TinyScript Manual

1 Introduction

TinyScript is a programming language for wireless sensor network. It lets programmers specify how each node reacts to certain events. We assume an one-hop network structure, in which node 0 (the root node) is connected to a base station through its universal asynchronous receiver/transmitter (UART), while other nodes can directly communicate with node 0 over radio. In this test, the computer in front of you is the base station.

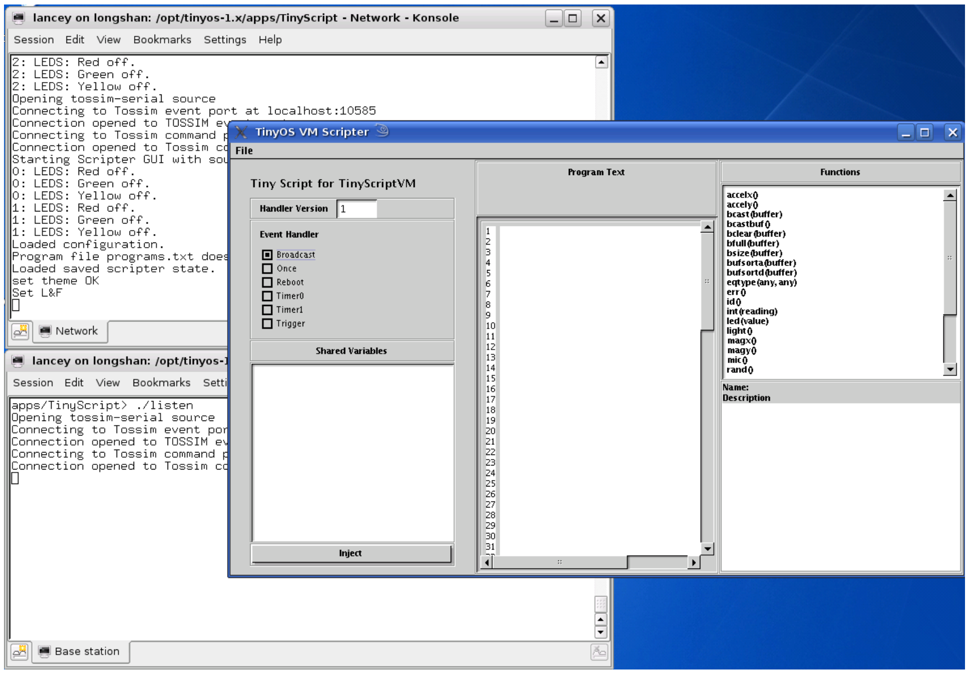
When you start the test, three windows will be open for you on the desktop, as shown in Figure 1. Please keep them open during the test. If you close any of these windows by accident, please ask your instructor to reopen it. In Figure 1, the window at the top layer is the **TinyScript programming environment**, where you programs are edited and run. The other two windows behind show simulation results. A network composed of four nodes is simulated in this test. The **Network** window shows the status of all the nodes in the network, including LEDs, sampled data, and transmitted data. The **Base station** window displays data received at the base station.

2 Concepts and Definitions

- **Identifiers** are composed of alphanumeric characters and underscores (_). The first character of an identifier must be a letter or an underscore. For example, *mydata*, *abc_3*, and *_3m* are identifiers, while *s@* and *3abc* are not.
- **Variables** are locations for storing data. Variable names must be identifiers. TinyScript variables are case insensitive: *var* is the same as *VAR* or *Var*.
- **Keywords** are certain words that cannot be used in programs to name variables. Keywords may be written entirely in upper-case letters or entirely in lower-case letters. For example, *shared* can also be written *SHARED*, but cannot be written *sHarEd*. The full list of TinyScript keywords follows:

  for to next step until end if then else private shared buffer not and or

- **Functions** are simply ways to execute a procedure, which may produce a value, modify variables, or trigger an event. A function takes a fixed number (zero or more) of parameters. Some functions return values, some do not. The return values of functions can be directly used as values or parameters to functions. Functions are all provided by TinyScript. You can check their definitions in the **TinyScript programming environment**.
- **Comments** are used to insert notes in one's program. They do not affect functionality. In TinyScript, ! indicates the start of a comment, which extends to the end of a line. For example,


lancey on longshan: /opt/tinyos-1.x/apps/TinyScript - Network - Konsole
Session Edit View Bookmarks Settings Help
Network
lancey on longshan: /opt/tinyos-1
Base station
TinyOS VM Scripter
File
Tiny Script for TinyScriptVM
Handler Version 1
Event Handler
Shared Variables
Inject
Program Text
Functions
Name:
Description

```
2: LEDS: Red off.
2: LEDS: Green off.
2: LEDS: Yellow off.
Opening tossim-serial source
Connecting to Tossim event port at localhost:10585
Connection opened to TOSSIM ev
Connecting to Tossim command p
Connection opened to Tossim co
Starting Scripter GUI with sou
0: LEDS: Red off.
0: LEDS: Green off.
0: LEDS: Yellow off.
1: LEDS: Red off.
1: LEDS: Green off.
1: LEDS: Yellow off.
Loaded configuration.
Program file programs.txt does
Loaded saved scripter state.
set theme OK
Set L&F
```

```
apps/TinyScript> ./listen
Opening tossim-serial source
Connecting to Tossim event por
Connection opened to TOSSIM ev
Connecting to Tossim command p
Connection opened to Tossim co
```

[✓ Broadcast] Once Reboot Timer0 Timer1 Trigger

accelx()
accely()
bcast(buffer)
bcastbuf()
bclear(buffer)
bfull(buffer)
bsize(buffer)
bufsorta(buffer)
bufsortd(buffer)
eqtype(any, any)
err()
id()
int(reading)
led(value)
light()
magx()
magy()
mic()
rand()

Figure 1: Programming and simulation environment.

Event name	When triggered
Broadcast	Broadcast occurs when the mote receives a message from another mote.
Once	Once occurs when the once handler is initially sent to the mote.
Reboot	Reboot occurs whenever the mote is reset.
Timer0	Timer0 occurs when the the first timer fires. The timer can be set using the *settimer0* function.
Timer1	Timer1 occurs when the the second timer fires. The timer can be set using the *settimer1* function.
Trigger	Trigger occurs when the *trigger* function is called.

Table 1: Events in TinyScript.

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

3 TinyScript Programming Language Construct

3.1 Events and Handlers

In TinyScript, a program is broken into pieces called **handlers** that are triggered by **events**. **Handlers** are the code executed after a particular event. An event is said to "fire" when the criteria for that event are met. For example, the *reboot* event is fired when the mote is powered on or reset. Table 1 shows the six events in TinyScript.

3.2 Variables and Data Types

TinyScript has three kinds of variables: **private**, **shared**, and **buffer**. **Private** variables are local to a handler; only the handler that declares the variable can access it. For example, if two handlers both have a private variable named *counter*, each one has its own, independent variable. In contrast, **shared** variables are not unique to handlers; this allows two handlers to share data. If two handlers both have a shared variable named *counter*, they can both read and write the same variable. **Shared** variables are only shared between handlers on the same node; they are not shared across nodes.

Buffer variables are arrays of a fixed maximum size, and are always shared. Buffers have a fixed maximum size of 14 values. If you store more than 14 data elements into a buffer, it will cause a buffer overflow error. 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. The tail of a buffer can be appended or removed. For example:

```
val = aggBuffer[]; ! Remove the last value in the buffer and assign it to val
aggBuffer[] = light(); ! Append a new light value to the buffer
```

All variables in TinyScript must be declared before any program statements. A variable declaration specifies the variable type (**private**, **shared**, or **buffer**), followed by the variable name and a semicolon. For example, the following program is invalid (and will produce a compilation error):

```
shared counter; ! Declare a shared variable, counter
counter = counter + 1; ! Increment it
shared index; ! Declare another shared variable, index: ERROR
```

TinyScript has two basic data types: **integers** (positive whole numbers in the range of 0 to 32,767) and **sensor readings**. For example:

```
private val;
val = 1; ! val is an integer
val = light(); ! val is now a light reading
val = magX(); ! val is now a magnetometer reading (x-axis)
```

A **buffer** variable also has a type, which defines what values can be placed in it. A buffer can only contain values of a single type. A buffer's contents and type can be cleared with the *bclear* function. A buffer takes the type of the first value put into it.

```
buffer bufOne;
buffer bufTwo;
bclear(bufOne); ! clear bufOne
bufOne[0] = 5; ! Put 5 in index 0: bufOne has size one, type integer
```

```
bufOne[] = id(); ! Append mote ID to bufOne, now has size two
bufOne[4] = 41; ! Put 41 in index 4; bufOne has size five, buf[2] and buf[3] are 0
bufOne[5] = light(); ! error: buffer is type integer, not light
bufOne[5] = int(light); ! legal: integer
```

3.3 Expression

Data in TinyScript can be manipulated using a number of operators in what is known as an expression. TinyScript supports logical operations, arithmetic operations, and comparison operations. Logical operations include **and, or**, and **not**. Arithmetic operations include **+** (add), **-** (subtract), ***** (multiply), and **/** (divide). Comparison operations include < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), and <> (not equal). Parenthesis pairs can be added to define precedence, or for readability. For example,

```
i = (5 + 2) * 2; ! 5 and 2 are first added, then the result is multiplied by 2. i = 14
```

Data types limit what operations on a variable are valid. Sensor readings are immutable. You cannot add an integer to a light reading, a light reading to a temperature reading, or even two light readings. If you want to process sensor readings, you must turn them into integers with the *int* function. For example,

```
private total;
private count;
private val;
val = light(); ! legal: val is now a light reading
val = light() / 2; ! error: cannot divide light reading by 2
val = light() + magX(); ! error: cannot add light and magX
val = light() + light(); ! error: cannot add two light readings
val = int(light()); ! legal: val is now an integer
total = total + val; ! legal: total was an int
count = count + 1; ! legal: count was an int
val = total/count; ! legal: average of readings
```

3.4 Control Structures

Control structures control the order in which statements in a program are executed. So far, all the examples execute lines of code sequentially, in order. Sometimes, you may want your program to skip statements or repeatedly execute part of the program.

The first set of control structures, **conditionals**, cause a program to perform different actions based on certain conditions. They take the following forms:

```
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, in other words, block 1 and block 2 can also contain if-then or if-then-else statement.

The **for** construct allows loops to be specified. There are two basic forms of loops, unconditional and conditional. Unconditional (for-to) loops run a specific number of times; they terminate when the loop variable takes a specific value. Conditional (for-until, for-while) 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 size can be set with the **step** keyword. The loop control structure takes the following forms:

```
for <x> = <expression> to <to-constant>
    <block1>
next <x>
for <x> = <expression> to <to-constant> step <step-constant>
    <block1>
next <x>
for <x> = <expression> until <until-expression>
    <block1>
next <x>
for <x> = <expression> step <step-constant> until <until-expression>
    <block1>
next <x>
```

For example, the following loop runs one hundred times, incrementing *count* from 1 to 101.

```
private i;
private count;
count = 1;
for i = 0 to 100
    count = count + 1;
next i
```

While this loop puts the values 2,4,6...,20 in the buffer.

```
private i;
buffer buf;
for i = 2 step 2 until i > 20
    buf[] = i;
next i
```

You can set the step to zero if the loop variable is not used in the conditional or the enclosed code block. This loop, for example, puts random values into a buffer until it is full:

Argument	Action	Argument	Action	Argument	Action
9	red off	17	red on	25	red toggle
10	green off	18	green on	26	green toggle
12	yellow off	20	yellow on	28	yellow toggle

Table 2: The led function.

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

3.5 Communication

The *uart* function takes a buffer as a parameter and sends that buffer contents over the mote's UART. The following *Timer0* handler creates a buffer containing a node id and a light sample and sends it to the UART.

```
buffer data;
data[0] = id();
data[1] = int(light());
uart(data);
```

You also need to edit the *Once* handler to start timer 0:

```
settimer0(20); ! Fires timer 0 every 2 seconds.
```

The function *settimer0* controls the rate at which timer0 fires and triggers the *timer0* event. The parameter is in terms of tenths of a second, so 20 sets the period to two seconds. Calling *settimer0* with 0 will stop the timer.

If the above script is running, you should see output similar to that in Figure 2. The **Network** window monitors each node in the network. The number at the beginning of each line indicates the node id. The **Base station** window shows the data collected at the base station. You may notice that though there are four motes in the network, only data from node 0 arrive at the base station. This is because only mote 0 is connected to the base station via its UART. If you want to transmit data from other nodes to the base station, you should use the *bcast* function to send a buffer over the radio to all other nodes. If a mote hears a broadcast packet, it triggers the *Broadcast* handler. So you can use the *bcastbuf* function in the *broadcast* handler to retrieve the data.

4 Programming Environment

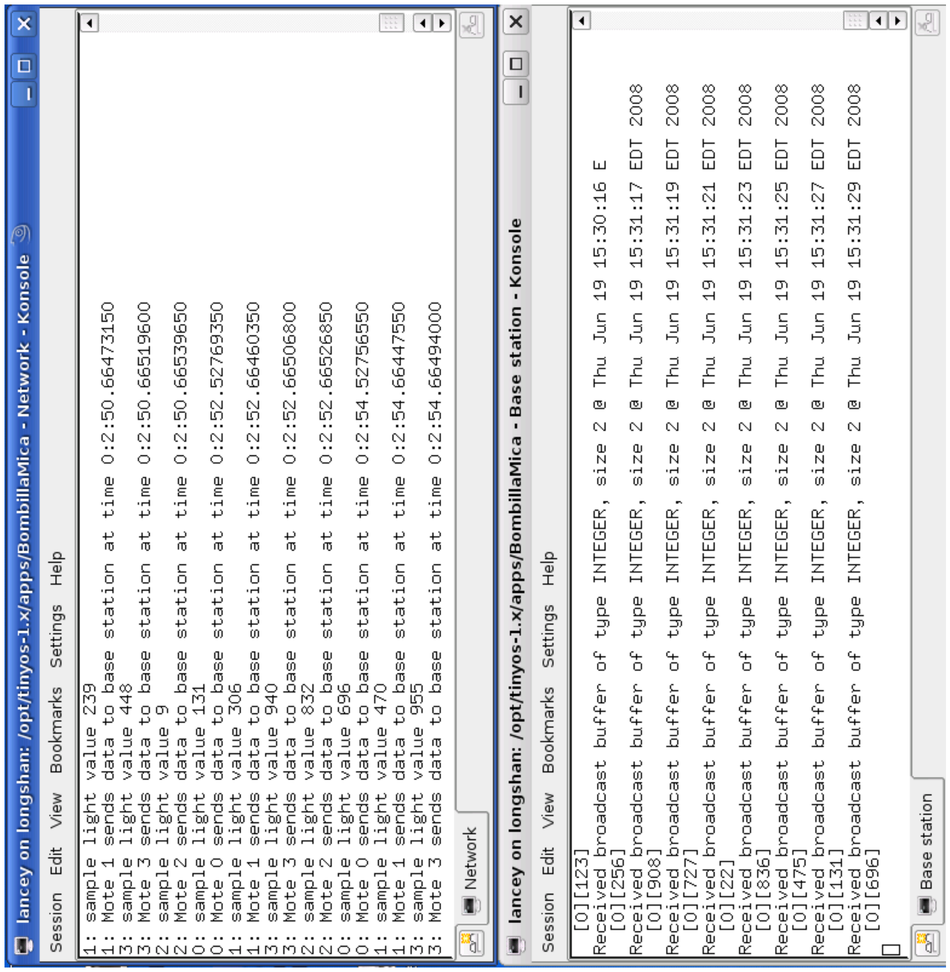
The TinyScript programming environment allows you to write and test TinyScript programs. As shown in Figure 6, the programming environment consists of several windows. The **Event Handler** buttons indicate which handler is selected for editing or injecting. The program for a selected handler is entered in the **Program Text** window and is executed by clicking the **Inject** button at the bottom left. Code entered in the **Program Text** window is not saved until the **Inject** button is clicked. Therefore, you may want to use the **Inject** button to save your programs before switching to another handler. Shared variables used in your programs are displayed in the **Shared Variables** window. The **Functions** window lists all the built-in functions. You can review the description of a function in the **Names Description** window by clicking on the function name in the list.

Please pay attention to error messages when you inject a handler or run the simulation. Error messages help you diagnose problems with your program. Grammar errors are detected when you inject a handler. For example, if you make a typo when you use the *uart* function and call it *uar*, a window as shown in Figure 3 will pop up when you try to inject your handler. You need to click the **OK** button in the window to close it first and go back to the **Program Text** field to fix the error. Your handler cannot be successfully injected or saved until all grammar errors are fixed. Other errors occur during runtime. For example, if you constantly append data to a buffer without clearing it, it will cause a buffer overflow error when you run the simulation. In this case, a window like Figure 4 will pop up when you run the simulation for a while. It indicates which handler causes this problem by pointing out in which context it happened. Error messages will also be printed in the **Network** window, as shown in Figure 5. In this case, you should stop the simulation, fix the error in the indicated handler and restart the simulation. Unfortunately, the error window does not go away even you fix the problem. You can ignore it and just monitor the **Network** window to see if the error is gone.

We will use a simple example to demonstrate how to work with the programming environment. First, select the *reboot* event in the **Event Handler** window. Then enter the following program in the **Program Text** window:

```
led(17);
```

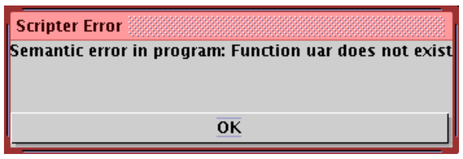
After clicking the **Inject** button, the **Network** window will print information as shown in Figure 7. It indicates that initially all the three LEDs on each node are off, then the red LED is illuminated. The **led** function is described in Table 2. If you do not get similar results, confirm that you have entered the correct program. If this doesn't fix the problem, please ask your instructor for help. If the text appears then you have successfully written your first TinyScript test program.


lancey on longshan: /opt/tinyos-1.x/apps/BombillaMica - Network - Konsole
Session Edit View Bookmarks Settings Help
Network
lancey on longshan: /opt/tinyos-1.x/apps/BombillaMica - Base station - Konsole
Base station

```
1: sample light value 239
1: Mote 1 sends data to base station at time 0:2:50.66473150
3: sample light value 448
3: Mote 3 sends data to base station at time 0:2:50.66519600
2: sample light value 9
2: Mote 2 sends data to base station at time 0:2:50.66539650
0: sample light value 131
0: Mote 0 sends data to base station at time 0:2:52.52769350
1: sample light value 306
1: Mote 1 sends data to base station at time 0:2:52.66460350
3: sample light value 940
3: Mote 3 sends data to base station at time 0:2:52.66506800
2: sample light value 832
2: Mote 2 sends data to base station at time 0:2:52.66526850
0: sample light value 696
0: Mote 0 sends data to base station at time 0:2:54.52756550
1: sample light value 470
1: Mote 1 sends data to base station at time 0:2:54.66447550
3: sample light value 955
3: Mote 3 sends data to base station at time 0:2:54.66494000
```

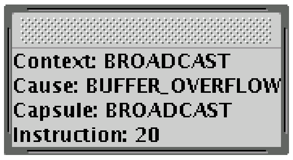
```
 [0][123]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:30:16 E
 [0][256]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:17 EDT 2008
 [0][908]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:19 EDT 2008
 [0][727]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:21 EDT 2008
 [0][22]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:23 EDT 2008
 [0][836]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:25 EDT 2008
 [0][475]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:27 EDT 2008
 [0][131]
Received broadcast buffer of type INTEGER, size 2 @ Thu Jun 19 15:31:29 EDT 2008
 [0][696]
```

Figure 2: Results on the monitor.



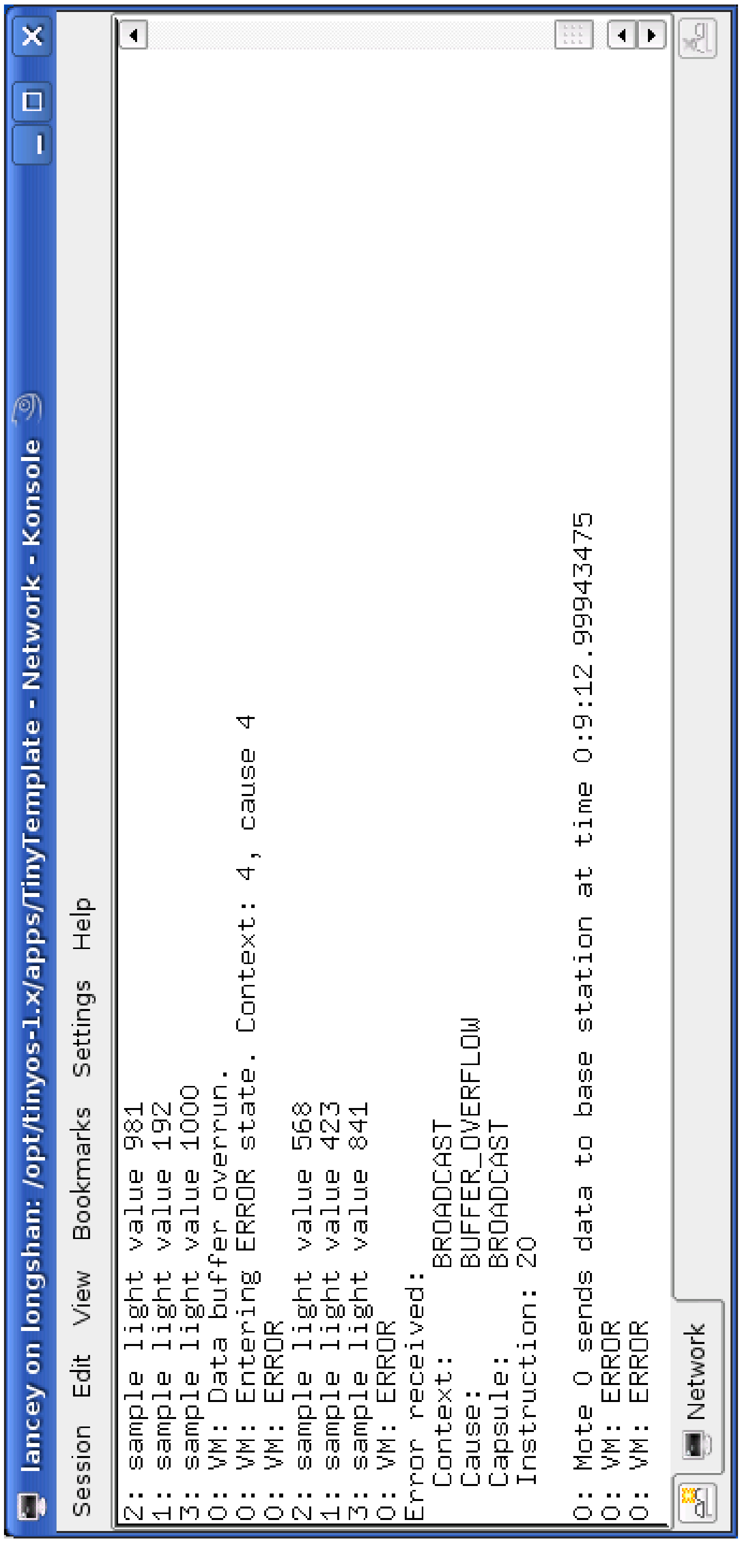
Semantic error in program: Function uar does not exist

Figure 3: Undefined function error.



Context: BROADCAST
Cause: BUFFER_OVERFLOW
Capsule: BROADCAST
Instruction: 20

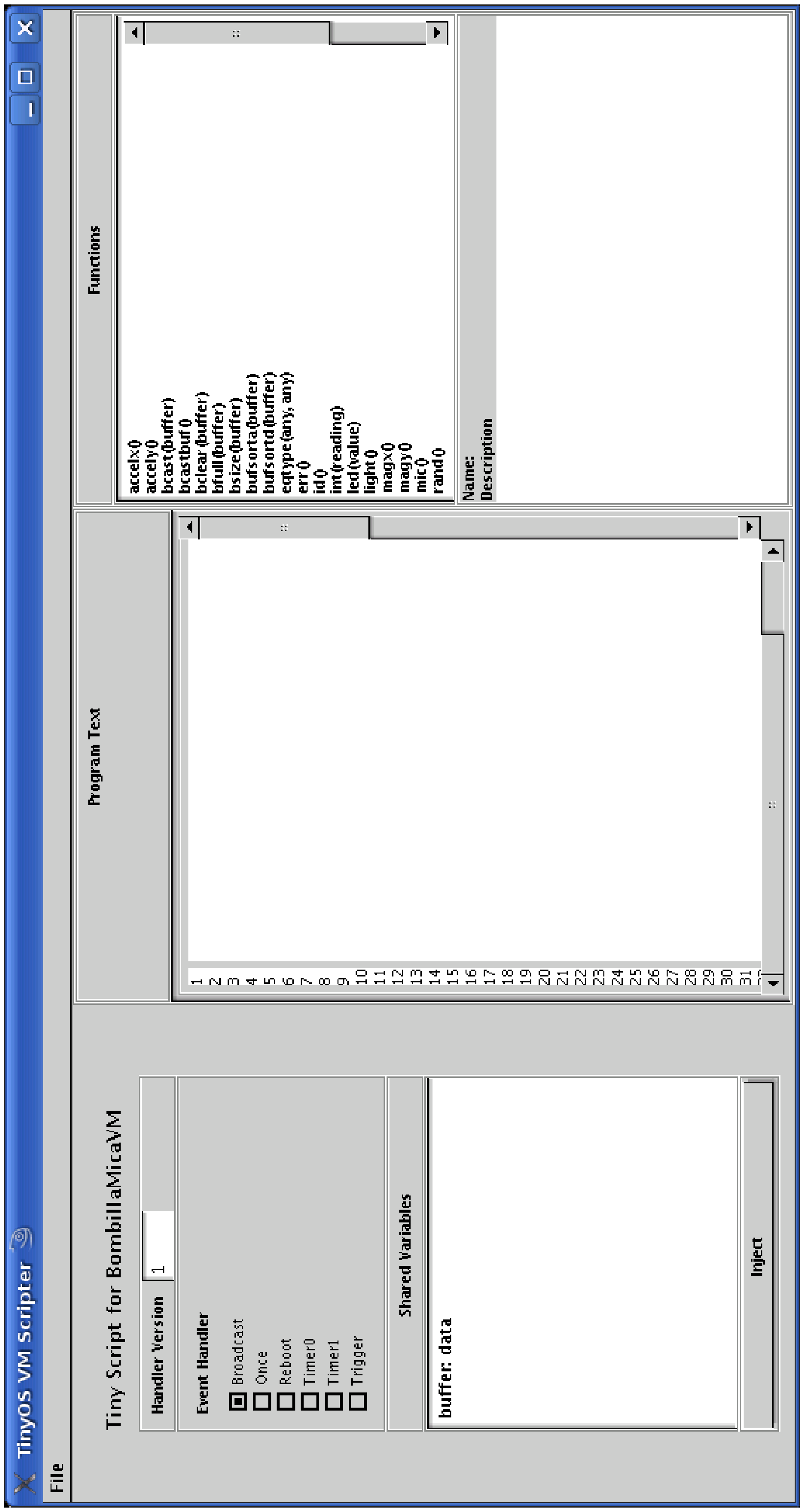
Figure 4: Buffer overflow error.


lancey on longshan: /opt/tinyos-1.x/apps/TinyTemplate - Network - Konsole
Session Edit View Bookmarks Settings Help
Network

```
2: sample light value 981
1: sample light value 192
3: sample light value 1000
0: VM: Data buffer overrun.
0: VM: Entering ERROR state. Context: 4, cause 4
0: VM: ERROR
2: sample light value 568
1: sample light value 423
3: sample light value 841
0: VM: ERROR
Error received:
  Context:     BROADCAST
  Cause:       BUFFER_OVERFLOW
  Capsule:     BROADCAST
  Instruction: 20

0: Mote 0 sends data to base station at time 0:9:12.99943475
0: VM: ERROR
0: VM: ERROR
```

Figure 5: Buffer overflow message in network window.



Tiny Script for BombillaMicaVM

Handler Version 1

Event Handler

- [✓ Broadcast]
- Once
- Reboot
- Timer0
- Timer1
- Trigger

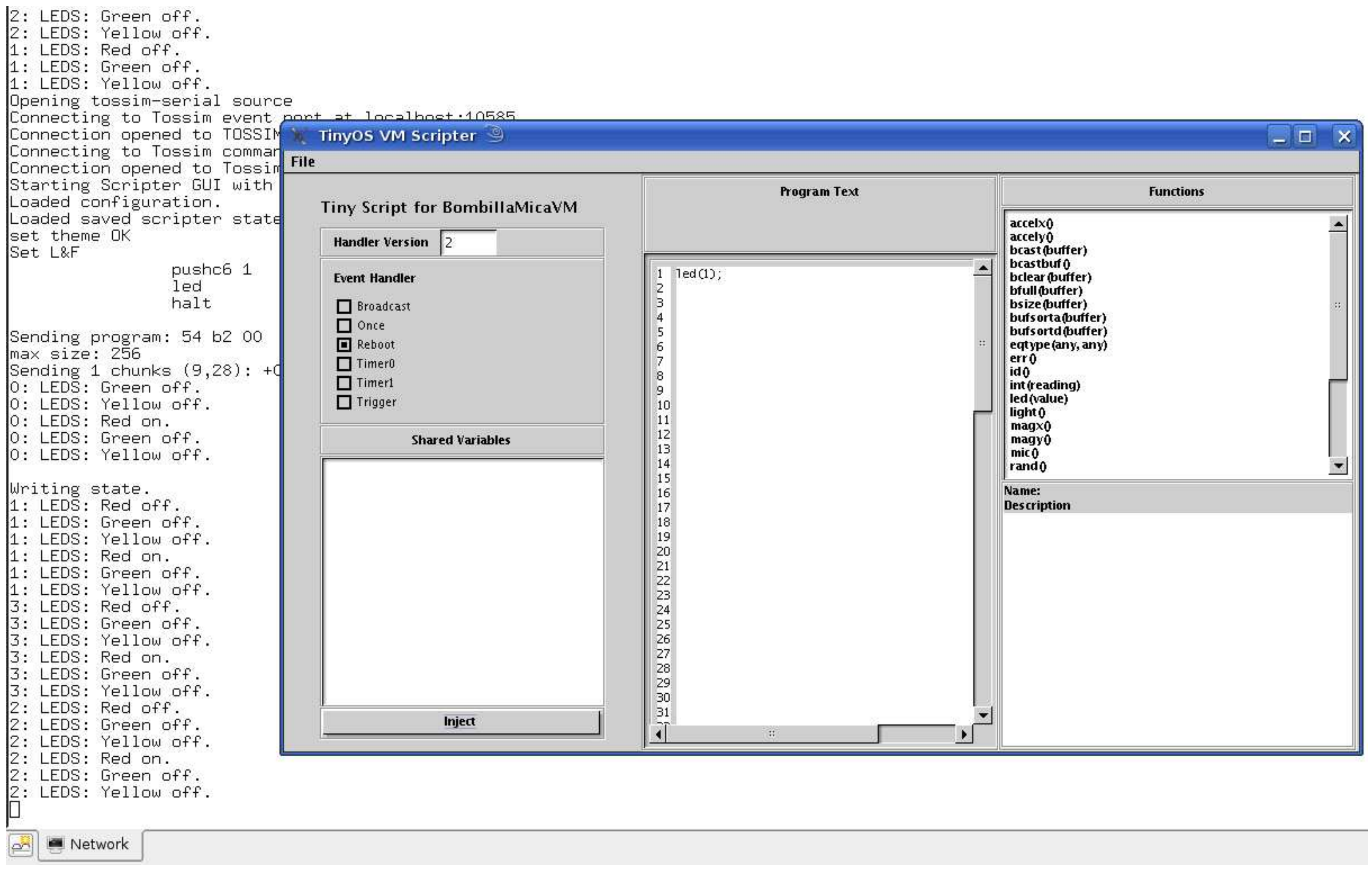
Shared Variables

buffer: data

```
accelx()
accely()
bcast(buffer)
bcastbuf()
bclear(buffer)
bfull(buffer)
bsize(buffer)
bufsorta(buffer)
bufsortd(buffer)
eqtype(any, any)
err()
id()
int(reading)
led(value)
light()
magx()
magy()
mic()
rand()
```

Name:
Description

Figure 6: Developing environment of TinyScript.



```
2: LEDS: Green off.
2: LEDS: Yellow off.
1: LEDS: Red off.
1: LEDS: Green off.
1: LEDS: Yellow off.
Opening tossim-serial source
Connecting to Tossim event port at localhost:10585
Connection opened to TOSSIM
Connecting to Tossim comman
Connection opened to Tossim
Starting Scripter GUI with
Loaded configuration.
Loaded saved scripter state
set theme OK
Set L&F
                pushc6 1
                led
                halt

Sending program: 54 b2 00
max size: 256
Sending 1 chunks (9,28): +0
0: LEDS: Green off.
0: LEDS: Yellow off.
0: LEDS: Red on.
0: LEDS: Green off.
0: LEDS: Yellow off.

Writing state.
1: LEDS: Red off.
1: LEDS: Green off.
1: LEDS: Yellow off.
1: LEDS: Red on.
1: LEDS: Green off.
1: LEDS: Yellow off.
3: LEDS: Red off.
3: LEDS: Green off.
3: LEDS: Yellow off.
3: LEDS: Red on.
3: LEDS: Green off.
3: LEDS: Yellow off.
2: LEDS: Red off.
2: LEDS: Green off.
2: LEDS: Yellow off.
2: LEDS: Red on.
2: LEDS: Green off.
2: LEDS: Yellow off.
```

TinyOS VM Scripter

Tiny Script for BombillaMicaVM

Handler Version: 2

Event Handler: Broadcast [ ] Once [ ] Reboot [✓] Timer0 [ ] Timer1 [ ] Trigger [ ]

Program Text:

```
led(1);
```

Figure 7: Example.