TinyTemplate Manual

1 Introduction

TinyTemplate is a programming language for a subset of wireless sensor network applications that use stationary nodes to periodically sample data and transmit them to a base station. TinyTemplate 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.

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 proctor to reopen it. In Figure 1, the window at the top layer is the **TinyTemplate 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 sampled data and data transmission to the base station. 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. TinyTemplate 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 TinyTemplate 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 TinyTemplate. You can check the definitions in the **TinyTemplate programming environment**.

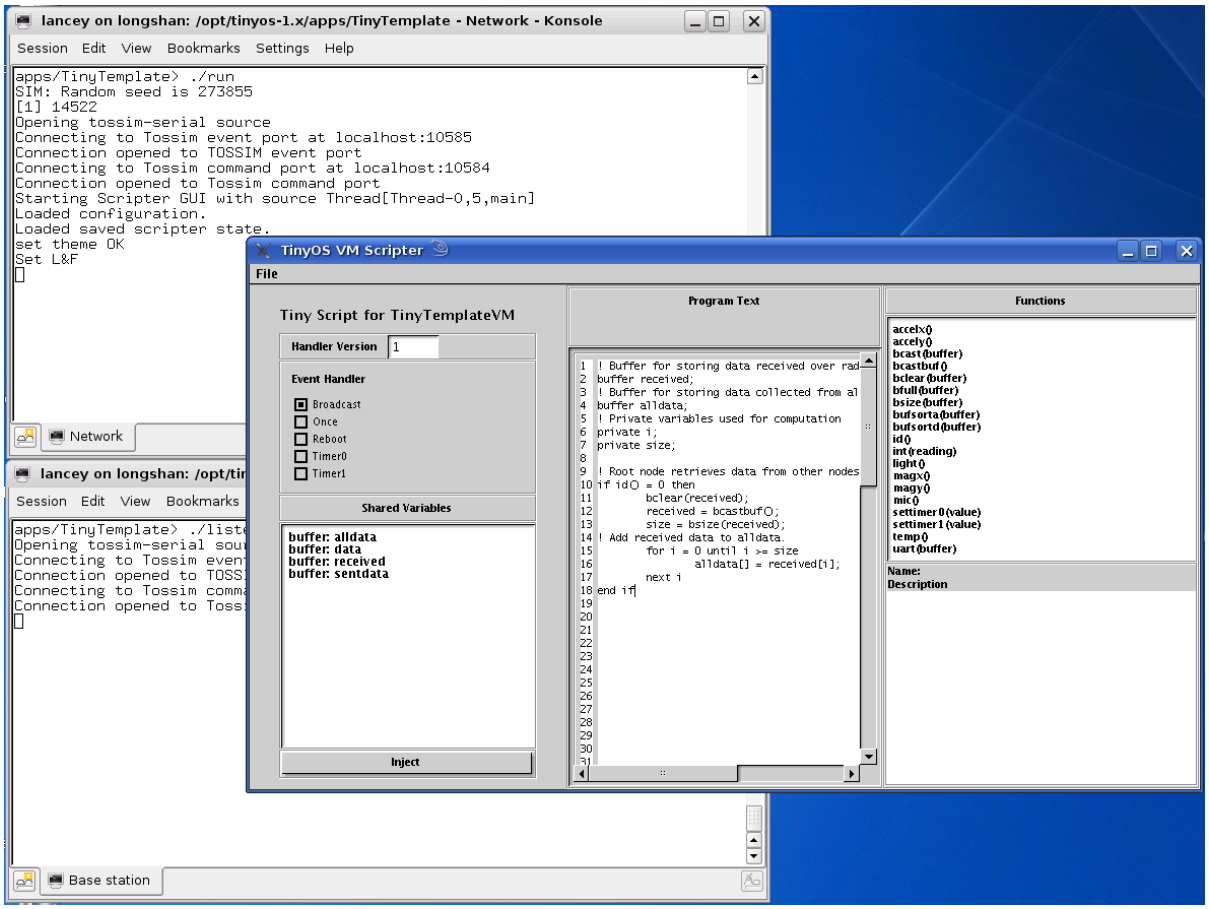

Figure 1: Programming and simulation environment.

- **Comments** are used to insert notes in one's program. They do not affect functionality. In TinyTemplate, ! indicates the start of a comment, which extends to the end of a line. For example,

  counter = counter + 1; ! Increment counter.

3 TinyTemplate Programming Language Construct

3.1 Events and Handlers

In TinyTemplate, 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 five handlers in TinyTemplate.

3.2 Variables and Data Types

TinyTemplate 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

Handler name	When the handler is triggered
Broadcast	when the mote receives a message from another mote over radio.
Once	when the once handler is initially sent to the mote.
Reboot	when the mote is reset. (Every time a handler is injected to the network, all the nodes reboot.)
Timer0	when the the first timer fires. The timer can be set using the *settimer0* function.
Timer1	when the the second timer fires. The timer can be set using the *settimer1* function.

Table 1: Events in TinyTemplate.

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 shared between handlers on the same node; they are not shared between 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. 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:

```
buffer aggBuffer;
aggBuffer[0] = 5; ! Put value 5 at the beginning of aggBuffer
aggBuffer[] = int(light()); ! Append a new light value to the buffer
val = aggBuffer[]; ! Remove the last value in the buffer and assign it to val
```

All variables in TinyTemplate 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.

TinyTemplate 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;
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 TinyTemplate can be manipulated using a number of operators in what is known as an expression. TinyTemplate 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 = (equal), < (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 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
```

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;
bclear(buf);
for i = 2 step 2 until i > 20
    buf[] = i;
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);
```

Note that even the *uart* function is called for every node, 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 Template

This section presents a template for TinyTemplate programs. It is also an example application that samples light every two seconds and transmits the sum of light readings from all the nodes in the network to the base station. Italic words are variables. Comments describe purpose of succeeding code segments and give hints on how to modify the template to create new applications.

Timer0 Handler:

```
! Collected samples
private sample;
! Buffer for storing local samples
buffer data;
! Buffer for storing data collected from all the nodes. Only root node uses this buffer.
buffer alldata;
! Buffer for storing data to be send to base station.
! Only root node uses this buffer.
buffer sentdata;
! Private variables used for computation.
private size;
private i;
! Data aggregation result
private sum;
! Sample sensor data
sample = int(light());
! Put sample to buffer data.
data[0] = sample;
if id() = 0 then
! Root node process alldata and put the aggregated results in sentdata.
! Replace with your code if you use other aggregation methods.
  size = bsize(alldata);
  if size > 0 then
    sum = 0;
    for i = 0 until i >= size
        sum = sum + alldata[i];
    next i
    sentdata[0] = sum;
! Root node transmits aggregated data to the base station.
    uart(sentdata);
! Clear alldata for next round of samples.
```

```
    bclear(alldata);
  end if
! Root node puts its own sample to alldata.
  alldata[] = data[0];
else
! Other nodes send their data to root node by broadcasting.
  bcast(data);
end if
```

Broadcast Handler:

```
! Buffer for storing data received over radio
buffer received;
! Buffer for storing data collected from all the nodes
buffer alldata;
! Private variables used for computation
private i;
private size;
! Root node retrieves data received over radio and put them in buffer alldata
if id() = 0 then
  bclear(received);
  received = bcastbuf();
  size = bsize(received);
! Add received data to alldata.
  for i = 0 until i >= size
    alldata[] = received[i];
  next i
end if
```

Reboot Handler:

```
! Set timer 0 period and start timer 0.
settimer0(20); ! Timer0 fires every 2 seconds.
```

5 Programming Environment

The TinyTemplate programming environment allows you to write and test TinyTemplate programs. As shown in Figure 2, the programming environment consists of several windows. The **Event Handler** buttons indicate which handler is selected for editing or injecting. The **Handler Version** indicates the version of the current handler. It automatically increases when a handler is injected. You can ignore it in this test. 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

program 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. TinyTemplate provides a programming template. Every time when TinyTemplate is started, the program template is shown in the **Program Text** window. You can modify it to create application of your own.

To run the template program, inject the **Broadcast** handler, the **Timer0** handler, and finally the **Reboot** handler. You should see outputs similar to that in Figure 3. The **Network** window shows the sample data on each node. 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 can check the correctness by adding the samples during one epoch in the **Network** window and compare the results with the data received in the **Base station** window. For example, during the first epoch, the sampled light levels are 94, 659, 1021, and 840. Adding them up yields 2614. This result matches with the first result received at the base station. To examine the returned data, you may want to stop the network from running the program so the screen stops scrolling. To do this, modify the argument for the **settimer0** function in the **Reboot** handler to 0 and inject the **Reboot** handler.

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 4 will pop up when you try to inject your handler. You need to click the **OK** button in the window 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 5 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 6. 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.

To write an application of your own, you can follow these steps.

1. Select the handler you want to edit by checking the corresponding **Event Handler** button. (Usually you can start with the **Timer0** handler to specify what each node does when its timer fires, then edit the **Broadcast** handler to specify what should be exectued when a node received a message over radio, and finally the**Reboot** handler to set the timer and start the timer.)

2. Modify the template code for that handler in the **Program Text** field to implement your own application.

3. Inject the handler by clicking the **Inject** button. If an error message window pops up, you need to fix the error in your handler and reinject it until no more errors are reported.

4. Select another handler and repeat the above steps.

5. Watch the results in the **Network** window and the **Base station** window to check the correctness of your code.

Figure 2: Programming environment of TinyTemplate.

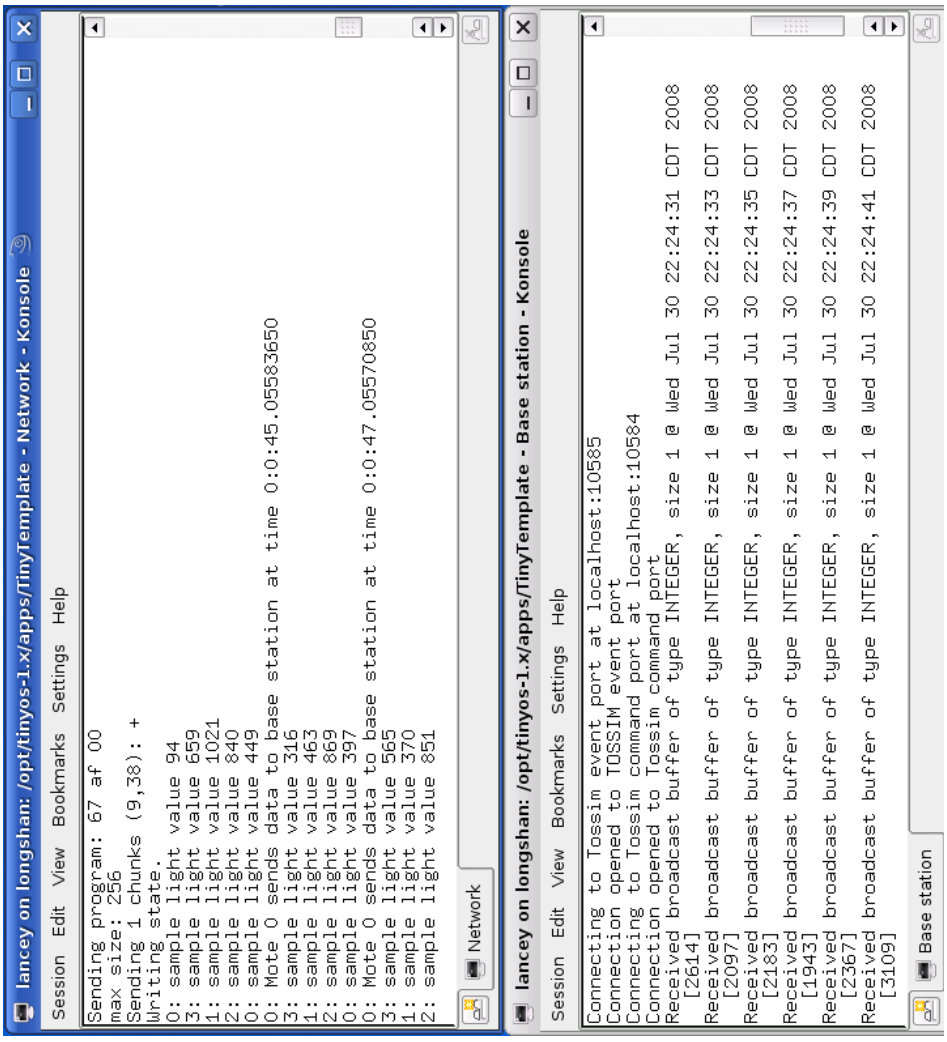

Figure 3: Results of the template program.

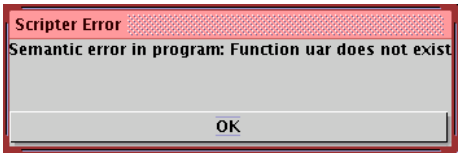

Figure 4: Undefined function error.

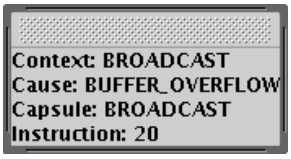

Figure 5: Buffer overflow error.

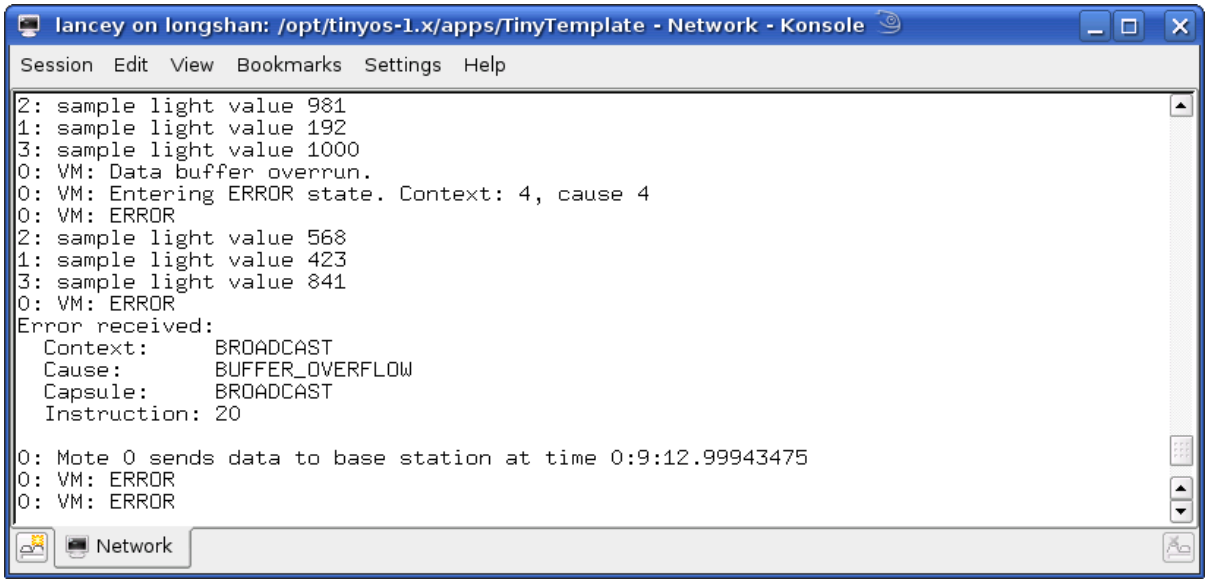

Figure 6: Buffer overflow message in network window.