicity has the advantage of making this easier.

It is important to point what specifically we mean by BASIC. Although its original developers, John Kemeny and Thomas Kurtz, feel that the language has been corrupted since its origins at Dartmouth in the 1960s [22], it is probably more accurate to say that BASIC has come to refer to an extremely widely varying family of languages. We are interested in the minimal, interpreted form of BASIC that emerged on home computers in the late 1970s. Those early 8-bit "microcomputers" were resource constrained in much the same way as modern sensor network nodes. At its most minimal this was TinyBASIC [35], which essentially the base language of the work described here.

4. IMPLEMENTATION

Our platform is based on uBASIC [12], an open source BASIC interpreter developed by Adam Dunkels and available under BSD license. uBASIC is written in C and is designed for embedded systems. The grammar of uBASIC resembles that of the TinyBASIC programming language. As described previously, TinyBASIC is a simplified dialect of BASIC designed for resource constrained computing environments and provides only simple programming constructs. We chose this variant of BASIC because its small memory requirement enables us to target a wide range of sensor platforms. Dunkels' web site indicates that uBASIC is eventually intended for use in "adding a simple scripting language to severely memory-constrained applications or systems (e.g. a scripting language to the web server applications in uIP or Contiki)". Our purpose is to see whether novice programmers can write simple sensor network node applications using BASIC.

We ported the core interpreter to the Mantis Operating System [3], a sensor network operating system that provides a consistent API for reading sensor values, network communication and threading. The original code combined with our extensions spans approximately 2000 lines of code. After compilation for the Crossbow MicaZ, our interpreter occupies 38kB of flash. For comparison, an empty Mantis application has a 29kB footprint when compiled with the same libraries. Our implementation is sufficiently compact that we may later add more complex protocols for routing and power management.

Our interpreter must be programmed directly through the serial connection. This allows for rapid development and testing but limits the extent to which motes can be reprogrammed after being deployed. We are considering adding programming via the network, which should not be onerous as the interpreter stores the program in simple string form.

4.1 Language design and extensions

We extended the BASIC grammar to include statements necessary for sensor network applications. Reading sensor values occurs with the SENSE statement, which take an expression indicating which sensor to read from (onboard MicaZ sensors are supported) and the name of the variable in which the value is to be stored. An analogous ADC statement includes for reading arbitrary ADC channels. The SENSE and ADC statements follow the semantics of the INPUT statement found in almost all BASIC implementations.

The LED statement controls the onboard LEDs on the platform. The LED statement is followed by a number specifying which LED is being manipulated and an expression indicating if the LED is to be turned on or off. A similar BUZZ statement exists to control the sounder found on many Crossbow sensor boards.

The SLEEP statement allows the programmer to halt operation for a desired duration. SLEEP directly maps to the thread sleep function provided by Mantis OS, which puts the mote into a low-power state when no thread is scheduled to run. The SLEEP statement is followed by an expression indicating how long the mote is to sleep, given in milliseconds. As we will discuss later, it is critical that the novice programmer understand the SLEEP statement.

To facilitate debugging, simple syntax checking is provided through error messages sent to the serial port. The user can also debug using the PRINT statement, which outputs the given expression list to the serial port.

We expose one-hop communication in the form of the SEND and RECEIVE statements. SEND mimics the functionality of PRINT, but communicates the data using the radio on the mote. The user can construct a message string to be broadcast using a combination of static text and integer expressions. The RECEIVE statement listens for messages sent with SEND. It mimics the BASIC INPUT statement. In our user study, we do not expose the RECEIVE function to the user but instead use it to implement base-station functionality.

In our future work, we hope to evaluate how well naive users respond to a range of communication options, each providing different message abstractions, routing algorithms and quality of service guarantees. A key challenge is how to hide (or expose) unreliable message delivery to the novice. However, this is beyond the scope of this paper.

In response to the user studies, we made several changes to our BASIC implementation, described in 6.2. These changes would not have been made without observation of the study group and the code they produced.

5. USAGE

We now describe how a programmer interacts with our BASIC implementation.

We have created a simple integrated development environment (IDE) for programming the sensor. When programming, the sensor is attached to the development machine via the serial port. When running, the sensor is detached, and the IDE serves to print messages received from sensors. Our interface has three main components. The first is a field labeled "BASIC Code" that is used for entering and editing user programs. The second field, "Mote Output" displays any messages produced by the PRINT command as well as any syntax errors generated by the program. This information comes from the serial port-connected Mote. The final field, "Base Station Output" displays any messages broadcast using the SEND message. These are received using a Mote acting as a promiscuous listener. The programmer may run and stop their application by selecting a menu item within the environment.

We deliberately designed our IDE to allow novices to program the motes without complication or external assistance. In studying the efficacy of BASIC, we did not want the development environment to be a distraction or source of failure. Furthermore, we wanted our user study to be runnable without proctor intervention. Our intention with the study is to focus on the difficulty of the programming language and thus our interface does not provide additional support in the form of code completion or line-by-line debugging. Our user interface did not pose any known usability challenges during our experimentation.

6. EVALUATION

The goals of our evaluation are

- to assess the ease at which novice and intermediate users can develop correct and power efficient simple sensor applications in both BASIC and TinyScript, and
- to determine the power and computational overhead of BASIC.

We addressed the first goal by conducting a user study on a population of novice and intermediate users, attempting to mirror the "worst case" of application domain experts. Half of users with absolutely no previous programming experience were able to complete simple sensor network tasks in BASIC, while that same category of user had much less success with TinyScript. Further, we found that novice and intermediate programmers both struggle with the event-driven model provided in TinyScript. To address the second goal, we directly measured power consumption as a function of desired compute rate, comparing a BASIC implementation with a C implementation. We found that interpreted BASIC is, of course, considerably slower than C, and thus has a far more limited maximum compute rate. Further, the power consumption costs grow much faster with increasing compute rate than C. However, for low rates, the two are quite comparable. For example, for a "sense and send" application running at a rate of 1 Hz, BASIC has only a 1.5% power overhead compared to C. Finally, we show that a compiled version of BASIC has a power profile similar to compiled C.

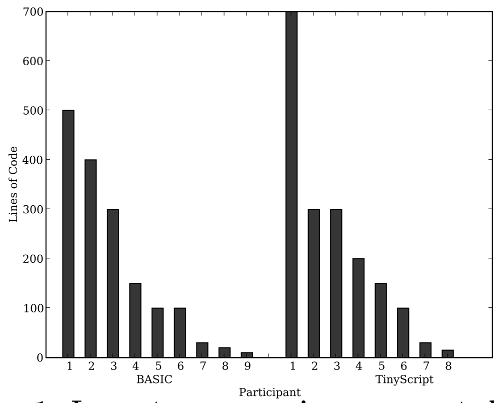


Figure 1: Largest program sizes as reported by intermediate users in our two study groups. An additional 23 users with no previous programming experience are not shown.

6.1 User study

To study the ease with which novices can use both BASIC and TinyScript to write simple sensor network applications, we conducted a user study with 40 participants.[3] We recruited participants from a population of current and recent graduate and undergraduate students at a university, specifically targeting persons with little to no programming experience. Our population includes a mix of students with concentration in both the sciences and liberal arts, and includes roughly equal numbers of participants we consider *novice users* and *intermediate users*. Novice users have no programming experience, while intermediate users have minimal programming experience. Users were randomly assigned to BASIC or TinyScript. Figure 1 illustrates the number of lines of code previously written by our intermediate users, and shows that that similar participants were assigned to each language.

For the BASIC study, we had 11 novices and 9 intermediate programmers while for the TinyScript study we had 12 novice and 8 intermediate programmers. The results we report here are summarized according to these experience levels, so the slight difference in the composition of the population sizes between the two languages is irrelevant. Among intermediate programmers in both languages groups, the mostly commonly reported languages with which the participant had some previous experience were "C/C++" (9 participants), "Java/C#" (6 participants) and "Matlab" (6 participants).

[3]The study's human subjects research protocol was approved by our Institutional Review Board, which permitted us to recruit participants from a very large and diverse pool, and to pay ($15) for their time.

6.1.1 BASIC experimental setup

Our experiments were carried out using two Crossbow MicaZ motes connected to a single PC. The PC is a Dell desktop with a 2.0 GHz processor and 1.5 GB of RAM running Windows XP. During the study, our software is the only application visible to the user. Adjacent to the setup is a desk lamp the user needs to complete the study. Only one of the motes could be directly programmed by the participant while the other acted as a base-station for receiving data sent from the participant's program. The BASIC interpreter is directly programmed via a serial connection that is attached to the mote throughout the experiment.

At the beginning of the study, each participant is presented with a tutorial document[4] explaining the BASIC programming language and the sensor hardware. The tutorial is broadly written to cover the entire (Tiny)BASIC grammar, including such topics as variables and control flow. Additionally, the sensor network extensions supporting communication and reading sensor data are described.

Unless care is taken, the results of a study such as this may be influenced by the quality of the tutorial. Before we collected the results presented here, we conducted a preliminary study evaluating the clarity of the tutorial and iterating its construction after each study. After evaluating the tutorial using three participants, we found that the tutorial sufficiently clear for evaluation.

The participant is given 30 minutes to familiarize themselves with the language and programming environment. During this time, they can use the IDE and mote to test out program examples from the tutorial or otherwise. The participant is not permitted to ask questions about the tutorial. Once the tutorial is completed, we ask the participant to fill out a questionnaire asking how difficult the tutorial was to follow, how well the user understand BASIC, the extent to which they followed tutorial examples and whether or not they felt the tutorial should be longer. The next section discusses the questionnaire results.

After this questionnaire is filled out, three exercises are given. We give the participant 20 minutes to complete each exercise. The exercises and tutorial are designed such that no example code in the tutorial can be easily transformed into an exercise solution. In this way, the set of exercises is non-trivial and forces the participant to apply the language primitives learned in the tutorial to succeed. Each exercise requires the use of at least three language features such as sensing, delay, flow control and communication.

The exercises are as follows. Efficient solutions are illustrated in Figure 2.

1 The user is asked to write a simple program that blinks one of the LEDs on the sensor hardware at the rate of 1 Hz. This tests the user's understanding of basic node programming and the use of actuation. A solution to this exercise is possible with 4 to 5 lines of code.

2 The second exercise asks the user to write an application that sends a message to the base-station when a desk lamp next to the user is turned off. This exercise requires that the user understand BASIC control flow, base-station communication and reading data from the sensors. The exercise instructs the user to write power-efficient code and that a responsiveness of 1-2 seconds is adequate.

3 The final exercise transforms the second into an actuation task in which an LED on the sensor is controlled by measuring the ambient light. In the exercise the user is instructed to illuminate an LED on the mote if the desk lamp were turned off. Other than this change, the exercises are identical.

After the participant completes the exercise or time expires, we ask them to fill out a questionnaire describing the experience, soliciting how well they understood the problem presented, the quality of their solution and the extent to which they felt frustrated throughout the exercise.

Throughout the tutorial and exercises, we periodically take a snapshot of the user's program to provide insight into the process of forming a solution and to identify any stumbling blocks. The final program is also saved so that we can independently validate each solution and test its quality.

6.1.2 TinyScript experimental setup

We evaluate the TinyScript version distributed with TinyOS version 1.1.15. Our experimental setup for evaluating TinyScript is nearly identical to that used in the BASIC study. The changes result from differences between the two development environments.

The TinyScript evaluation used the same PC as that used in the BASIC study, running a version of Ubuntu Linux in a VMWare hosted virtual machine.[5] While it is reasonable to suspect our participants are more familiar with Windows XP, no part of our study had the participants interacting directly with operating system-level user interface and thus we feel that the change does not influence our results.

We designed our TinyScript tutorial using existing tutorial documents created by TinyScript's authors. We edited these tutorials into one cohesive document, elaborating on certain concepts to make them more suitable for extremely naive programmers. Further, we changed the tone and structure of the tutorial to match that of the BASIC tutorial, making exceptions to increase the clarity of the language. As with BASIC, we iterated on the TinyScript tutorial using participants recruited from the same population used during the final study as to maximize the tutorial's clarity. In all, five participants were used. We also made a copy of the tutorial available to the original author of the TinyScript materials.

The remainder of the study, including the time allocated to the tutorial and exercises and exercise ordering, was unchanged for the TinyScript group.

6.1.3 Results

Tutorial questions. Following the tutorial, we asked participants in each language study group to rate their experience with the tutorial across four questions. We used these questions to get a sense for the participants response to the language independent of their performance on the exercises. The four questions were as follows: “I felt the tutorial was easy to understand”, “I feel that I understand [the language]”, “I followed the tutorial and tried many of the examples”, “I feel that the tutorial should be longer”. The rating scale here, and in all figures that includes ratings is on

[4]The tutorial for each language, survey instruments, and other materials are available at http://www.eecs.northwestern.edu/∼pdinda/BASICSTUDY.

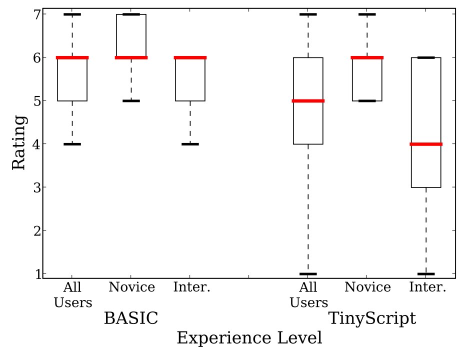
[5]Existence of the VM is invisible to the user.

```
10 led 1 0
20 sleep 1000
30 led 1 1
40 sleep 1000
50 goto 10
```
Exercise 1

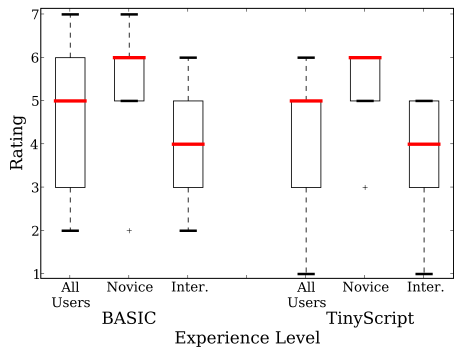
```
10 sense 0 a
20 if a < 800 then send "light is off"
30 sleep 1000
40 goto 10
```
Exercise 2

```
10 sense 0 a
20 if a < 800 then led 1 1
30 if a > 800 then led 1 0
40 sleep 1000
50 goto 10
```
Exercise 3

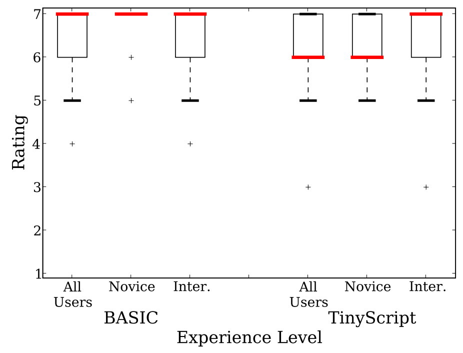
Figure 2: Example efficient solutions for the exercises.



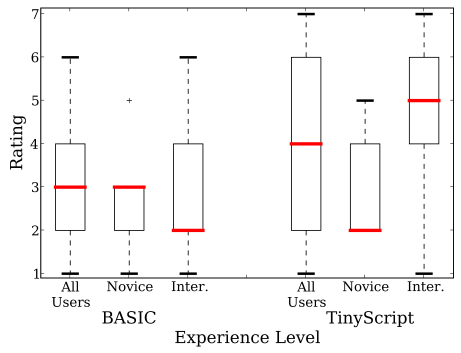
(a) “I feel that I understand [the language].”



(b) “I followed the tutorial and tried many of the examples.”



(c) “I followed the tutorial and tried many of the examples.”



(d) “I feel that the tutorial should be longer.”

Figure 3: User responses to prompts given after completion of tutorial.

a standard Leikert scale ranging from 1 to 7, where 1 corresponds to “strongly disagree” and 7 corresponds to “strongly agree” respectively. The results for each of the prompts are given in Figure 3. They are broken down by language, and by novice and intermediate users. The graphs are standard Box plots showing the distribution of responses (25, 50, and 75 percentiles), with outliers shown individually.

Overall, we find similar trends across both languages. Participants show slightly less confidence with TinyScript as indicated by their response to “I felt the tutorial was easy to understand”, with inexperienced TinyScript programmers giving the lowest ratings. Likewise, the majority of the inexperienced TinyScript programmers indicated that the tutorial was not long enough. We attribute the differences in responses between the two languages to the difference in complexity between BASIC and TinyScript, with TinyScript asking its programmers to understand concepts such as event-driven flow control and data types. Responses to the prompt “I feel that I understand [the language]” are similar for each experience level across both languages. We believe this result is due to the fact that the participants have not yet experienced coding applications in either language.

User experience of tasks. After working on each exercise, we asked participants to rate their experience. For each exercise, we asked the following questions, using the same scale used in the tutorial questions: “I understood what the exercise was asking me to do”, “I was able to complete the exercise in the provided time”, “I felt frustrated through the exercise”. In Figure 4 we present these results. For the first question, we see similar results between the two languages, indicating that the communication of our exercises does not seem to have introduced additional variation between the languages. However, in the remaining questions we begin to see more differentiation between the BASIC and TinyScript programmers.

Measured performance on tasks. In Figures 5(a)–(d), we present all data about the measured performance of users in our study, as well as summaries by language and expertise. As Figure 5(a) shows, all intermediate BASIC programmers were able to complete at least one of the exercises.

In examining the BASIC solutions provided by participants, we discovered a common error in exercise 2 in which participants reversed the usage of the PRINT and SEND

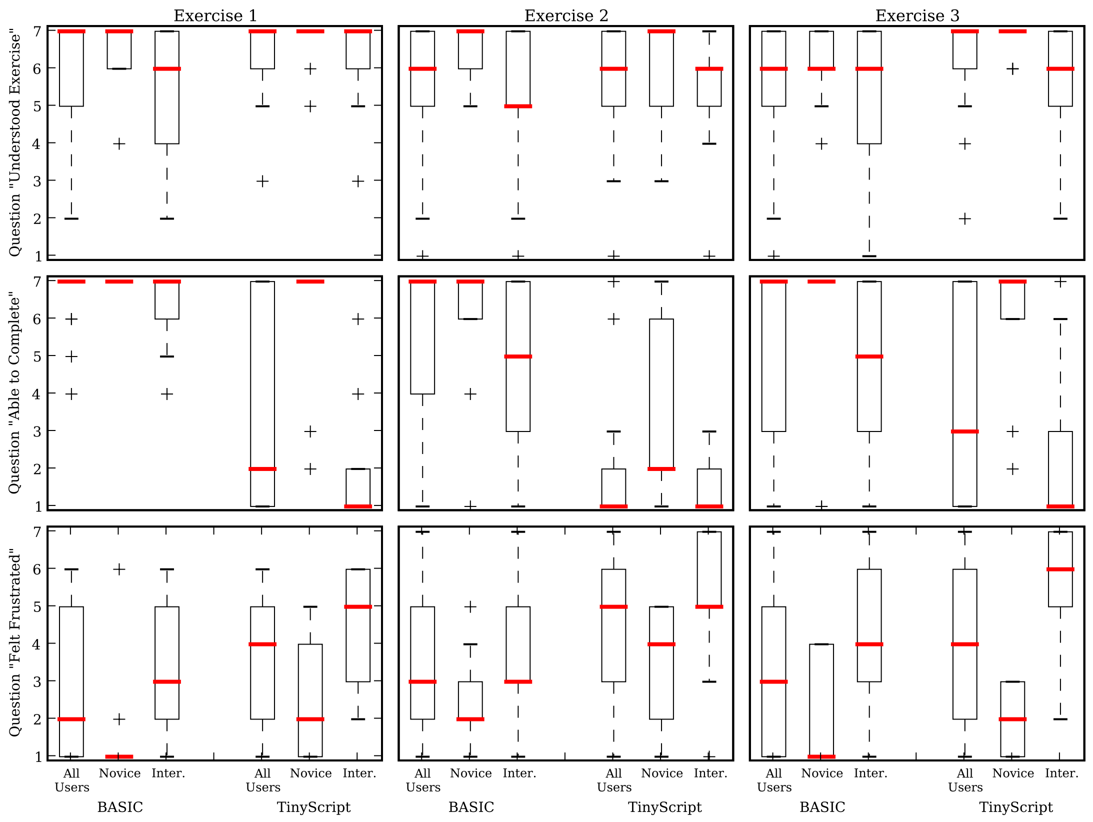


Figure 4: User responses to a set of prompts given after each exercise. Columns correspond to different exercises while rows indicate the question being answered.

statements. Recall that the semantics of the two statements are nearly identical, which each taking a list of expressions that is then communicated to either a "Base Station Output" panel in BASIC IDE in the case of the SEND statement and a nearly identical output window labeled "Mote Output" contains data communicated via the PRINT statement. We attribute the confusion between the two statements to the design of our user interface and the lack of distinction between the two communication modalities. We note solutions that use the PRINT statement instead of SEND but are otherwise correct in the column titled "Correct (PRINT)." As will be discussed, many participants using TinyScript experienced an identical confusion while attempting to complete the second exercise.

Ignoring errors caused by this confusion, we find completion rates for intermediate programmers in BASIC were 100%, 89%, and 67% for exercises 1, 2, and 3, respectively. In Figure 5(b), we present the comparable results for the intermediate programmers using TinyScript. For the first and third exercises, the group performed approximately as well as their counterparts using BASIC, with correct solutions provided by 100% and 57% of the participants. However, this trend does not continue for participants attempting the second exercise, in which none of the TinyScript users were able to provide correct solutions.

Inspection of participants' solutions for exercise 2 reveals that many struggled with using the array abstraction provided by TinyScript. Use of the arrays are required for completing the exercise as the communication functions provided by TinyScript exclusively use array arguments. While this fact makes direct comparison between BASIC and TinyScript problematic for this exercise, there is nonetheless something to be learned from the manner in which users struggled. The array data structures provided by TinyScript appear to have been designed to simplify their use: the arrays are of a fixed size (10) and provide a shorthand for push and pop operations, giving the structure the feeling of an array-stack hybrid. Many participants struggled with this functionality, creating solutions that would often cause the buffer to overflow. In Section 6.2 we discuss how we use these findings to influence our implementation of arrays in BASIC.

The confusion regarding the distinction between serial and wireless communication was also visible in the TinyScript group. Analogous to the PRINT and SEND statements are uart() and send() functions. Similar numbers of participants confused their use as in the BASIC study, though no participants were able to provide a solution that was otherwise correct.

Finally, we find that both groups were able to provide efficient implementations at similar rates. Efficient implementations are those that use each language's (either implicit or explicit) abstractions for power management.

We now draw our attention to the ability of novice participants to complete the set of exercises. We report these results for BASIC and TinyScript in Figures 5(c) and (d) respectively. Here, the contrast between the two languages

	Exercise 1			Exercise 2				Exercise 3			Total	
Subject	Correct	LoC	Efficient	Correct	Correct (PRINT)	LoC	Efficient	Correct	LoC	Efficient	Correct	Efficient
1	Yes	5	Yes	Yes	*	5	Yes	No	*	*	2	2
2	Yes	5	Yes	Yes	*	3	No	Yes	4	No	3	1
3	Yes	6	Yes	No	Yes	5	Yes	Yes	6	Yes	2	3
4	Yes	5	Yes	No	Yes	4	Yes	Yes	5	Yes	2	3
5	Yes	5	Yes	No	No	*	*	No	*	*	1	1
6	Yes	5	Yes	No	Yes	5	Yes	No	*	*	1	2
7	Yes	5	Yes	No	Yes	5	Yes	Yes	4	No	2	2
8	Yes	5	Yes	Yes	*	5	Yes	Yes	5	Yes	3	3
9	Yes	5	Yes	Yes	*	10	Yes	Yes	5	Yes	3	3
	Percentage Correct		100.0%	Percentage Correct			44.4%	Percentage Correct		66.7%		
	Percentage Efficient		100.0%	Percentage Correct (PRINT)			88.9%	Percentage Efficient		66.7%		
	Avg. LoC (Std. Dev.)		5.11 (0.33)	Percentage Efficient			87.5%	Avg. LoC (Std. Dev.)		4.83 (0.75)		
				Avg. LoC (Std. Dev.)			5.25 (2.05)					

(a) Exercise results for intermediate programmers using BASIC.

	Exercise 1			Exercise 2				Exercise 3			Total	
Subject	Correct	LoC	Efficient	Correct	Correct (Uart)	LoC	Efficient	Correct	LoC	Efficient	Correct	Efficient
1	Yes	8	Yes	No	No	*	*	Yes	10	No	2	1
2	Yes	9	Yes	No	No	*	*	Yes	12	No	2	1
3	Yes	8	Yes	No	No	*	*	Yes	16	Yes	2	2
4	Yes	9	Yes	No	No	*	*	No	*	*	1	1
5	Yes	8	Yes	No	No	*	*	Yes	9	No	2	1
6	Yes	8	Yes	No	No	*	*	No	*	*	1	1
7	Yes	6	Yes	No	No	*	*	No	*	*	1	1
8	No	*	*	No	No	*	*	Yes	10	Yes	1	1
	Percentage Correct		100.0%	Percentage Correct			0.0%	Percentage Correct		71.4%		
	Percentage Efficient		100.0%	Total Correct (PRINT)			0.0%	Percentage Efficient		50.0%		
	Avg. LoC (Std. Dev.)		8.00 (1.00)	Percentage Efficient			0.0%	Avg. LoC (Std. Dev.)		11.40 (2.79)		
				Avg. LoC (Std. Dev.)			*					

(b) Exercise results for intermediate programmers using TinyScript.

	Exercise 1			Exercise 2				Exercise 3			Total	
Subject	Correct	LoC	Efficient	Correct	Correct (PRINT)	LoC	Efficient	Correct	LoC	Efficient	Correct	Efficient
1	Yes	5	Yes	No	Yes	4	Yes	Yes	5	Yes	2	3
2	No	*	*	No	Yes	4	Yes	No	*	*	0	1
3	Yes	5	Yes	No	No	*	*	No	*	*	1	1
4	Yes	9	Yes	Yes	*	5	Yes	Yes	6	Yes	3	3
5	No	*	*	No	*	*	*	Yes	12	Yes	1	1
6	No	*	*	No	*	*	*	No	*	*	0	0
7	Yes	5	Yes	No	*	*	*	No	*	*	1	1
8	No	*	*	No	*	*	*	No	*	*	0	0
9	Yes	5	Yes	Yes	*	4	Yes	Yes	3	No	3	2
10	No	*	*	No	*	*	*	No	*	*	0	0
11	Yes	5	Yes	Yes	*	3	No	Yes	6	No	3	1
	Percentage Correct		54.5%	Percentage Correct			27.3%	Percentage Correct		45.5%		
	Percentage Efficient		100.0%	Percentage Correct (PRINT)			45.5%	Percentage Efficient		60.0%		
	Avg. LoC (Std. Dev.)		5.66 (1.63)	Percentage Efficient			80.0%	Avg. LoC (Std. Dev.)		6.40 (3.36)		
				Avg. LoC (Std. Dev.)			4.00 (0.71)					

(c) Exercise results for novice programmers using BASIC.

	Exercise 1			Exercise 2				Exercise 3			Total	
Subject	Correct	LoC	Efficient	Correct	Correct (Uart)	LoC	Efficient	Correct	LoC	Efficient	Correct	Efficient
1	No	*	*	No	No	*	*	No	*	*	0	0
2	No	*	*	No	No	*	*	No	*	*	0	0
3	No	*	*	No	No	*	*	Yes	15	No	1	0
4	No	*	*	No	No	*	*	No	*	*	0	0
5	No	*	*	No	No	*	*	No	*	*	0	0
6	No	*	*	No	No	*	*	No	*	*	0	0
7	No	*	*	No	No	*	*	No	*	*	0	0
8	No	*	*	No	No	*	*	No	*	*	0	0
9	No	*	*	No	No	*	*	Yes	8	Yes	1	1
10	No	*	*	No	No	*	*	No	*	*	0	0
11	No	*	*	No	No	*	*	No	*	*	0	0
12	No	*	*	No	No	*	*	No	*	*	0	0
	Percentage Correct		0.0%	Percentage Correct			0.0%	Percentage Correct		16.7%		
	Percentage Efficient		0.0%	Percentage Correct (PRINT)			0.0%	Percentage Efficient		50.0%		
	Avg. LoC (Std. Dev.)		*	Percentage Efficient			0.0%	Avg. LoC (Std. Dev.)		11.50 (4.95)		
				Avg. LoC (Std. Dev.)			*					

(d) Exercise results for novice programmers using TinyScript.

Figure 5: Exercise results for BASIC and TinyScript.

```
10 sense 0 a
20 if a < 800 then led 1 1
30 sleep 2000
40 if a > 800 then led 1 0
50 sleep 2000
60 goto 10
```

Figure 6: One participant's solution to exercise 3. The extra SLEEP statement indicates a misunderstanding of proper duty cycling.

```
10 sleep period 1000
20 sense 0 a
30 if a < 800 then send "light is off"
40 resume
```

Figure 7: An example solution to exercise 2 using the periodic sleep language extension made in response to feedback from the user study.

becomes much more pronounced. For the first exercise, 54% of participants using BASIC were able to complete the first exercise, whereas none of the participants using TinyScript could do so. Likewise, 45% of the BASIC participants completed exercise 2, while none of the TinyScript participants had success. Finally, 17% of the participants using TinyScript had success with exercise 3, compared to 46% of the BASIC group. Overall, on the order of *half of the novices—people who have never programmed before in any language—were able to produce good solutions in BASIC*, a considerably larger fraction than with TinyScript.

We attribute the different rates of success between the two groups to the relative complexity of the TinyScript language. Each additional language feature (e.g., data types, event-driven control flow, etc.) imposes a real cost on the novice developer. Given the typical programming background of our intermediate participants, it is likely that they have previously encountered topics such as types and variable scope.

Outside of the coarse categorization of "correct" versus "incorrect" solutions, we observed another interested phenomenon: few solutions submitted by the participants used TinyScript's event model and instead relied on an imperative approach. Our TinyScript manual gives equal weight to both approaches, and even provides an example of a periodic TinyScript program[6]. Only 3 out of the 15 total correct solutions across all of the TinyScript exercises used the event-driven model.

Overall, our studies show that our extended BASIC allows for novices who have never programmed before to complete simple sensor network applications after a short tutorial. We have also found that both novice and intermediate programmers struggle with the event-driven model provided by TinyScript.

6.2 Language modifications

In examining the code produced by our study participants, we found participants in both the BASIC and TinyScript groups that either neglected to duty-cycle their applications or did so in such a way that introduced unnecessary delay to event detection. An example of this can be seen in Figure 6.

In response to this issue, we designed and implemented a simple extension to our BASIC interpreter to facilitate the kind of duty-cycled applications common in WSNs. The language extension adds the PERIOD keyword which can be used as a modifier to the SLEEP statement along with RESUME, a new statement that takes no arguments. An example of how this statement can be applied to exercise 2 can be found in Figure 7. Should the execution time of the code between the SLEEP and RESUME statement extend beyond the given period, the interpreter issues a warning.

We also added array data structures to the uBASIC language. Arrays variables are declared using the DIM keyword followed by an array size enclosed in angle brackets. Our decision to fix array size at allocation time and limit array accesses to by-index comes from watching users struggle with array indexing using TinyScript's fixed-size buffers.

The study also prompted several smaller changes. In our original implementation, arguments to SENSE and LED functions specifying which sensor or LED to access were given as integer expressions to maximize flexibility. These arguments appeared to confuse several users of our language during our evaluation so these arguments were changed to keywords indicating the color of the LED actuated or the name of the sensor. We believe that more semantic approach will further ease without a major loss of flexibility.

To support a more diverse set of applications, we added built-in function support to the BASIC interpreter, allowing for collections of domain-specific functions to be added to the interpreter at compile time. The functions are written in embedded C, eliminating a potential energy and performance bottleneck for complex operations. This extensibility is intended for expert users who can build libraries that can be easily used by novices.

Along with these changes, we are considering unifying the PRINT and SEND statements into a single communication abstraction, with the interpreter handling the choice of medium. Given our experience, such an abstraction would prove less confusing for naive programmers. We plan on exploring this idea as we focus on salient communication abstractions novice WSN developers in future versions of our BASIC implementation.

6.3 Power consumption

To understand the viability of using BASIC for sensor network applications we measured the energy required to execute typical sensor network tasks and provide an estimate of the typical overhead we would expect for programs written in BASIC. We also measured the overhead of purely computational code in order to estimate the overhead in the worst-case. For interpreted BASIC, we find that while computationally intensive code sees a considerable drop in efficiency compared to C, a typical sensor network task involving data acquisition and communication suffers only a 1.5% increase in power consumption. Furthermore, compiled BASIC shows performance comparable to code written directly in embedded C.

6.3.1 Experimental setup

In all our experiments, we compare an application written in BASIC to one written in embedded C using Mantis OS for reading sensor values and performing communica-

[6]This lead one participant to write "The solution is in the manual!" in a free response question after exercise 1. The example provided in the manual requires non-trivial modification to act as a solution to exercise 1.

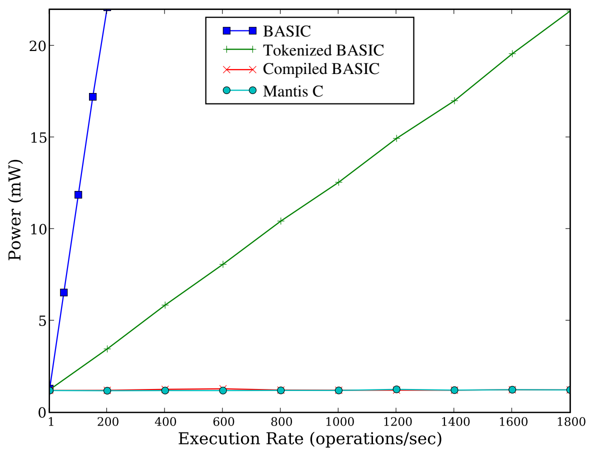


Figure 8: Power consumption as a function of execution rate.

tion. We compare the same BASIC code across our baseline interpreter as well as a modified interpreter that maintains a tokenized representation of the BASIC program. To evaluate compiled BASIC, as a proof-of-concept, we use an existing BASIC to C translator [34] to generate baseline C code to which we then add the appropriate calls to Mantis. While we feel that the rapid development environment provided by an interpreted language is one factor that aids novice programmers, we imagine that an application developer could deploy a compiled version of his or her code after it has been developed, debugged, and tested. In each power comparison, the code is functionally identical, with both the interpreter and both sets of embedded C code relying on the same Mantis OS system calls.

To measure the worst-case overhead of the BASIC interpreter, we created a simple benchmark application that executes a loop summing all integer values within a range, sleeps for a fixed amount of time, and repeats. We use this application to estimate the power requirements of several different rates of execution between the two implementations. For the purposes of our comparison, we equate one iteration of the loop with one operation.

We also evaluate a typical sensor network use case by comparing the power requirements of a "sense and send" application. The goal of this application is to sense a value from the environment at the rate of 1Hz and conditionally send a message to a base station if that value passes a given threshold. For this experiment, we set the threshold low enough such that the value is always transmitted.

We perform our measurements using a Crossbow MicaZ mote with an MTS300 sensor board. Power measurements are taken using a National Instruments 6036E data acquisition card connected to a PC running Windows XP. We measure the voltage across a 10Ω resistor in series with the power supply as a proxy for the current.

6.3.2 Results

Figure 8 illustrates the computational and power overhead as we sweep the desired execution rate of our benchmark. For an iteration rate of one operation (iteration), the interpreter and native code solution result in a difference of 0.1 mW. The non-optimized, non-tokenizing BASIC interpreter reaches a saturation point at approximately 200 operations per second at which point the interpreter cannot execute any faster. The tokenizing implementation experiences its saturation point at approximately 1800 operations per second. At this execution rate, the interpreter uses about 18x more power than the native code solution. Note that the compiled version of BASIC allows for an execution rate roughly identical to that given by embedded C. This is because the limited semantics of our BASIC grammar allow for a straight-forward translation into embedded C.

For the "sense and send" application, we measure a difference in power consumption of only 0.03 mW, with the average power consumption of the BASIC application at 2.08 mW compared to the native C application at 2.05 mW. The interpreted solution results in a 1.5% increase over C in power consumption for a simple application. The tokenized version experiences a similar overhead of approximately 1.5%, while the compiled BASIC version has negligible overhead.

Our results indicate that a purely interpreted language like BASIC is acceptable for sensor network applications that are not compute-intensive, which is many of them. Tokenization increases the range of applications for which it is appropriate. Finally, compiled BASIC has computation performance and power characteristics that are virtually identical to native C.

7. CONCLUSIONS

We have addressed the problem of making sensor networks easier to program by non-experts by exploring the use of an extended BASIC programming language in this domain. Our contributions include: (1) a BASIC implementation for modern sensor networks, (2) the first-ever user study evaluating how well novice (no programming experience) and intermediate (some programming experience) users can accomplish simple sensor network tasks in our BASIC and in TinyScript (an alternative also designed for inexperienced programmers), and (3) an evaluation of power-consumption issues in interpreted languages like BASIC. Half of users with no previous programming experience of any kind were able to program simple network tasks using our BASIC. Our experimental results show that use of a BASIC interpreter has little impact on the power consumption of applications in which computational demands are low, while compiled BASIC behaves nearly identically to compiled C. We strongly encourage further evaluation of prospective languages for sensor networks via carefully designed user studies.

8. REFERENCES

[1] Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., and Han, R. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proc. Int. Wkshp. Wireless Sensor Networks and Applications* (Sept. 2003), pp. 50–59.

[2] Automated crack measurement project. `http://www.iti.northwestern.edu/acm`.

[3] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10*, 4 (2005), 563–579.

[4] Bonivento, A., Carloni, L. P., and Sangiovanni-Vincentelli, A. Platform based design for wireless sensor networks. *Mobile Networks and Applications* (May 2006).

[5] Cerpa, A., and Estrin, D. ASCENT: Adaptive Self-Configuring sEnsor Networks Topologies. *IEEE Trans. Mobile Computing 3*, 3 (July 2004).

[6] Cheong, E., Lee, E. A., and Zhao, Y. Viptos: A graphical development and simulation environment for TinyOS-based wireless sensor networks. Tech. rep., EECS Department, University of California, Berkeley, Feb. 2006.

[7] Dartmouth College Computation Center. *A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*, 1964.

[8] Davis, D. M. The perpetual novice: An undervalued resource in the age of experts. *Mind, Culture, and Activity 4*, 1 (January 1997), 42–52.

[9] Decker, A. *How Students Measure Up: An Assessment Instrument for Introductory Computer Science*. PhD thesis, Department of Computer Science and Engineering, SUNY, 2007.

[10] Dowding, C. H., and McKenna, L. M. Crack response to long-term and environmental and blast vibration effects. *Journal of Geotechnical and Geoenvironmental Engineering 131*, 9 (September 2005), 1151–1161.

[11] Dowding, C. H., Ozer, H., and Kotowsky, M. Wireless crack measurment for control of construction vibrations. In *Proceedings of the Atlanta GeoCongress* (2006), Engineering in the Information Technology Age, Geo-Institute of the American Society of Civil Engineers.

[12] Dunkels, A. ubasic. http://www.sics.se/∼adam/ubasic/.

[13] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors* (November 2004).

[14] Fenton, N., and Pfleeger, S. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, 1998.

[15] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation Conf.* (June 2003).

[16] Gummadi, R., Gnawali, O., and Govindan, R. Macro-programming wireless sensor networks using Kairos. In *Proc. Int. Conf. Distributed Computing in Sensor Systems* (July 2005).

[17] Hartung, C., Han, R., Seielstad, C., and Holbrook, S. Firewxnet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services* (2006), ACM, pp. 28–41.

[18] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems* (2000), pp. 93–104.

[19] Horey, J., Nelson, E., and Maccabe, A. B. Tables: A table-based language environment for sensor networks. Tech. Rep. TR-CS-2007-19, The University of New Mexico, 2007.

[20] Jevtic, S., Kotowsky, M., Dick, R. P., Dinda, P. A., and Dowding, C. Lucid dreaming: Reliable analog event detection for energy-constrained applications. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)* (April 2007).

[21] Kan, S. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2002.

[22] Kemeny, J., and Kurtz, T. *Back to BASIC: The History, Corruption, and Future of the Language*. Addison Wesley, 1985.

[23] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN* (2007), ACM, pp. 254–263.

[24] Kuorilehto, M., Kohvakka, M., Hännikäinen, M., and Hämäläinen, T. D. High abstraction level design and implementation framework for wireless sensor networks. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer, July 2005, pp. 384–393.

[25] Levis, P., and Culler, D. Mate: A tiny virtual machine for sensor networks. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems* (Oct. 2002).

[26] Levis, P., Gay, D., and Culler, D. Bridging the gap: Programming sensor networks with application specific virtual machines. Tech. Rep. CSD-04-1343, UC Berkeley, August 2004.

[27] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*. Springer, 2005.

[28] Liu, T., Sadler, C. M., Zhang, P., and Martonosi, M. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2004), ACM, pp. 256–269.

[29] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[30] Microsoft. .net micro framework. http://msdn.microsoft.com/en-us/embedded/bb267253.aspx.

[31] Microsystems, S. Sunspotsworld - home of project sun spot. http://www.sunspotworld.com/.

[32] Newton, R., Morrisett, G., and Welsh, M. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM, pp. 489–498.

[33] Newton, R., and Welsh, M. Region streams: Functional macroprogramming for sensor networks. In *Proc. Int. Wkshp. Data Management for Sensor Networks* (Aug. 2004).

[34] Ohura, M. b2c: Basic to c translator. http://www.netfort.gr.jp/ ohura/b2c/README.html.

[35] Pittman, T. TINY BASIC user manual. Software Manual from Itty Bitty Computers, 1976. Available from http://www.ittybittycomputers.com/IttyBitty/TinyBasic/.

[36] Polastre, J., Szewczyk, R., and Culler, D. Telos: enabling ultra-low power wireless research. In *Proc. Int. Symp. Information Processing in Sensor Networks* (Apr. 2005).

[37] Polastre, J., Szewczyk, R., Mainwaring, A., Culler, D., and Anderson, J. Analysis of wireless sensor networks for habitat monitoring. *Wireless sensor networks* (2004), 399–423.

[38] Werner-Allen, G., Johnson, J., Ruiz, M., Lees, J., and Welsh, M. Monitoring volcanic eruptions with a wireless sensor network. In *Proc. Workshop on Wireless Sensor Networks, 2005.* (2005).