# The TinyScript Language

Philip Levis
pal@cs.berkeley.edu

Version 1.0
July 12, 2004

## Contents

## 1 Introduction

This document describes TinyScript, a language that the Maté VM framework supports for programming sensor networks. TinyScript is an imperative, BASIC-like language with dynamic typing and basic control structures such as conditionals and loops.

## 2 Script Structure

This is an example TinyScript program that increments a counter:

```
! Define a shared variable, counter
shared counter;

! Increment it
counter = counter + 1;
```

In TinyScript programs, all variables must be declared before any program statements. For example, the following program is invalid (and will throw a compilation error):

```
! Define a shared variable, counter
shared counter;

! Increment it
counter = counter + 1;

! Define a shared variable, index: ERROR
shared index;
```

Statements generally end with a semicolon. `!` declares the start of a comment, which extends to the end of a line. For example,

```
shared counter; ! Define a shared variable, counter
counter = counter + 1; ! Increment it
```

is valid TinyScript code.

TinyScript function and variable identifiers are case insensitive: `var` is the same as `VAR` or `Var`. Identifiers are composed of alphanumeric characters and the underscore ('_'), but the first character of an identifier must be a letter or the underscore. The following are all valid identifiers:

```
temp
a51_b
TEMP
temp7
buffer_Index
_3
```

The following are invalid identifiers:

```
@a
4b
sd?
a 5
```

To be precise, identifiers must follow the pattern `[A-Za-z_][A-Za-z0-9_]*`.

Certain words are TinyScript keywords, and cannot be used in programs to name variables or functions. In the above scripts, `shared` is a keyword, declaring `counter` to be a shared variable. The full list of keywords is:

```
not     and     or      xor     eqv     imp
for     to      next    step    until   while
end     private shared  buffer  if       then
else
```

TinyScript function and variable identifiers are case insensitive: `var` is the same as `VAR` or `Var`. Keywords exist as both their uppercase and lowercase versions: `shared` can also be written `SHARED`, but cannot be written `sHarEd`.

## 3 Variables

TinyScript programs have two basic variable types: scalars and buffers. Scalars represent a single data item, such as an integer or a sensor reading. Buffers are small collections of values. In the above programs, the keyword `shared` declared the variables to be scalars. Handlers can declare two kinds of scalars: `private` and `shared`. Private variables are local to that handler; the statement `private a;` in two different handlers refers to two different variables. In contrast, `shared a;` in two different handlers refer to the same variable. Using a shared variable, handers can pass data to one another. Buffers, declared with `buffer`, are implicitly shared variables.

Both values and buffers are dynamically typed. That is, the variables themselves have no explicit type in a program; instead, their type is determined dynamically as a program runs. In this program,

```
shared counter;
shared sensor;
counter = random();
sensor = light();
```

the variable `counter` takes the type integer (`rand()` returns an integer) while `sensor` takes the type light (`light()` returns a light value).

Types constrain how values can be modified, and how buffers can be accessed. There is one basic scalar type, integer (16-bit, signed). Additionally, every sensor has its own type. Integers can be modified freely, through arithmetic, assignment, and other transformation. Sensor readings, however, are immutable. You cannot add two sensor readings, even if from the same sensor.

The idea is that sensor readings should only be what is actually read from a sensor. Transformations on these readings should be distinguishable from actual readings. To modify sensor readings, they must be cast to an integer with the `int()` function. For example, this program computes an exponentially weighted moving average of the light sensor:

```
shared sensor;
shared aggregate;

sensor = light();
aggregate = (aggregate / 2) + (int(sensor) / 2); ! Cast light reading to integer
```

Buffers also have a type, which defines what values can be placed in it. A cleared buffer has no type, and takes the type of the first value placed in it. In the following program, `aggBuffer` is cleared, which clears its type. An integer (`aggregate`) is added to the buffer, making the buffer of the type integer.

```
shared sensor;
shared aggregate;
buffer aggBuffer;

sensor = light();
aggregate = aggregate + int(sensor); ! Cast light reading to integer
aggregate = aggregate / 2;

bclear(aggBuffer); ! Clear all buffer entries and type
aggBuffer[] = aggregate; ! Append aggregate value to buffer
                         ! Buffer is now of type integer
```

Buffers have a fixed maximum size of ten values. The function `bfull()` can be used to see if a buffer is full, while `bsize()` indicates how many entries it currently has. Individual buffer values can be accessed by indexing into a buffer. The following program obtains the median value stored in a buffer:

```
shared size;
shared median;
buffer aggBuffer;

bsorta(aggBuffer);        ! Sort buffer entries in ascending order
size = bsize(aggBuffer);  ! Number of entries in buffer
median = aggBuffer[size / 2];  ! Return median value
```

An empty index value implies the tail (last value) of a buffer on access, or after the tail on assignment (append). For example:

```
buffer aggBuffer;
shared val;

val = aggBuffer[];             ! Val is the last value in the buffer
aggBuffer[] = light();    ! Append a new light value to the buffer
```

## 4 Functions

The above code examples used several functions, such as `light()`, `bsorta()`, and `int()`. Functions take a fixed number (zero or more) of parameters. For example, `bclear()` takes a single parameter, a buffer to clear, and `rand()` takes no parameters. Some functions return values (e.g., `rand()`), while others do not (e.g., `bsorta()`).

Function parameters may have type requirements. However, as TinyScript is dynamically typed, these types are not checked at compile time. For example, `bclear()` takes a single parameter, a buffer. Passing it an integer will cause a run-time error.

Return values of function calls may be ignored. For example,

```
rand();
```

is a valid program.

The return values of functions can be directly used as values or parameters to functions:

| Name | Operator | Example |
|---|---|---|
| Addition | + | val = val + 2; |
| Subtraction | - | val = a - b; |
| Division | / | val = bsize() / 2; |
| Multiplication | * | val = 2 * b; |
| Exponentiation | ^ | val = val ^ 2 |

(a) Arithmetic Operations

| Name | Operator | Example |
|---|---|---|
| Less than | < | val = val < 2; |
| Greater than | > | val = a > b; |
| Less than or equal | <= | val = bsize() <= 8; |
| Greater than or equal | >= | val = b >= 2; |
| Not equal | <> | cond = val <> b |

(b) Comparison Operations

Figure 1: TinyScript Computational Primitives

```
buffer aggBuf;
shared val;

val = aggBuf[rand() \% bsize(aggBuff)]; ! Hope size isn't zero
val = sqrt(bsize(aggBuff)) + 2;
```

Note that assigning to a buffer is different than assigning to an element of a buffer. Assigning a scalar to a buffer appends it. If a scalar is assigned to an element of a buffer, the buffer is dynamically sized to include that element if need be. Buffers can be assigned to one another (which results in a copy), but a buffer cannot be assigned to be the element of another buffer.

```
buffer aggBuf;
buffer aggBuf2
shared val;

aggBuf2 = aggBuf;
aggBuf[] = val;
aggBuf2[val \% 8] = val;
aggBuf[1] = aggBuf2;  ! ERROR
```

Currently, TinyScript does not support scripting functions.

## 5 Arithmetic, Logic, and Conditionals

Figure 1(a) shows the set of arithmetic operations TinyScript provides, as well as their syntax.

TinyScript also supports logical operations, which are show in Figure 2(a). All of these operations only accept integers as operands. For the boolean operators (e.g., and, not), a value of zero is considered false; all other values are considered true. All operators use 0 as false and 1 as true. So, `1 and 2` resolves to 1, while `0 and 34` resolves to 0. Figure 2 contains the truth tables for the boolean operators.

| Name | Operator | Example |
|---|---|---|
| And | AND, and | ready = full and idle; |
| Or | OR, or | ready = full OR idle; |
| Not | NOT, not | ready = not idle; |
| Exclusive or | XOR, xor | diff = a XOR b; |
| Equivalent | EQV, eqv | rval = a eqv b; |
| Implies | IMP, imp | ready = a imp b; |
| Logical And | & | bits = packetbits & mask; |
| Logical Or | \| | bits = firstbit \| secondbi t; |
| Logical Not | ~ | mask = ~bits; |

(a) Logical Operations

|  | and |  | or |  | not |
|---|---|---|---|---|---|
|  | F | T | F | T |  |
| F | **F** | **F** | **F** | **T** | **T** |
| T | **F** | **T** | **T** | **T** | **F** |

|  | xor |  | eqv |  | imp |  |
|---|---|---|---|---|---|---|
|  | F | T | F | T | F | T |
| F | **F** | **T** | **T** | **F** | **T** | **T** |
| T | **T** | **F** | **F** | **T** | **F** | **T** |

(b) Truth Tables

Figure 2: TinyScript Logical Primitives

The logical operations manipulate integer bit fields. Instead of manipulating the integer as a singe value, they operate on each bit, in a manner similar to C operators. For example, `1 and 2` resolves to 1, while `1 & 2` resolves to zero (1 and 2 share no common bits), and `1 | 2` resolves to 3.

Finally, TinyScript has standard comparison operators, as shown in Figure 1(b). They resolve to one if true, zero if false.

## 5.1 Control Structures

TinyScript supports standard language control structures such as conditionals and loops.

The first set of control structures, conditionals, take this form:

```
if <expression> then
 <block 1>
end if
```

```
if <expression> then
 <block 1>
else
 <block 2>
end if
```

If expression resolves to true, then block 1 executes. If the statement has an else clause and expression resolves to false, then block 2 executes. There can be nested if-then statements:

```
shared idle;
buffer buf;

if bfull(buf) then
  idle = 0;
  if rand() & 1 then
    send(buf);
    bclear(buf);
  end if
  idle = 1;
end if
```

TinyScript provides loops through the `for` construct. There are two basic forms, unconditional and conditional. Unconditional (for-to) loops run a specific number of times; their termination condition when the loop variable takes a specific value. Conditional (for-until) loops run until an arbitrary condition becomes true. `next` defines the end of the loop block, and increments the loop variable. By default, the variable increments by one. However, the increment step can be set with the `step` keyword. In summary:

```
for <x> = <expression> to <to-constant>
  ...
next <x>

for <x> = <expression> to <to-constant> step <step-constant>
  ...
next <x>

for <x> = <expression> until <until-exp>
  ...
next <x>

for <x> = <expression> step <step-constant> until <until-exp>
  ...
next <x>
```

For example, this loop will run one hundred times, blinking the leds,

```
private i;

for i = 1 to 100
  leds(i & 7)
next i
```

while this loop will put the values 1,3,5...21 in the buffer (when it has ten values, it will be full),

```
private i;
buffer buf;

bclear(buf);
for i = 1 step 1 until i > 10
  buf[] = i * 2;
next i
```

Standard while loops can be implemented by setting a step of zero. This loop, for example, will put random values into a buffer until it is full:

```
private i;
buffer buf;

for i = 0 step 0 until bfull(buf)
  buf[] = rand();
next i
```

Parenthesis pairs can be added to define precedence, or for readability.

```
private i;
buffer buf;

bclear(buf);
for i = 1 step 1 until i > 10
  buf[] = ((i * 2) + 1);
next i
i = (5 + 2 * 2);    ! i = 9
i = (5 + 2) * 2;    ! i = 14
i = ((((5))));      ! i = 5
```

Maté is under active development. Bugs, contexts, and functions can be sent to Phil Levis (`pal@cs.berkeley.edu`).

## 6 Appendix A: Grammar

*TinyScript-file:*
    *variable-list*$_{OPT}$ *statement-list*$_{OPT}$

*variable-list:*
    *variable*
    *variable-list variable*

*variable:*
    `shared` *identifier*
    `private` *identifier*
    `buffer` *identifier*

*statement-list:*
    *statement*
    *statement-list statement*

*statement:*
    *assignment*

*control-statement*
*function-statement*

*control:*
*if-statement*
*for-unconditional*
*for-conditional*

*if-statement:*
`if` *expression* `then` *then-clause* `end if`

*then-clause:*
*statement-list*
*statement-list* `else` *statement-list*

*for-unconditional:*
`for` *variable-ref* `=` *expression* `to` *constant-expression step statement-list* `next` *variable-ref*

*for-conditional:*
`for` *scalar-ref* `=` *constant-expression step for-condition statement-list* `next` *scalar-ref*

*for-condition:*
`until` *expression*
`while` *expression*

*constant-expression:*
*constant*

*function-statement:*
*function-call* `;`

*function-call:*
*name* `(` *parameter-list*$_{OPT}$`)`

*parameter-list:*
*parameter*
*parameter-list* `,` *parameter*

*parameter:*
*expression*

*assignment:*
*l-value* `=` *expression* `;`

*l-value:*
*var-ref*
*buffer-ref*

*scalar-ref:*
*identifer*

*var-ref:*
*identifier*

*buffer-ref:*
*identifier* `[]`
*identifier* `[` *expression* `]`

*expression:*
*function-call*
*constant-expression*
*variable-expression*
*paren-expression*
*unary-expression*
*binary-expression*
*conditional-expression*

*variable-expression:*
*buffer-access*
*variable-access*

*buffer-access:*
*identifier* `[]`
*identifier* `[` *expression* `]`

*variable-access:*
*identifier*

*paren-expression:*
`(` *expression* `)`

*unary-expression:*
`-` *expression*
≈ *expression*
`NOT` *expression*

*binary-expression:*
*expression* `+` *expression*
*expression* `-` *expression*
*expression* `*` *expression*
*expression* `/` *expression*
*expression* `%` *expression*
*expression* `&` *expression*
*expression* `|` *expression*
*expression* `#` *expression*
*expression* `AND` *expression*
*expression* `XOR` *expression*
*expression* `OR` *expression*
*expression* `EQV` *expression*
*expression* `IMP` *expression*

*conditional-expression:*
*expression* = *expression*
*expression* != *expression*
*expression* >= *expression*
*expression* > *expression*
*expression* <= *expression*
*expression* < *expression*

*constant:*

```
[1-9][0-9]*
```

*identifier:*

```
[A-Za-z_][A-Za-z_0-9]*
```