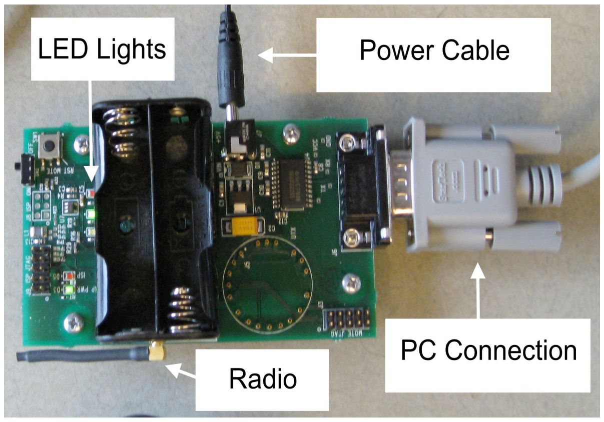
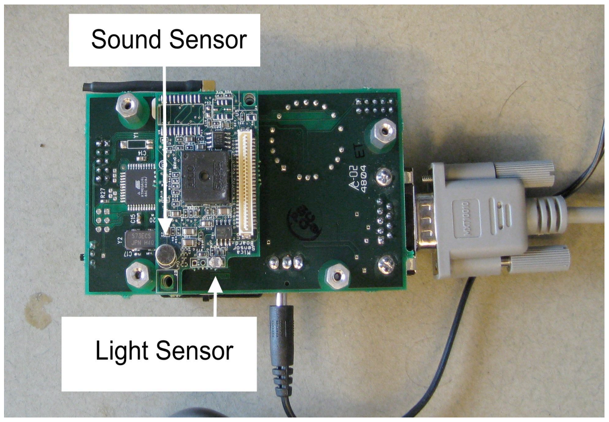
TinyScript Tutorial

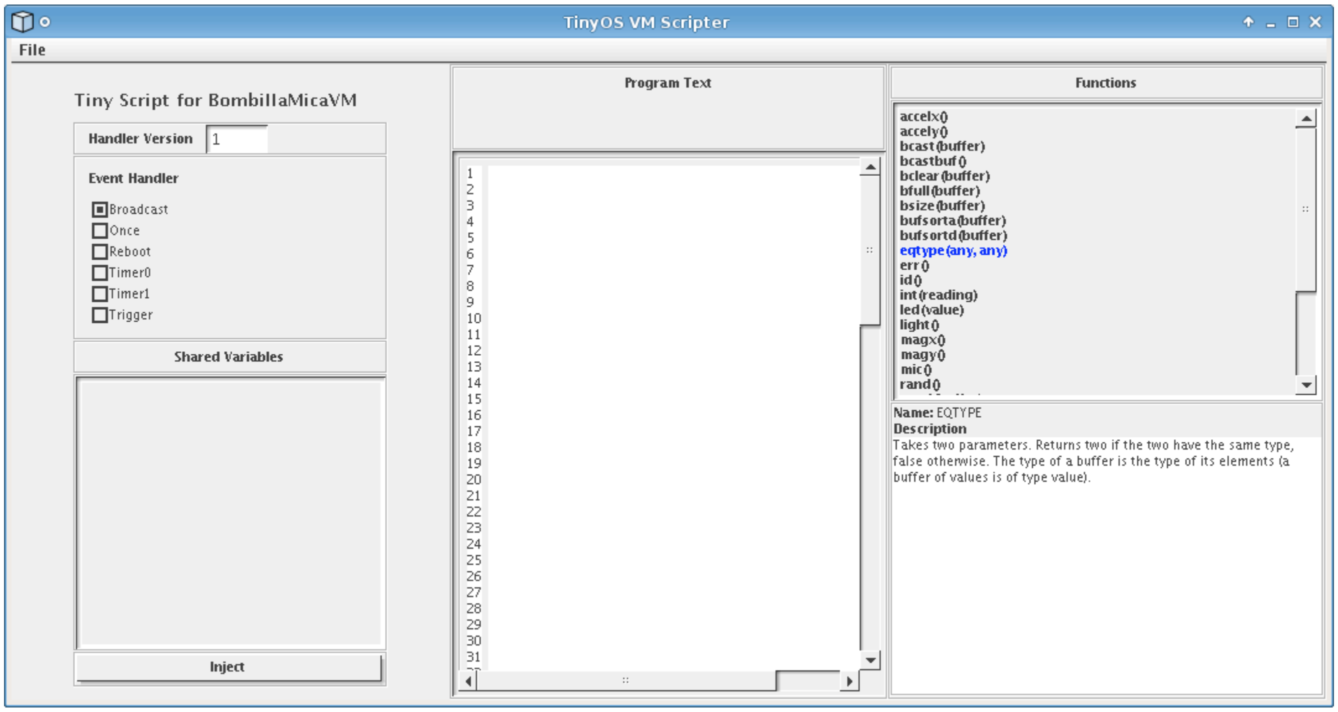
Introduction

In this exercise, you will be programming in the TinyScript programming language, a language designed for small sensors, known as “motes”. Below is an image of the Mote you will be using. On the top side of the mote, there are three lights colored red, yellow and green. On the bottom are addition to sensors for detecting light and sound. Fixed to the side of the mote is an antenna the motes can use to communicate with one another. Motes such as these are typically used by researchers studying a phenomena that can be detected via the built in sensors, such as detecting the presence of people or animals via the microphone.





Programming for the first time can be very challenging, and while TinyScript is designed to make writing your first programs as easy as possible some concepts might not be immediately clear. You will be able to make a note of these instances at the end of this tutorial.



Handler Version: 1

Event Handler: [✓ Broadcast]

Name: EQTYPE
Description
Takes two parameters. Returns two if the two have the same type, false otherwise. The type of a buffer is the type of its elements (a buffer of values is of type value).

The programming environment.

The environment you will be using allows you to write and quickly test TinyScript programs and consists of three windows. The programming environment (shown on the previous page) consists of several panels. Programs are entered in the “Program Text” field and are sent to the mote by clicking on the “Inject” button. In TinyScript, your programs are broken into pieces and executed according to a number of events. An event is said to “fire” when the criteria for that event is met. For example, the “Once” event is fired when the code associated with the “Once” handler is injected into the mote (causing that code to execute exactly once). The programs you write in TinyScript are executed when these events are fired. When you write code in the “Program Text” field, it is executed when the event selected in the “Event Handler” list is fired. Specific events will be talked about in greater detail later in the tutorial. For now, select the “Once” event handler by clicking the gray box next to the word “Once”, enter the following program into the “Program Text” field and click “Inject”:

```
led(1);
```

After clicking Inject, the red light on the mote should illuminate. If the light does not appear, double check that you have entered in the correct program. If you are still having issues, please ask your proctor for help. If everything the text appears then you have successfully written your first TinyScript program! Note that your program is only saved upon clicking the “Inject” button. If you enter code for a particular handler then choose another handler from the “Event Handler” list without first injecting that code, it will be lost.

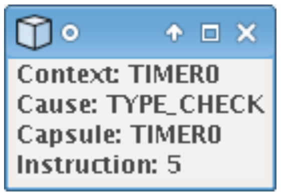
Throughout this tutorial, you are strongly encouraged to try all of the provided examples.

Comments
In TinyScript, it is possible designate regions of text in the program code as exempt from execution using the '!' character. A region of exempt code is referred to as a “comment” and is used to add a text description to a line or region of code. For example, the following line contains a comment explaining what the line does:

```
led(1); ! Turn on the red light
```

Adding a '!' will cause all of the remaining text of a line to be ignored. Throughout the tutorial, you will encounter code examples with comments designed to help you follow along.

Errors
While you are programming, you might come across an error such as this:



Context: TIMER0
Cause: TYPE_CHECK
Capsule: TIMER0
Instruction: 5

When the mote encouters an error, it will raise this window and begin flashing its LEDs. This indicates that an error occurred while your program was executing. There are a number of possible causes for an error such as this, so double check your code before injecting it again. Throughout this tutorial, a number of common error cases will be explained.

Functions

In the previous example, the line `led(1);` is referred to as a call to the `led` function. A function is simply a way to execute an operation, which may produce a value or side effect. For example, a function might read a value from one of the built in sensors, illuminate the mote's lights or communicate a value to another mote. The parenthesis following the name of the function indicate that the function is being called. Functions are called with zero or more arguments – the values between the parenthesis. The function operates based on the arguments given. In this case, the argument to the `led` function specifies which lights to illuminate. A list of functions and their descriptions can be found in the list within the “Functions” panel on the right side of the development environment. The following table highlights several functions and their effects:

Function	**Effect**
`id();`	Returns the unique identifier of this mote.
`rand();`	Returns a random value in the range -32768 to 32767.
`sleep(duration);`	Put the mote into a low-power state for `duration`, given in tenths of a second.
`led(value);`	Illuminate the LEDs according to the argument `value`.

The arguments to the led function are explained below:

Value	**Effect**
1	Turn on the red light.
2	Turn on the green light.
4	Turn on the yellow light.
0	Turn off all of the lights.

The `sleep` function is used to put the mote into a low power state for the given duration. When the mote is asleep, it uses substantially less power because it does not need to execute program code. Sensor applications can use this to reduce power consumption when it is not necessary to meet an immediate deadline as many applications can tolerate some delays in reading data.

Type the following code into the “Once” handler and press “Inject”:

```
led(1);
sleep(10);
led(0);
sleep(10);
led(1);
```

This should cause the red light to turn on for one second, turn off for another second and finally turn back on.

Events and Handlers

Programs in TinyScript are executed in response to set of events. For example, when the mote receives a message from another mote via its radio, the code associated with the "Broadcast" event is executed. The program code written to execute after a particular event is referred to as that event's handler. The handler executes exactly once each time the event fires. Later in the tutorial, we will discuss how two events can communicate and share data. A list of the available events and how they are triggered can be found below:

Event Name	When Triggered
Broadcast	When the mote receives a message from another mote.
Once	When the "once" handler is initially sent to the mote.
Reboot	Whenever the mote is reset.
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.
Trigger	When the `trigger` function is called.

Note that the last three events are invoked directly by the programmer. The programmer uses the function `settimer0` to set the frequency at which the Timer0 event is fired. The only argument to the function is a whole-number value specifying the period between firings of Timer0 in tenths of a second. The same holds for Timer1, instead using the `settimer1` function. Similarly, calling the `trigger` function causes the Trigger event to immediately fire once. This event can be used to perform operations common to multiple event handlers, eliminating duplicate code.

A More Complex Example

Periodic programs in TinyScript usually have two scripts: the first is what to do periodically, the second starts the periodic execution. Enter and inject the following script into the "Timer0" handler:

```
private reading;

reading = int(light());
led(reading / 128);
```

This code does several things: the first line declares a variable named reading (variables will be explained in the next section – for now just think of it as a named piece of data), the second line sets the value of reading to the current value of the light sensor and the last line illuminates the lights based on that value. This code causes different lights to illuminate based on the current light value; however, only injecting this code into the Mote will not cause the code to be executed. To make the "Timer0" event fire, you need to install a script that will start it. Edit the "Once" handler to be:

```
settimer0(10);
```

The lights should now illuminate based on the current light value sensed by the mote. Try moving the mote around and see if different lights illuminate.

Variables and Expressions

A variable in TinyScript is a location for storing data. In TinyScript programs, all variables must be declared before any program statements. For example:

```
! Define a shared variable named counter
shared counter;
! Increase its value by 1
counter = counter + 1;
```

TinyScript function and variable names are case insensitive: var is the same as VAR or Var. Names are composed of alphanumeric characters and the underscore, but the first character of an name must be a letter or the underscore. The following are all valid names:

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

The following are invalid names:

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

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`, it is a single variable that they share. If one handler changes it, the other handler will see the change.

TinyScript has two basic types of data: integers (a whole number in the range of -32768 to 32767) and sensor readings. Sensor readings are further divided into a type for each of the different kinds of sensor data (a microphone reading is distinct from a light reading). A type is merely a label assigned to a value saying what kind of data it holds and what you are allowed to do with that data. For example:

```
private val;
```

```
val = 1; ! val is an integer
val = light(); ! val is now a light reading
val = mic(); ! val is now a microphone reading
```

All data in TinyScript can be manipulated using a number of mathematical operators in what is known as an expression. TinyScript supports familiar operations including adding, subtracting, multiplying and dividing as well as several others.

Operator	**Effect**	**Example**	**Value of i**
+	Adds two numbers or variables	i = 1 + 2;	3
-	Subtracts two numbers or variables	i = 100 – 90;	10
*	Multiplies two numbers or variables	i = 5 * 3;	15
/	Divides two numbers of variables	i = 5 / 3;	1
%	Generates the remainder of dividing two numbers or variables	i = 5 % 3;	2

TinyScript Arithmetic Operators.

The possible operations that can be performed on a piece of data depend on its type. Integer data can be manipulated using any of the above operators. Sensor readings are immutable, meaning they cannot be manipulated via the operators. You cannot add an integer to a light reading, a light reading to a temperature reading, or even add two light readings. Attempting any of these will cause an error. If you want to process sensor readings, you must turn them into integers with the `int` function, passing in the sensor reading as an argument. For example:

```
private total;
private count;
private val;

val = light(); ! val is now a light reading
val = int(light()); ! val is now an integer
total = total + val; ! total is an integer
count = count + 1; ! count is an integer
val = total/count; ! val is now the average of readings as an integer
```

The evaluation of an expression proceeds algebraically and parenthesis pairs can be added to define the order that the expression is executed or to improve readability:

```
private i;
i = (5 + 2 * 2); ! i is now 9
i = (5 + 2) * 2; ! i is now 14
i = ((((5)))); ! i is now 5
```

Finally, buffer variables are used to house a fixed number of either sensor readings or integers and are always shared. Just like private and shared scalars, buffers are typed; a buffer can only contain values of a single type. A buffer's contents and type can be cleared with the `bclear` function. If a buffer has no type, it takes the type of the first value put into it.

Buffer values can be accessed and assigned by indexing into the buffer using and integer enclosed by two angle brackets. The index value starts at zero, so to access the first value of a buffer named `buf`, you would enter `buf[0]`. If you do not include a value within the angle brackets, the last value of the buffer is assumed. For example:

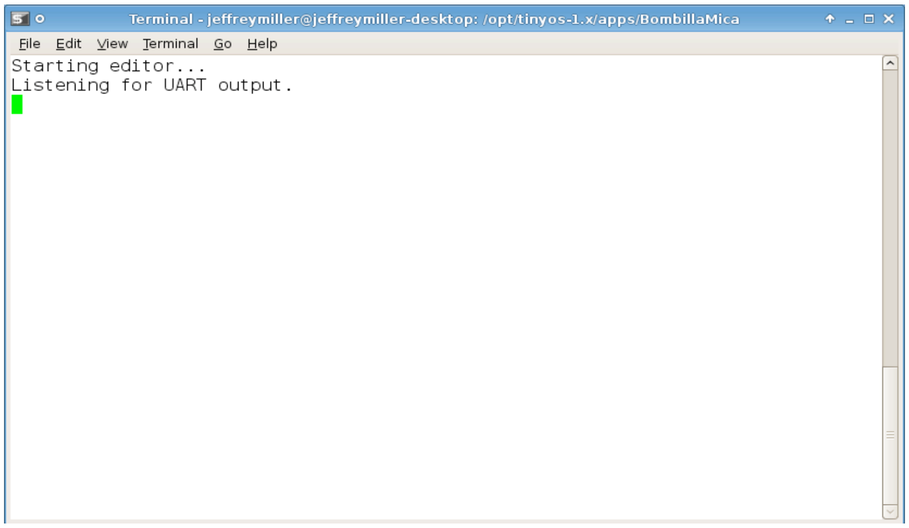
```
buffer bufOne;

bclear(bufOne); ! clear bufferOne
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 zero
bufOne[3] = bufOne[4] ! Put the value of index 4 into index 3
bufOne[5] = int(light); ! BufOne can only contain integers
```

Buffers have a fixed maximum size of ten values. Trying to add more than ten items will cause an 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. A listing of all of the buffer functions can be found in the following table:

Name	**Effect**
`bclear(buffer);`	Clear `buffer` so that it is empty and has no type.
`bfull(buffer);`	Returns `true` if buffer is full, false otherwise.
`bsize(buffer);`	Returns the number of elements in `buffer` as an integer.
`bufsorta(buffer);`	Sorts the elements in `buffer` in ascending order.
`bufsortd(buffer);`	Sorts the elements in `buffer` in descending order.

TinyScript Buffer Functions.

Communication
Fundamental to the operation of a wireless sensor is some modality for communicating data, either to other motes or to a collection point. TinyScript provides three methods for communication, one requiring a wired connection to the mote and two wireless. The wired method is accessed through the `uart` function. `uart` takes a buffer as an argument and displays that value in the UART output window, shown on the next page. This window should be open on the computer you are using. Please ask your proctor for help if you cannot locate it.


Terminal - jeffreymiller@jeffreymiller-desktop: /opt/tinyos-1.x/apps/BombillaMica
File Edit View Terminal Go Help

```
Starting editor...
Listening for UART output.
```

For example, enter the following program into the “Once” handler and click “Inject”:

```
buffer data;

data[] = 1;
data[] = 2;
data[] = 3;

uart(data);
```

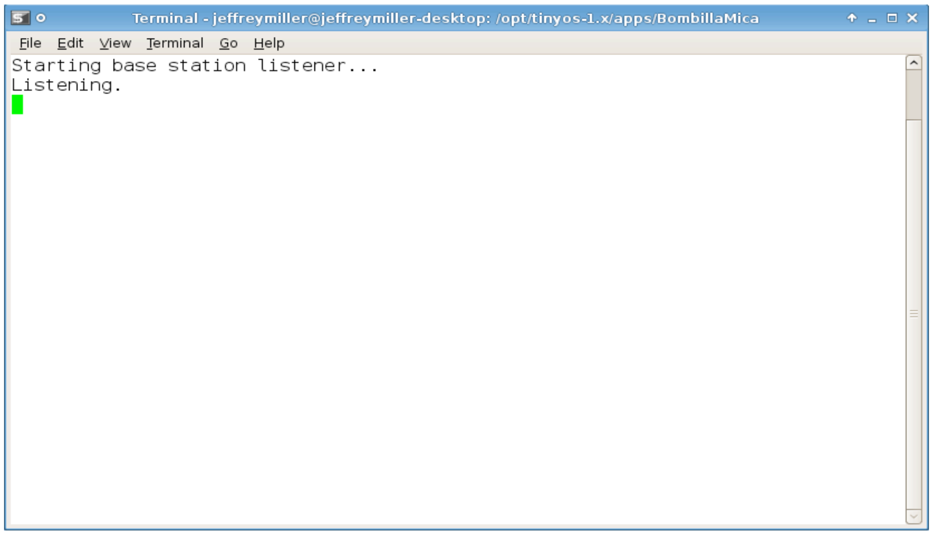
This should produces this output in the UART window:

```
Received UART buffer of type INTEGER, size 3 @ Tue Jul 08 09:43:42 CDT 2008
  [1][2][3]
```

The output produced by `uart` is simply a dump of the contents of the transferred buffer.

The motes are also equipped with radios that allow for communication with other sensors. Wireless communication is accomplished via two built in functions: `send` and `broadcast`. Both functions take a buffer argument containing the data to be sent. The two functions differ in the way they deliver data to their destination. `broadcast` is the more primitive function, transferring the contents of the buffer to all motes within radio range, typically 10-50 meters. `broadcast` does not guarantee that the data will make it to any particular mote. When a mote receives a message the broadcast event is fired. Code in the broadcast event handler can access the sent data by calling the `bcastbuf` function, which returns a buffer containing the sent data.

`send` differs from broadcast by sending data directly to a base station mote, which can be used to log or process the data received. In our setup, send will produces output in the following window:


Terminal - jeffreymiller@jeffreymiller-desktop: /opt/tinyos-1.x/apps/BombillaMica
File Edit View Terminal Go Help

```
Starting base station listener...
Listening.
```

The output produced by send is the same as uart. Try entering and injecting the following program into the "Once" handler:

```
buffer data;

data[] = rand();
data[] = rand();
data[] = rand();

send(data);
```

You should see three random values in the base station window.

Reading Sensors

The sensors built into the mote can be accessed using special functions that return the current value of a given sensor. A listing of the available sensor values and the functions used to access them can be found in the following table:

Function	**Sensor**
light();	Light
mic();	Microphone (sound)

Enter and inject the following program into the "Once" handler:

```
buffer value;
value[] = light();
uart(value);
```

The program reads a value from the light sensor and sends it to the UART output. Notice the value that the output produces. Sensor readings will generally range from 0 to about 1000, with higher values meaning that sensor is reading a higher level (e.g., higher numbers indicate brighter ambient light when using the light sensor).

Logic

An important part of any computer program is the ability to perform different actions based on certain conditions. For example, a web-based email service will only let you see your inbox if you enter the correct password, otherwise it might ask you to reenter your password. At the heart of this is a simple operation comparing two password values that reflects much of the logic used by a programmer to control his or her program.

In TinyScript, this kind of control is handled by the if-then statement and is the first statement we have encountered that allows us to control the flow of our program. Type and inject the following program into the “Once” handler.

```
private i;

i = 30;
if i > 20 then
     led(2);
else
     led(1);
end if
```

If the logical expression `i > 20` resolves to true, then the line `led(2);` executes. If the expression resolves to false, then `led(1);` executes. As you might expect, in this simple example `led(2);` (illuminating the green light) will always be executed because `i` will always have the value 30. A logical expression can be composed of the arithmetic operators on page 7 and the following comparison and logic operators:

Symbol	Operation on a (operator) b
>	True if a is greater than b, false otherwise
<	True if a is less than b, false otherwise
>=	True if a is greater than or equal to b, false otherwise
<=	True if a is less than or equal to b, false otherwise
=	True if a equals b, false otherwise

TinyScript Comparison Operators

Name	**Operation on a (operator) b**	**Example**
and	True if both a and b are true, false otherwise	val = a and b;
or	True if either a or b are true, false otherwise	val = a or b;
not	(Not only functions on one argument) True if argument is false, false otherwise	val = not a;

TinyScript Logic Operators

if-then statements can also be nested within each other, as in the following example, which turns on the red and green lights depending on the light and microphone values, respectively:

```
if int(light()) > 200 then
     led(1);
     if int(mic()) > 10 then
          led(2);
     end if
end if
```

Control Structures
TinyScript provides a way for you to repeat pieces of code through the `for` construct. This is referred to as a loop in computer programming as it is a means for specifying that a single block of code be executed multiple times, perhaps with one or more variables changing values each time the block of code is called. There are two basic forms, unconditional and conditional. Unconditional (for-to) loops run a specific number of times and terminate when the loop variable takes a specific value. Conditional (for-until) loops run until an arbitrary condition becomes true.

For example, try the following for-to loop sums all values from 1 to 10 by injecting it into the “Once” handler:

```
private sum;
private i;
buffer buf;

sum = 0;
for i = 1 to 11
     sum = sum + i;
next i
buf[] = sum;
uart(buf);
```

This should display the sum of 1 through 10 on the uart output window.

Note that the value of `i` is never 11; the loop terminates when the value of the variable being looped over is greater than or equal to the target value. The `next` keyword defines the end of the loop block, and increments the loop variable. It is possible to adjust the amount by which the variable is changed with each iteration using the step keyword. The next example calculates the sum of only the odd values from 1 to 10:

```
private sum;
private i;
buffer buf;

sum = 0;
for i = 1 to 10 step 2
     sum = sum + i;
next i

buf[] = sum;
uart(buf);
```

The following example illustrates the use of a for-until loop. This loop will put the values 2,4,6...20 in the buffer (when it has ten values, it will be full):

```
private i;
buffer buf;
bclear(buf);

for i = 1 until i > 10
     buf[] = i * 2;
next i

uart(buf);
```

A loop that will only execute while a logical condition is true 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

uart(buf);
```

In this example, the variable i is never modified and the loop continues to execute until `bfull(buf)` becomes false. Note, however, that `bfull` is only executed after the loop code is first called.