A Low-Overhead Script Language for Tiny Networked Embedded Systems

Adam Dunkels
adam@sics.se
Swedish Institute of Computer Science

September 2006

SICS Technical Report T2006:15
ISSN 1100-3154
ISRN:SICS-T–2006/15-SE

Abstract

With sensor networks starting to get mainstream acceptance, programmability is of increasing importance. Customers and field engineers will need to reprogram existing deployments and software developers will need to test and debug software in network testbeds. Script languages, which are a popular mechanism for reprogramming in general-purpose computing, have not been considered for wireless sensor networks because of the perceived overhead of interpreting a script language on tiny sensor nodes. In this paper we show that a structured script language is both feasible and efficient for programming tiny sensor nodes. We present a structured script language, SCript, and develop an interpreter for the language. To reduce program distribution energy the SCript interpreter stores a tokenized representation of the scripts which is distributed through the wireless network. The ROM and RAM footprint of the interpreter is similar to that of existing virtual machines for sensor networks. We show that the interpretation overhead of our language is on par with that of existing virtual machines. Thus script languages, previously considered as too expensive for tiny sensor nodes, are a viable alternative to virtual machines.

1 Introduction

As wireless sensor networks are beginning to see mainstream adoption, programmability issues are of increasing importance. Software developers will need to develop and test new applications both in deployed networks and in testbeds. Field engineers will need to debug, update, and maintain existing systems. Reprogramming an already deployed sensor network is likely to be more cost effective than collecting the old sensor network and deploying a new one. Because of this many different approaches to reprogramming sensor networks have been investigated; binary upgrades of native code [3, 4, 7, 12, 18, 21] and virtual machines [1, 3, 12, 14, 15] have been the most popular. Script languages, which are very popular for general-purpose computing [19], have been largely unexplored for wireless sensor networks because of the perceived high execution time overhead. Unlike virtual machine code, script languages can be directly interpreted by the sensor nodes without the need for a compiler. By having a script interpreter on each sensor node it is possible to interactively send commands and programs by connecting to the sensor node either through a serial cable or over the wireless network.

Existing script languages systems for embedded systems are either too large to fit our target platforms or use languages that many programmers are unfamiliar with. Lua [10] is an example of the former: the code size of the interpreter is 63 kilobytes which is too large for our target platforms. The Forth language is an example of the latter. Forth is designed for very small machines and while a Forth interpreter can be made very small, the Forth language is unknown to most programmers.

In this paper we investigate the use of a structured C-like scripting languages for reprogramming sensor networks. Our chief contribution is that we show that using script languages for reprogramming wireless

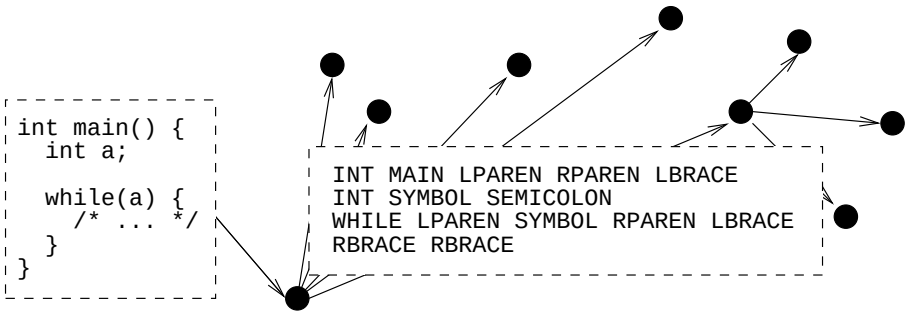

Figure 1: Network dissemination of a SCript script. The script is sent in textual form to a node in the sensor network. The node tokenizes the script and injects the tokenized version into the network. The tokenized version is smaller than the script in textual form.

sensor networks is both feasible and efficient, even for tiny sensor nodes. We present a C-like structured script language, SCript, which supports many of the features found in general-purpose script languages, including if statements, while loops, and variable scoping. We implement an interpreter for the language with a code footprint which is small enough for the interpreter be run even on tiny sensor nodes. Our system runs on the MSP430 microcontroller found in many popular sensor node platforms.

Network distribution of SCript scripts are done in tokenized form as shown in Figure 1, which reduces the distribution energy cost compared to distributing the original version of the script. We show that the distribution energy cost for tokenized SCript scripts is similar to that of a stack-based virtual machine. We measure the execution time overhead of our SCript interpreter and find it to be similar to virtual machine approaches.

Our target hardware platforms are the Tmote Sky [20] and ESB [22] boards. Both boards are equipped with MSP430 microcontrollers but with slightly different memory configurations. The microcontroller of the Tmote board has 48 kilobytes of on-chip flash ROM and 10 kilobytes of RAM whereas the ESB has 60 kilobytes of flash RAM and only 2 kilobytes of RAM. These memory limitations are typical for a wide range of embedded systems.

The architecture of our SCript interpreter is shown in Figure 2. The tokenizer module converts SCript scripts into a stream of tokens that are executed by the interpreter module. The stream of tokens can be stored for later distribution over the network. The tokenizing process compresses SCript keywords into single-byte tokens and variable names into single-byte identifiers. Hence the script in tokenized form is smaller than the original script and network distribution of the tokenized script is less costly in terms of energy than distribution of the original script. The tokenized script includes shortened versions of the variables names used in the textual version of the script. Comments in the textual script are removed before the tokenized script is stored. The SCript interpreter can be seen as a virtual machine where the tokenized version of the SCript script is the byte code for the virtual machine. Unlike other virtual machine approaches, however, code for the SCript interpreter does not need to be compiled before inserted into the sensor network.

The rest of this paper is structured as follows. In Section 2 we present related work and review methods for reprogramming sensor networks. We present the SCript language in Section 3 and its implementation as an interpreter in Section 4. We evaluate SCript and its implementation in Section 5 and conclude the paper in Section 6.

2 Related Work

2.1 Distribution Protocols

There are several types of protocols for distributing code updates in sensor networks. Trickle [17] represents what perhaps is the most basic type of software distribution protocol, in which a program contained in

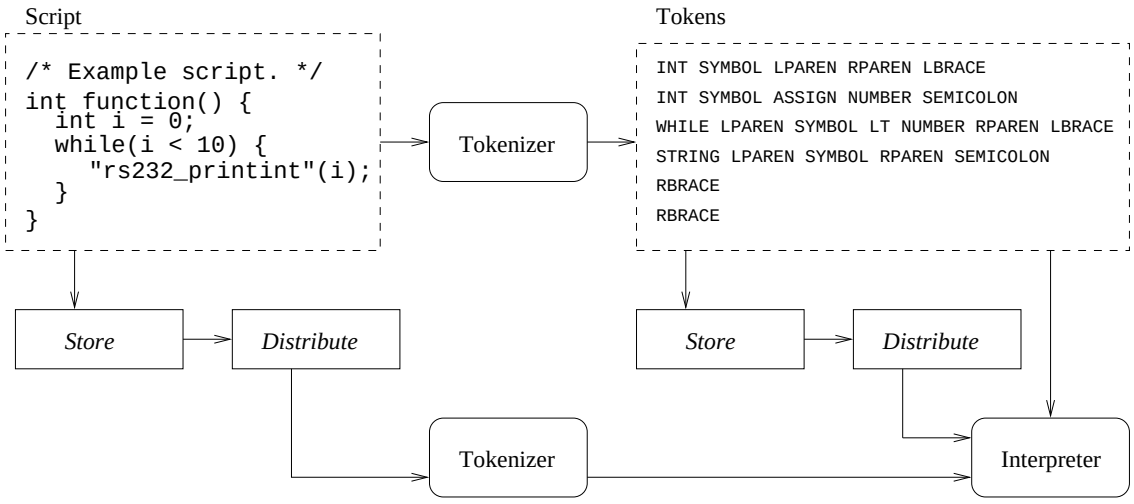

Figure 2: All sensor nodes contain the entire SCript system. The tokenizer turns the script into single-byte tokens that are interpreter by the interpreter. The tokenized script is smaller than the script in textual form because all keywords and variable names have been replaced with single-byte tokens. While it is possible to store and distribute SCript scripts in script form, scripts are distributed in tokenized form across wireless sensor networks.

a single data packet is distributed throughout the entire network. Each program is given a version number and when a node receives a program with a higher version number than what it currently has the node replaces the old version of the program with the new version. Trickle uses a mechanism called polite gossiping to avoid overloading the network with packets.

Deluge [9] adds support for multi-packet programs to Trickle. Deluge is used in TinyOS [8] to distribute an entire operating system image to a network of nodes. When a node has received an entire system image, the node replaces the current system image with the new one and reboots the system. Deluge has an average overhead in terms of number of network packets of 3.35 [9].

2.2 Native Code

Methods for updating the native code running on the sensor nodes can be divided into two groups: those that require support from an underlying operating system and those that work regardless of the operating system, if any, running on the sensor nodes. Dynamic loading of native code modules typically require support from the operating system, whereas replacing or updating the entire system image can be performed without support from the operating system.

Early native code update mechanisms [9] replaced the entire system image with a new image containing the updated software. While such an approach is very flexible in that all levels of the system can be updated, the method consumes more energy than more modular approaches because of the large amounts of data that needs to be distributed throughout the network. Later developments investigate the use of binary difference and edit scripts to distribute only the differences between the new and the old system image [12, 18, 21].

Contiki [4] and SOS [7] are operating systems that, unlike TinyOS [8], support dynamic loading of native code modules. Modules in SOS are compiled to position-independent code whereas Contiki supports dynamic linking using the ELF file format [3]. Both systems provide mechanisms for interaction between loaded programs.

2.3 Virtual Machines

Virtual machines have been investigated for sensor networks as an approach to reduce the distribution energy costs for software updates. The code size of the programs running on top of the virtual machine can

be kept to a minimum since the virtual machine can be tailored to the needs of a applications for a specific domain such as sensor networks. The drawback of virtual machines is the increased execution overhead over native code.

Maté [14] was the first virtual machine specifically targeted to wireless sensor networks. Maté is a stack-based virtual machine that runs on top of TinyOS. Maté instructions are 8 bits wide. Each virtual machine instruction is executed in a separate TinyOS run-to-completion task. Levis et al. [15] have further investigated the use of application specific virtual machines (ASVMs) that are compiled for the needs of a particular application or set of applications. Other examples of stack-based virtual machines for sensor networks are DVM [1] and CVM [3]. There are also Java-based virtual machines for sensor networks: VM* system [13] and the Contiki Java VM [3].

2.4 Script Languages

Existing structured script languages are typically too large for our target platforms. The SensorWare system [2] uses a reduced version of Tcl to provide a script-based programming environment for sensor networks. However, the system is designed for sensor nodes with an order of magnitude more memory resources than our target platform; the SensorWare system occupies 180 kilobytes of memory which is many times the size of our target platform.

Rappit [6] is a development framework for scripting languages for embedded systems. Rappit uses a host environment running Python that sends commands to the embedded systems. The host environment runs on a resource-rich server system outside of the sensor network that translates commands into simpler messages that are executed by the embedded system. Tapper [23] is a command language and lightweight stack-based script engine for sensor nodes built with Rappit. The Tapper language provides primitives for accessing hardware devices such as analog to digital converters and for sending and receiving radio packets. Both Rappit and Tapper do, however, require assistance from a host system for interpreting the commands. SCript interprets and executes scripts directly on the target systems. Furthermore, unlike the terse command languages of Rappit and Tapper, the SCript language is designed to be a structured programming language.

3 The SCript Language

SCript is an imperative programming language that is designed to be similar to C so that programmers quickly can get familiar with the language. A SCript program looks very similar to a C program, as can be seen in Figure 3. SCript supports iteration statements (while loops), selection statements (if statements), function calls, and arithmetic expressions. The SCript grammar is shown in Figure 4.

SCript allows scripts restricted access to native functions, variables, and memory locations. Native functions and variables are accessed from a SCript script by using the textual name of the native function or variable enclosed in double quotes. The underlying system must provide the SCript interpreter with a symbol table with pointers to functions or variables along with their textual names so that the SCript interpreter can lookup names at run time. The symbol table may be either automatically generated at compile time or manually constructed by the system developer. SCript scripts can only call native functions whose names are in the symbol table. The symbol table can therefore be used to restrict access for SCript scripts. To restrict memory access from the SCript scripts, the SCript interpreter holds a list of allowed memory locations and does not allow access to other locations.

SCript use a single data type for representing both integers and pointers. This helps to keep the language simple enough to implement on memory-constrained embedded systems while providing a mechanism for accessing any allowed memory address in the system. Pointer arithmetic is defined in SCript so that a pointer expression can address individual bytes. A limited form of array addressing, which only allows for integer arrays, is supported.

A SCript program consists of one or more functions. SCript functions take a variable number of parameters and must be defined before they can be called. Program execution always starts at the `main` function, which must be the last function to be defined. Native functions can be called without being defined or declared.

```
int global_var = 3;

void function(int a) {
  /* Call to the native function
     print_int: */
  "print_int"(a);
}

/* Execution starts with the
   main function. */
void main() {
  int i;

  i = 0;
  while(i < global_var) {
    if(i % 2 == 0) {
      function(i);
    }
    /* Addition of the native
       variable increase. */
    i = i + *"increase";
  }
}
```

Figure 3: An example SCript script.

A SCript script is contained in a single file. On the sensor node the script file may not need to be physically stored as a file but can reside in RAM, ROM, or on an external memory. The current implementation does only handle scripts that are stored in immediately addressable ROM or RAM. SCript can also run in an interactive mode where language statements can be entered by a user that is either over a network or through direct connection to the device running SCript. The interactive mode is intended for using SCript as a debugging language. In interactive mode, functions cannot be defined and while loops are not supported.

Comments in SCript are denoted as in C/C++, and Java. Multi-line comments begin with "/*" and end with "*/". Single-line comments begin with "//" and end at the end of the line.

3.1 SCript Statements

SCript has four statements: selection statements, iteration statements, function calls, and a conditional blocked wait statement. Statements are executed in the order they are defined within a function. Multiple statements are grouped together in a compound statement. A compound statement begins with a left brace ("{") and ends with a right brace ("}"). Expressions in SCript are made up of numbers, variable names, calls to SCript functions and native functions, and arithmetical operators. Functions can take a variable number of parameter and may return a value. Variables can be declared globally or locally. Global variables are accessible from all functions in a SCript script whereas local variables only are accessible in the scope in which they are declared. Variables must be declared before they are used.

3.1.1 Selection and Iteration Statements

SCript has one selection statement and one iteration statement: the if and while statements. The if statement takes a conditional expression and one or two compound statements, called the primary and secondary compound statement. The secondary compound statements is optional and if it exists it is separated from the

```
script     := ( vardecl | function )+ .
function   := type symbol "(" ( parameters ) ")" compound .
type       := "int" | "void" .
parameter  := type symbol
parameters := parameter ( "," parameter )* .
compound   := "{" statement* "}" .
vardecl    := type varinit ( "," varinit )* ";" .
varinit    := symbol ( "=" expression )? .
statement  := compound |
              vardecl |
              "if" "(" expression ")" compound ( "else" compound )? |
              "while" "(" expression ")" compound |
              "wait" "(" expression ")" ";" |
              symbol ( "[" expression "]" )? "=" expression ";" |
              string ( "[" expression "]" )? "=" expression ";" |
              symbol "(" ( arguments )? ")" ";" |
              string "(" ( arguments )? ")" ";" .
arguments  := expression ( "," expression )* .
expression := condition ( ("<" | ">" | "==") condition )? .
condition  := term ( ( "+" | "-" | "&" | "|" | "<<" | ">>" ) term )? .
term       := factor ( ( "*" | "/" | "%" ) factor )? .
factor     := number | varfactor | "*" factor | "&" string |
              "(" expression ")" .
varfactor  := symbol ( "(" arguments ")" )? |
              string ( "(" arguments ")" )? .
```

Figure 4: The SCript grammar in EBNF format. For brevity the rules for the terminals `symbol`, `string`, and `number` are not included.

primary compound statement by the keyword “else”. Depending on the value of the conditional expression the primary or secondary compound statement is executed.

The while statement takes a conditional expression and a single compound statement. If the conditional expression evaluates to non-zero the compound statement is executed. When the compound statement has been executed the conditional expression is reevaluated and the compound statement is executed again if the value of the conditional expression is non-zero. The process is repeated until the conditional expression evaluates to zero.

3.1.2 Functions

Functions in SCript take zero or more parameters. Functions must be defined before they can be used. Functions can return integer values and function calls can be part of arithmetic expressions. Function arguments are passed by value.

The function with a name of “main” is special in SCript. The main function is where execution starts when a SCript script is run. The main function must always be defined as the last function in a script. Functions or variables defined below the main function are never executed by the SCript interpreter.

3.1.3 Conditional Blocking Wait

SCript has a conditional blocking wait statement which allows SCript programs to block while a conditional expression is zero. Program execution does not continue until the conditional expression turns non-zero. The conditional blocking wait is modeled after the conditional blocking wait mechanism from protothreads [5]. The SCript conditional blocking mechanism is different from blocking wait mechanisms

in other programming systems in that it does not require the program to do any setup before invoking the conditional blocked wait statement. In contrast, the VM* programming environment [13] requires four lines of code and two local variables to perform a conditional blocked wait operation.

3.1.4 Arithmetic Expressions

Arithmetic expressions in SCript are made up of numbers, variables, function calls, and mathematical operators. The precedence of the operators follow the standard mathematical notational precedence conventions. Parentheses can be used to change the precedence within an expression. In addition to addition, subtraction, multiplication, and division, SCript also currently supports bit-wise boolean or and and, C's bit shifting operators, the equality operator, and the less-than and greater-than relational operators. Additional operators may be added in the future.

3.1.5 Variables

Variables are either declared globally at the top of the script program or locally at the beginning of a compound statement. The global variables are visible in all functions defined by the SCript script, whereas the scope of the local variables is within the compound statement where they are declared. It is possible to declare two or more variables with the same name. If so, the last declared variable will be used until its scope is left.

3.2 Accessing Native Variables and Calling Native Functions

SCript supports getting and setting native variables, as well as calling native functions from within SCript programs. SCript programs can take the address of native variables and write data to them. However, only a single data type, the native integer data type, is currently supported. Native functions that are called from SCript programs can take up to four arguments. All arguments must be the of the native integer data type.

Native variables and functions do not need to be declared by the SCript program before they are used. Instead, SCript use double quotes around the names of native functions or variables to distinguish them from SCript functions or variables. Like C, the address of a native function or variable can be taken by using the ampersand operator. The value of a native variable is accessed by using the asterisk operator in front of the variable name. Native functions are called just like SCript functions, but with double quotes surrounding the name of the native function. The example code in Figure 3 both calls a native function (print_int()) and accesses a native variable (increase). By using pointer arithmetic, SCript scripts can also access memory locations other than those pointed to by native variables. The SCript interpreter may, however, restrict memory access for SCript scripts.

3.3 Language Extensions

In addition to the core language constructs, SCript provides a set of language extensions targeting the special needs for sensor network programming. Common operations when programming sensor nodes are accessing sensors, sending and receiving data from other nodes on the network, and managing timers. SCript provides functions for all these operations. The purpose of adding these functions as language extensions rather than by using the native function interface is to reduce the size of the tokenized SCript scripts and to increase the execution time efficiency of SCript scripts.

4 The SCript Interpreter

We have implemented the SCript interpreter on top of the Contiki operating system [4]. Contiki is an event-driven operating system for tiny sensor nodes with a small memory footprint. Contiki provides a multi-threading library which we use to implement the SCript interpreter. SCript runs as a Contiki process with the interpreter running in a separate thread. The interpreter thread is scheduled by the SCript process. After scheduling the interpreter thread the SCript process posts a continuation event to itself, thus making sure that Contiki will call the SCript process after handling other events in the system.

We implement the SCript interpreter using a standard recursive descent parser, directly implemented in C. While tools such as Yacc [11] would make both development of the tokenizer and the parser easier and the run-time efficiency better, the code produced by such tools requires too much code space and data memory to be useful in memory-constrained embedded systems. To quantify this, we compiled a reduced version of the SCript grammar with the Bison, the GNU version of Yacc. This resulted in a 15 kilobytes large object code file. Furthermore, the code produced by Bison allocates tens of kilobytes of memory with either the C malloc() or alloca() functions. This is not possible on our target system since it has only a few kilobytes of RAM.

The current implementation of the SCript interpreter reads SCript scripts from ROM or RAM. It is currently not possible to store scripts in an external location such as an external EEPROM or a serial flash ROM. However, there is nothing in the design of the interpreter that prevents scripts to be stored at other locations. Future versions of the interpreter may include this functionality.

4.1 Tokenization

The tokenizer translates the textual representation of a SCript script into a stream of single-byte tokens that are executed by the SCript interpreter. The tokenizer removes all comments from the code before the tokens are delivered to the interpreter. The tokenizer recognizes all SCript operators, strings, and symbols (variable names).

4.2 Storage for Variables

During execution, SCript variables are stored on the symbol stack. An entry on the symbol stack contains three fields: a pointer to the symbolic name of the variable, the length of the symbolic name, the type of the variable, and the value of the variable. The name of the variable is not directly stored on the stack, but only a reference to the variable name. Variables can be either integer variables or functions. The variable type is used to ensure that function calls only are able to call defined functions and not integer variables.

When the interpreter finds a variable declaration in the SCript program the interpreter allocates a new entry on the symbol stack. The name of the entry is set to point to the location in the script where the symbolic name of the variable is declared. Scoping of variables is implemented by saving the symbol stack pointer when it starts executing a compound statement. The symbol stack pointer is restored when the compound statement has been executed. This ensures that any variables declared within the compound statement are removed from the symbol stack when execution of the compound statement has completed.

The memory for the symbol stack is statically allocated when the SCript interpreter is compiled. The maximum number of variables that can be used by SCript programs is a compile time option.

4.3 Execution of SCript Statements

The interpreter thread yields after executing a statement in order to allow other Contiki programs to run. The Contiki interface module posts a continuation event to itself when the interpreter thread has yielded so that the SCript interpreter will run immediately after Contiki has processed other events in the system.

4.3.1 The Token Pointer

The tokenizer provides a function for obtaining and setting a pointer within the stread of tokens, called the token pointer. The interpreter can only set the token pointer to a value that it previously obtained from the tokenizer and cannot increase or decrease the token pointer. The interpreter uses the token pointer when executing while loops, if statements, and function calls.

4.3.2 Arithmetic Expressions

Arithmetic expressions are executed by parsing the expression with the recursive descent parser. Expressions are evaluated as they are parsed and when the entire expression has been parsed the interpreter has the

result of the expression. Expressions can contain regular numbers, SCript script variables, native variables, and calls to both SCript script functions and native C functions.

4.3.3 If Statements

If statements consist of three parts; a conditional expression, and a primary and a secondary compound statements. The primary compound statements is executed if the conditional expression evaluates to true. The secondary compound statement is optional is executed is the conditional expression evaluates to false.

If statements are executed by first evaluating the conditional expression of the if statement. If the expression evaluates to non-zero the primary compound statement, which directly follows the conditional expression, is executed. If the conditional expression evaluates to zero, the primary compound statement following it is skipped. If the conditional expression evaluates to zero the primary compound statement is skipped and the secondary is executed.

Skipping a compound statement is done by reading the tokenized data stream while counting the number of right and left braces. The counter is increased for every left brace and decreased for every right brace. If the counter reaches zero an entire compound statement has been skipped. Counting braces ensures that nested compound statements are correctly accounted for.

4.3.4 While Loops

To execute a while loop, the interpreter stores the token pointer before evaluation the while statement's conditional expression. This is done so that the interpreter can reevaluate the conditional expression. Next, the conditional expression is evaluated. If it evaluates to non-zero, the compound statement following the conditional expression is executed. When the compound statement has been executed, the saved token pointer is restored and the conditional expression is reevaluated. The compound statement is executed again if the expression evaluates to non-zero, and the process is repeated as long as the conditional expression evaluates to non-zero. If the conditional expression evaluates to zero the compound statement of the while statement is skipped by using the same procedure as in the execution of the if statement.

4.3.5 Function Calls

Function calls are executed by storing and restoring the token pointer. To execute a function call, the interpreter looks up the name of the function on the symbol stack. The symbol stack contains the names of all functions that have been defined by the script, together with the token state at the point that the function was defined. When executing a function call, the interpreter first saves the token state at the function call and sets the token state to the value found on the symbol stack. When the called function returns, the token state is restored. To support nested function calls, the interpreter saves the token state on the native C stack.

4.3.6 Conditional Blocking Waits

Conditional blocking waits consist of a single conditional expression. The script should not continue its execution if the conditional expression evaluates to zero. The naive implementation of the conditional blocking wait statement is to enter a busy-wait loop, continuously reevaluating the conditional expression until is evaluates to non-zero. This would, however, make it impossible for the operating system to turn the microcontroller into sleep mode thus wasting valuable energy.

To allow the operating system to put the microcontroller to sleep, the SCript interpreter sets a flag indicating that the script is waiting in a conditional blocked wait. This flag is inspected by the Contiki interface module which can cause its process to wait for incoming events rather than to post a continuation event to itself.

The interpreter executes conditional blocking wait statements by storing the token pointer before evaluating the conditional expression. If the conditional expression evaluates to zero the interpreter sets the waiting flag and yields the interpreter thread. Timers and incoming packets will cause events to be posted to the SCript interpreter process thus invoking the interpreter to reevaluate the conditional statement. This implementation of the SCript conditional blocking wait statement ensures that the microcontroller can be put to sleep during conditional blocked waiting statements.

4.4 Accessing Native Functions and Variables

To access native variables and functions from within the SCript script the SCript interpreter uses a list of string representations and addresses of all native variables or functions that are available in the system. Since Contiki already includes a table of all symbols in the Contiki core [3], the SCript interpreter uses this symbol table when looking up names of native variables or functions. To execute a call to a native function or an access to a native variable, the interpreter performs a Contiki symbol table lookup. Thus the execution time of the symbol table lookup determines the speed at which native variables or functions are accessed. The Contiki symbol table is implemented as a binary search which makes lookups fast.

The SCript interpreter keeps a list of memory ranges that SCript scripts are allowed to access. SCript scripts cannot access memory outside of the allowed ranges. Moreover, SCript scripts cannot call native functions other than those found in the symbol table. By providing the SCript interpreter with an alternative symbol table it is possible to restrict access to native functions as well.

4.5 Storing the Tokenized Data Stream

The tokenized data stream can be stored in memory for network distribution or for later execution. The tokens are stored as single bytes in a token array which can be either in RAM or ROM. Strings are stored verbatim in the token array. Names of SCript variables are not stored verbatim. Rather, during the process of storing the tokenized data stream each variable is assigned a numerical value that is unique within the variable's scope. This value is stored in the token array rather than the symbolic name of the variable. To be able to assign unique numbers to each variable, the storage module has knowledge of SCript scoping rules.

4.6 Required Operating System Components

The SCript interpreter makes use of two modules provided by the Contiki operating system: the symbol table and the multi-threading library. Neither of these modules are specifically tied to Contiki but can be used independently. The SCript language extensions use Contiki timers and functions for sending and receiving packets. By reimplementing the language extension functionality as stub functions, and by breaking out the thread and symbol table modules from the Contiki code based, we were able to port the SCript interpreter to run as user process under FreeBSD in a few minutes.

5 Evaluation

We use three programs to evaluate SCript: a simple program for blinking the on-board LEDs, an implementation of the Surge protocol [16], and an implementation of the Trickle network dissemination protocol [17]. We implemented the three programs both as Contiki programs in C and in SCript. The SCript implementation of Surge is shown in Figure 5. We compile our code with the MSP430 port of the GCC compiler version 3.2.3. We measure execution time on a MSP430 microcontroller clocked at 2.4576 MHz. We setup a timer interrupt that increases a counter every millisecond. To measure execution time we record the value of the timer before and after invoking the function to be measured and compare the two timer values. For every measurement, we invoke the function 100 times and calculate the average execution time.

5.1 Program Size

We compare the size of the three programs in textual script format, in tokenized format, compiled to CVM [3] byte code, and for the same program implemented as Contiki program and compiled to MSP430 machine code. To produce CVM byte code, we instrumented the SCript interpreter to produce byte code during parsing of the SCript scripts. The CVM is a regular stack-based virtual machine with single-byte operators. We extended the CVM with the possibility to call native code functions using the textual name of the native function. Table 1 shows the resulting program sizes for the three programs. The size of the textual script includes all comments in the code. We see that the size of the tokenized program is on par with that of the MSP430 machine code size and the CVM byte code.

```
void surge(int base) {
  while(1) {
    // Set timer 0 to 2 seconds
    timer(0, 10 * 2);

    // Wait until either a packet is
    // received or timer 0 expires
    wait(received() | expired(0));

    if(expired(0)) {
      int sensor_data;
      timer(0, 0); // Reset timer 0
      sensor_data = sensor(0);
      packet[0] = 1;     // Packet type,
      packet[1] = sensor_data;  // data
    }

    if(received()) {
      if(base) {
        "rs232_printint"(packet[1]);
      }
    }
  }
}
```

Figure 5: Example implementation of Surge in SCript.

Script	Script size	Tokenized size	CVM code size	MSP430 code size
LED blinker	163	99	69	136
Surge	513	167	121	162
Trickle	1379	448	352	386

Table 1: Size in bytes of the script, the tokenized script, the compiled CVM code, and the native MSP430 code for three programs.

5.2 Execution Overhead and Energy Costs

5.2.1 Tokenization Overhead

When a SCript script is inserted in a sensor network, the first node that gets the script tokenizes the script into a token stream which is redistributed across the network. In networks with a base station, the scripts can be tokenized by the base station. Tokenization is done once for every program to be distributed across a network. We measured the execution time for the tokenization process for the Surge and Trickle programs. Tokenization of the Trickle program, the largest of the three programs, takes 0.2 seconds while tokenization of the Surge program takes only 0.04 seconds.

5.2.2 Interpretation Overhead

We measure the execution time for nine simple SCript statements: SCript function call, native function call, pointer arithmetic and indexing, variable addition and assignment, variable multiplication and assignment, array indexing, an empty if statement, constant addition and variable assignment, and variable assignment. Table 2 shows the results of the measurements. We see that the execution time of tokenized SCript state-

SCript statement	Execution time, script (ms)	Execution time, tokenized (ms)	Tokenized, CPU cycles	MSP430 cycles	Speed ratio	Tokenized size	MSP430 size	Size ratio
function(a);	0.66	0.11	274			5		
”native”(a);	0.23	0.091	224	10	22:1	13	8	1.6:1
*(”array” + 1) = a;	0.24	0.083	204	9	23:1	16	12	1.3:1
a = b + c;	0.55	0.082	203	10	20:1	6	12	1:2
a = b * c;	0.55	0.081	200	20	10:1	6	26	1:4.3
a[1] = b;	0.41	0.072	176	9	20:1	7	12	1:1.7
if(a) {} else {}	0.29	0.068	168	5	34:1	9	8	1.1:1
a = 1 + 2;	0.28	0.046	113	3	38:1	6	6	1:1
a = 0;	0.25	0.035	85	3	28:1	4	6	1:1.5

Table 2: Execution time for nine basic SCript statements.

ments is between 10 and 38 times that of the native code equivalents.This is on par with the Maté virtual machine, which reports a 35 times slowdown [14]. For comparison, the execution of a non-tokenized versions of the same statements is five times that of the tokenized version. The size of the tokenized statements is about the same as the same statements in native code. However, the actual size of these statements in a native program is expected to be slightly smaller on average as a C compiler is able to optimize code across several statements.

To estimate the efficiency of executing a program in tokenized form compared to executing a program in script form we measured the execution time for the Surge script from Figure 5 both in tokenized form and in script form. The execution time of the program in tokenized form is 0.007 ms and the execution time of the program in script form 0.03 ms which is four times longer. From this we can conclude that tokenization is the most expensive operation in the SCript system. In comparison, we measured the the execution time of the Surge protocol implemented in C and compiled to native MSP430 code to be 0.0007 ms which is one tenth of the execution time of the tokenized Surge script.

5.2.3 Energy Costs

We define the energy costs of a program to be the sum of the energy cost of distributing the program and the energy cost of executing the program. Typically, the distribution energy costs are much higher than the execution energy costs [3]. We use the energy model from Dunkels et al. [3] to give estimates for the energy consumption of reprogramming a single sensor node with a SCript script. The model is based on measurements of the radio hardware on the Tmote Sky board together with the approximate overhead of the Deluge network code dissemination protocol [9]. With this model, an approximate lower bound on the energy cost for receiving the Surge program in Figure 5 3 mJ. We calculate an approximation of the energy consumption of one iteration of the Surge program by multiplying the power consumption of the Tmote Sky board with the microcontroller running [3] with the execution time of the tokenized Surge script. We find that lower bound of the network distribution energy is equal to the execution energy of 65000 iterations of the tokenized version of the Surge script. Thus the energy for executing the script is much smaller than distribution energy consumption.

5.3 Memory Footprint

Table 3 shows the memory footprint of the modules of the SCript interpreter. The RAM footprint of the interpreter is the total memory usage; the SCript interpreter does not allocate any dynamic memory. The largest part of the RAM footprint is the stack of the SCript interpreter thread. The required size of the stack depends on the structure of the scripts that the interpreter will be executing. This cannot be predetermined unless all scripts that the interpreter ever will run are analyzed beforehand. Scripts with many nested compound statements and function calls will need a larger stack. To accommodate a wide range of different scripts we intentionally overprovision the stack. To come up with an estimate of the required stack size we measured the amount of stack space used for our three programs and found that they

Module	ROM footprint	RAM footprint
SCript interpreter	2954	42
SCript thread stack		266
SCript symbol table	272	60
Script tokenizer	1384	8
Pre-tokenizer	600	24
Contiki interface	174	12
Total SCript interpreter	**5384**	**372**
SCript packet interface	86	4
SCript packet buffers		64
SCript timer interface	92	32
Total SCript extensions	**178**	**100**
Contiki multi-threading	284	6

Table 3: Memory footprint for the SCript interpreter, the SCript extensions, and the Contiki multi-threading library.

required a maximum of 130 bytes. To allow for larger scripts, we set the stack size to twice of that: 260 bytes. The additional six bytes are used for housekeeping variables.

The memory footprint of the SCript interpreter is similar to that of most virtual machines designed for sensor networks. The ROM footprint of the Maté machine [14] is 7 kilobytes and the RAM footprint 602 bytes. DVM [1] has a ROM footprint of 13 kilobytes and the footprint of the VM* environment is between 5 and 10 kilobytes. The VM* RAM footprint is between 500 and 2000 bytes. The Tapper command environment [23] has a ROM footprint between 3.5 and 5.2 kilobytes and a RAM footprint between 1 and 4 kilobytes.

Compared to other structured script interpreters for sensor networks and embedded systems, the SCript interpreter is significantly smaller. The Sensorware tinyTcl interpreter [2] requires 74 kilobytes of ROM and runs on devices with as much as 64 megabytes of RAM. The ROM footprint of the Lua interpreter [10] is 63 kilobytes.

6 Conclusions

Script languages are a popular approach to reprogramming in general-purpose computing but have previously not been considered for wireless sensor networks because of the perceived high interpretation overhead. We have presented SCript, a structured script language for tiny sensor nodes, that has many of the features found in general-purpose scripting languages, including if statements and while loops, functions, and scoped variables. We have developed an interpreter for the language that runs on the MSP430 microcontroller used in many popular sensor node platforms. The ROM and RAM footprint is similar to that of existing virtual machines for sensor networks. We measure the execution time overhead of the SCript interpreter on real hardware and show that it is on par with the execution time overhead of virtual machines. Unlike virtual machines, the script language approach does not require recompilation and special compilers. Thus script languages, previously considered as too expensive for tiny sensor nodes, are a viable alternative to virtual machines.

Acknowledgments

This work was partly financed by the European Commission under contract IST-004536-RUNES.

References

[1] R. Balani, S. Han, R. Rengaswamy, I. Tsigkogiannis, and M. B Srivastava. Multi-level software reconfiguration for sensor networks. In *ACM EMSOFT*, October 2006.

[2] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[3] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, 2006.

[4] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, 2006.

[6] J. Hahn, Q. Xie, and P. H. Chou. Rappit: framework for synthesis of host-assisted scripting engines for adaptive embedded systems. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2005.

[7] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM Press, 2005.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[9] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[10] R. Ierusalimschy, L. Henrique de Figueiredo, and W. Celes Filho. Lua — an extensible extension language. *Software — Practice and Experience*, 26(6):635–652, 1996.

[11] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[12] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, 2005.

[13] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proc. SenSys'05*, San Diego, CA, USA, November 2005.

[14] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[15] P. Levis, D. Gay, and D Culler. Active sensor networks. In *Proc. USENIX/ACM NSDI'05*, Boston, MA, USA, May 2005.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137, 2003.

[17] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of NSDI'04*, March 2004.

[18] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the European Workshop on Wireless Sensor Networks, EWSN 2006*, 2006.

[19] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3), 1998.

[20] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[21] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[22] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - low power nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences*, Hawaii, USA, 2005.

[23] Q. Xie, J. Liu, and P. H. Chou. Tapper: a lightweight scripting engine for highly constrained wireless sensor nodes. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks (Poster session)*, 2006.