

Maté: Safe and Rapid Sensor Network Scripting

Philip Levis
pal@cs.berkeley.edu

Version 1.0
March 22, 2004

Contents

1	Introduction	3
2	Programming with Maté	3
3	End Users: TinyScript	4
3.1	Scripter	4
3.2	Script Structure	5
3.3	Variables	6
3.4	Functions	8
3.5	Arithmetic, Logic, and Conditionals	9
3.6	Control Structures	10
3.7	Error Conditions	12
3.8	Event Handlers	12
3.9	Parallelism	13
4	Domain Experts: Building an ALR	14
5	Expert Users: Extending Maté	14
5.1	Library Functions	14
5.2	Synchronization	16
5.2.1	Opcode Component Naming Convention	18
5.3	Events	18

1 Introduction

Programming sensor networks is difficult. They are large scale, distributed systems with limited resources, lossy communication, whose testing and debugging is made especially difficult by their being embedded in uncontrolled environments. Often, it is hard to know exactly what a certain algorithm or application will do without deploying it; changes that seem minor can greatly change system behavior. Deployment, however, is a time consuming and arduous process.

Sensor networks have a wide range of users, most of whom are not expert programmers. However, these users need to use sensor networks in their applications, such as habitat or structural monitoring, and therefore need be able to program. This programming cannot involve many low-level details of the system, or require expert knowledge of the system hardware or a great deal of programming experience.

Maté is a system designed to provide an accessible programming interface to sensor network end users. The core of Maté is a tiny VM that runs on top of TinyOS, a sensor network operating system. The bytecode interpreter can be customized for specific applications or deployments. End users write programs as simple, high-level scripts, which compile to the VM instruction set. These programs then self-propagate through a network. Reprogramming only requires introducing a single copy of a new program. This copy will then install itself across the entire network.

This document describes Maté and how to use it. Its purpose is to serve as an introduction to the system for users. It is divided into four major sections, with increasing degrees of technical detail. Section 3 describes the TinyScript language, and how one uses the Scripter tool to inject programs into a Maté network. This section is intended for end users, and requires no knowledge of TinyOS. Section 4 describes how to build a Maté VM and scripting environment. As programs will be constrained by decisions made during this process, it is intended for users who are knowledgeable about the targeted application domain. Finally, Section 5 describes how to extend Maté VMs by adding new function primitives and execution events by writing TinyOS components in nesC.

2 Programming with Maté

There are two basic parts to programming a Maté network. The first is a Maté VM installed on the network's motes. The second is a Java UI for writing and injecting programs into the network. Maté nesC code (for VMs) can be found in `tos/lib/VM` while the Java tool code can be found in `tools/java/net/tinyos/script`.

Section 4 describes how users can build customized VMs; to start, however, you may

want to use Bombilla, a simple example VM. Bombilla can be found in `apps/Bombilla`. To get started, go to `apps/Bombilla` and compile for your desired platform (e.g., `make pc` or `make mica2`). This will build a TinyOS VM as well as a scripting environment.

If you're using motes, then you should install the VM on a few nodes with `make reinstall.x` where `x` is the moteID for each node. You shouldn't use a GenericBase as a programming point for a Maté network. For example, if you're using a serial port to send data, you should plug a Maté node into the programming board. To efficiently program a network, you need to be able to reliably install new programs on a single node (e.g., over the serial cable). If you use a GenericBase, it's possible that some fragments of programs may be lost, preventing the network from reprogramming.

Once you've set up your programming point, start the Scripter from the Bombilla directory (it must be from this directory, to correctly find all of the VM-specific configuration files), specifying the communication port. For example,

```
java net.tinyos.script.Scripter -comm tossim-serial
```

starts a Scripter to talk to TOSSIM through a virtual serial port to mote 0. You can then write programs and inject them into a network.

3 End Users: TinyScript

Maté has an event-driven execution model. Every Maté VM has a set of events that trigger execution, such as receiving a packet, or a timer firing. When one of these events occur, it triggers the appropriate handler script to run. Handlers are written in a simple BASIC-like language called TinyScript.

3.1 Scripter

Scripter is a Java tool that compiles TinyScript programs to the instruction set of a Maté VM. As Maté VMs are customizable, programs must be compiled for a specific VM, and will only run properly on that VM. Compiling the same program to two different VMs will probably result in different binary code for each.

Figure 1 shows a screenshot of the current Scripter GUI. Scripter **must be run from the VM application directory** and has an optional communication argument:

```
java net.tinyos.script.Scripter [-comm source] <vm-description>
```

The `comm` parameter tells the Scripter where to send code packets, whether it be to TOSSIM, a serial port, or a SerialForwarder.

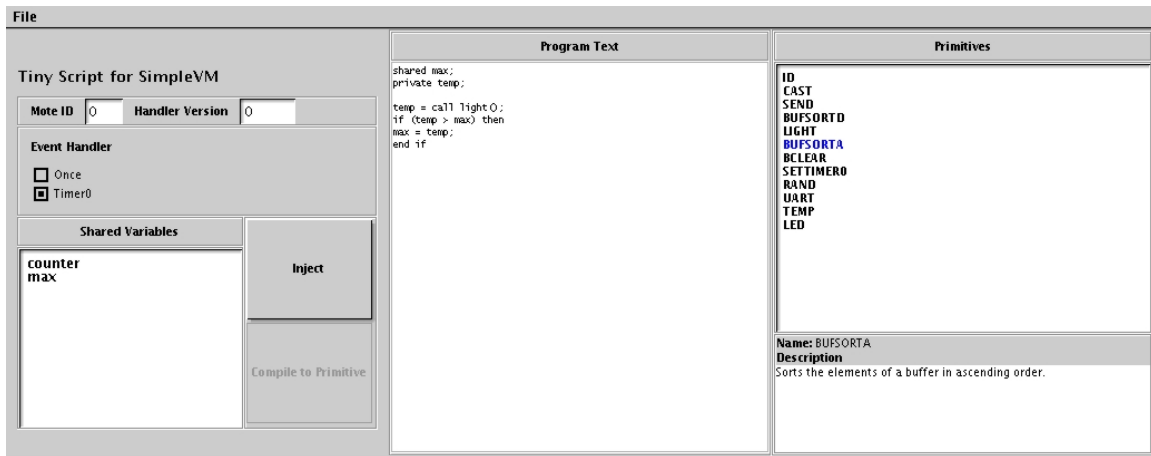


Figure 1: The Scripter Interface

The Scripter window is divided into three parts. The left subwindow is where you select what event handler you want to install, and the version number. Motes ignore handlers with older version numbers than what they currently have. Normally, the Scripter automatically handles these; each successful compilation of a given handler increments its version number. However, you can manually change it if need be.

The left subwindow also keeps a tally of all currently used shared variables (these are discussed in Section 3.3). When a shared variable is no longer used by any handler, the Scripter deallocates it.

The center window is where you write TinyScript, which the rest of this section describes. The right window provides the list of available primitives (library functions). Clicking on one of the primitives brings up a brief description of its use in the lower text area.

To install a new event handler, you choose which event you wish to write for, write its code in the Program Text window and press the Inject button. TinyScript programs have a maximum size (approximately 240 bytes). Writing programs that are too long will cause a compile-time error when the Scripter tries to inject them into a network.

If you quit Scripter using the File menu, then the interface saves its current state for later use. That is, it keeps track of the current handler code, versions, and shared variables; you can quit and restart the interface, resuming where you left off.

3.2 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;
```

There can only be one statement per line, and 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.

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.

TinyScript programs are for the most part case-sensitive. Keywords exist as both their uppercase and lowercase versions: `shared` can also be written `SHARED`, but cannot be written `sHarEd`.

3.3 Variables

TinyScript programs have two basic variable types: values and buffers. In the above programs, the keyword `shared` declared the variables to be values. Values can also be declared with the keyword `private`: Section 3.9 describes the distinction between the two. The keyword `buffer` declares a data buffer. Values represent a single data item, such as an integer or a sensor reading. Buffers are small collections of values.

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 = call random();
sensor = call light();

```

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

Types constrain how values can be modified, and how buffers can be accessed. There is one basic 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. Additionally, allowing transformation raises problematic questions: what is the type of a light reading added to a humidity reading?

To modify sensor readings, they must be cast to an integer. For example, this program computes an exponentially weighted moving average of the light sensor:

```

shared sensor;
shared aggregate;

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

```

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

call 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. 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;

call bsorta(aggBuffer);      ! Sort buffer entries in ascending order
size = call 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[] = call light(); ! Append a new light value to the buffer
```

3.4 Functions

The above code examples used several functions, such as `light()`, `bsorta()`, and `cast()`. Programs invoke functions using the `call` keyword. 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 a dynamically typed language, these types are not checked at compile time. For example, `bclear()` takes a single parameter, a buffer. Passing it an integer will cause an error.

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

```
call rand();
```

is a valid program.

Every Maté VM has a set of functions it provides as primitives (they are built into the VM). Customizing this set of functions is one of the ways that Maté programming environments can be tailored to specific deployments or applications (Section 4 describes

Name	Operator	Example	Name	Operator	Example
Addition	+	val = val + 2;	Less than	<	val = val < 2;
Subtraction	-	val = a - b;	Greater than	>	val = a > b;
Division	/	val = call bsize() / 2;	Less than or equal	<=	val = call bsize() <= 8;
Multiplication	*	val = 2 * b;	Greater than or equal	>=	val = b >= 2;
Exponentiation	^	val = val ^ 2	Not equal	<>	cond = val <> b

(a) Arithmetic Operations

(b) Comparison Operations

Figure 2: TinyScript Computational Primitives

the details of how to do so). A Maté scripting environment should provide information on what functions are available, as well as what they do, their required parameters, and return values.

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

```
buffer aggBuf;
shared val;

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

Note that assigning to a buffer is very different than assigning to an element of a buffer.

```
buffer aggBuf;
buffer aggBuf2
shared val;

aggBuf2 = aggBuf;
aggBuf[] = val;
```

Currently, TinyScript does not support user-written functions.

3.5 Arithmetic, Logic, and Conditionals

Figure 2(a) shows the set of arithmetic operations TinyScript provides, as well as their syntax.

All of these operations have a shorthand (similar to the C language) for self-assignment. For example, `val *= 2;` is the same as `val = val * 2;`.

TinyScript also supports logical operations, which are show in Figure 3(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 3 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 secondbit;
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 3: TinyScript Logical Primitives

The logical operations manipulate integer bit fields. Instead of manipulating the integer as a single 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 2(b). They resolve to one if true, zero if false.

3.6 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 call bfull(buf) then
  idle = 0;
  if call rand() & 1 then
    call send(buf);
    call bclear(buf);
  end if
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
    call leds(i & 7)
next i

```

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

```

private i;
buffer buf;

call 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 call bfull(buf)  
  buf[] = call rand();  
next i
```

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

```
private i;  
buffer buf;  
  
call 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
```

3.7 Error Conditions

Program errors can be divided into two classes: compile-time and run-time. Compile-time errors cause the compiler to fail; they are often language mistakes, such as passing too few parameters to a function. Run-time errors are detected as a program runs on a Maté VM, can cause the VM to stop execution and enter an error condition. The error condition involves flashing all of the mote LEDS simultaneously, and broadcasting an error message, alternating over the radio and over the UART. Examples of run-time errors include type checks (e.g., adding a sensor reading to an integer) and buffer overflow (trying to put more in a buffer than it can hold).

Once a mote has entered an error condition, the only way to return it to operation is to reprogram it; hopefully, the new program will not have the error.

3.8 Event Handlers

All Maté scripts run in response to an event. A given Maté VM has a static set of events that trigger execution. To program a Maté network, one writes scripts for these event handlers. Some events, such as Timer, are of general use. Timer executes periodically (e.g., every second). Other events have very narrow and specific uses. The SendR event, for example, is

triggered by the `sendr()` function and allows a program to modify packet headers, which is useful if implementing an ad-hoc routing protocol.

The Once event is unusual; it only runs once, when new code is installed for it. This allows users to manage a network more easily than with periodic events. For example, one can write a Once handler that changes a mote's radio transmit power; this need not run many times, and doing so would be wasteful.

3.9 Parallelism

Maté VMs can run multiple event handlers can run simultaneously. When it installs new code, the runtime determines which handlers can run concurrently in a safe manner, and disallows parallelism that could corrupt data.

Handlers can declare two kinds of variables: `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, handlers can pass data to one another. Buffers are implicitly shared variables.

For example, imagine a situation in which you want a program to measure the maximum time that elapses between receiving packets. One way to accomplish this is to use two handlers, Receive and Timer, with two shared variables, `current` and `max`. Whenever Timer fires, it increments `current`. When Receive triggers, it tests to see if `current` is greater than `max`; if so, it sets `max` to `current` and resets `current` to zero.

Timer:	Receive:
<code>shared current;</code>	<code>shared current;</code>
<code>shared max</code>	
	<code>current = 0;</code>
<code>current = current + 1;</code>	
<code>if (current > max) then</code>	
<code>max = current;</code>	
<code>end if</code>	

In this program, an error could occur if Timer and Receive run at the same time. Specifically, if the Receive statement `current = 0;` runs just after `if (current > max) then` but before `max = current`, then `max` could be set to zero. This is clearly incorrect.

This form of program error is commonly known as a race condition, as the result of the computation is dependent on the race between the two handlers to modify their variables.

In Maté programs, these sorts of race conditions cannot occur. When a VM installs a program, it analyzes the code to determine all of the variables the program uses. It doesn't allow two handlers that could have race conditions to run in parallel. In the above example, Timer and Receive would not run at the same time; if one were running, the other would

wait. However, if two handlers can run safely at the same time, the VM lets them do so. This is useful if one of the handlers has, say, a long loop. If, for example, the Receive handler had to wait for this long loop to complete, then multiple packets may arrive, causing the handler to drop some of them.

4 Domain Experts: Building an ALR

Maté ALRs can be built in two ways: with a GUI, and with a file. The VM building tool is VMBuilder:

```
java net.tinyos.script.VMBuilder [options]
options:
  -t=<directory>      Load all contexts and primitives from subdirectories contexts and opcodes
  -d=<directory>      Load all contexts and primitives from directory
  -l=<language file>   Use the language specified in this file
  -f=<file>            Load context or opcode from file
  -nw <vmfile>        Do not start the GUI; load VM specification from file
```

There are sample VM specification files in `lib/VM/samples`, as well as a README describing the file format. For now, we suggest you use build VMs from files. However, if you want to browse the available contexts and library functions, then run VMBuilder with `-t=<dir>`, where `dir` is the path to `lib/VM`.

5 Expert Users: Extending Maté

Maté has a small set of events and library functions, but it is also designed to allow users to add new ones. In this section, we step through the process of adding an event or library function to Maté so you can incorporate it into your ALR.

5.1 Library Functions

For VMBuilder to recognize a library function, there must be an Opcode Description File (.odf). The library function component should be a configuration, which wires the actual module to all of its needed services. We'll use `OPid` as a running example. All library functions provide the `MateBytecode` interface, which is how the VM engine executes them.

`OPid` provides the `id()` library call, which pushes the mote ID onto the operand stack. There are three files: `OPidM.nc`, `OPid.nc`, and `OPid.odf`. `OidM` is a module, that actually implements the library function:

```
includes Mate;
```

```

module OPidM {
  provides interface MateBytecode;
  uses {
    interface MateStacks as Stacks;
  }
}

implementation {

  command result_t MateBytecode.execute(uint8_t instr,
                                         MateContext* context) {
    dbg(DBG_USR1, "VM (%i): Pushing local address: %i.\n", (int)context->which, (int)TOS_LOCAL_ADDRESS)
    call Stacks.pushValue(context, TOS_LOCAL_ADDRESS);
    return SUCCESS;
  }
}

```

It just takes TOS_LOCAL_ADDRESS and pushed it onto the operand stack, where it can be used as a parameter to another function, stored in a variable, or modified with arithmetic.

OPid.nc is a configuration that wires OPidM to all of its needed services:

```

includes Mate;

configuration OPid {
  provides interface MateBytecode;
}

implementation {
  components OPidM, BStacksProxy;

  MateBytecode = OPidM;
  OPidM.Stacks -> BStacksProxy;
}

```

This approach means that, merely by incorporating OPid in the component list of the VM, it will be wired to everything it needs. The Proxy components are a way of having many opcodes share an implementation of an interface without specifying what it is. The top-level VM wires all of the Proxy components to specific implementations. This means, for example, that the implementation of the operand stacks can be changed without needing to change any opcodes. There is a Proxy for each major interface. The full set is:

- BBufferProxy
- BContextSynchProxy
- BErrorProxy
- BLocksProxy
- BQueueProxy

- BStacksProxy
- BVirusProxy

The final file is the description. ODF files have a single XML-like element in them, which describes the function.

```
<PRIMITIVE name=ID opcode=id numparams=0 returnval=TRUE desc="Returns the mote's ID.">
```

They have five required elements: name, opcode, numparams, returnval, and desc. Name defines the function name exported to TinyScript. Opcode defines the name the primitive opcode takes to the assembler. These must both be the same as the component name (e.g., “id”); name should be all upper case, and opcode should be all lower case. The numparams element states how many parameters the function takes: id takes none. The returnval element specifies whether the primitive returns a value. The id() function, for example, does, while the send() function does not. Finally, the desc element is the description provided in the Scripter interface. It should describe what parameters the function expects, what it does, and whether it has a return value.

The best way to learn how to write a new function is to look at a few of the existing ones.

5.2 Synchronization

Some instructions represent shared resources. For example, `bpush1` and `getvar4` access shared variables. For race free program, the VM execution engine must be aware of this and control scheduling appropriately.

All of the VM context synchronization is handled in the `BContextSynch` component. It keeps track of shared resources through the `MateBytecodeLock` interface. If an instruction manages a shared resource, then it must provide this interface. Additionally, in its ODF, it must have the optional element “locks” set to true.

For example, let’s look at `OPbpush1`. This instruction pushes the two shared buffers, `buffer0` and `buffer1`, onto the operand stack. It is not a library function; instead, it is an element of the instruction set that TinyScript compiles to. In `tscript.ldf`, it lists

```
<PRIMITIVE opcode="bpush1" locks=true>
```

Then, in `OPbpush1M`:

```
module OPbpush1M {
  provides interface MateBytecode;
  provides interface MateBytecodeLock;
}
```


MateBytecodeLock has a single command:

```
interface MateBytecodeLock {
    command int16_t lockNum(uint8_t instr);
}
```

This takes a instruction opcode and returns a unique lock number. The idea is that certain opcodes have a lock associated with them. `bpush1`, for example, has one bit of embedded operand, for the two buffers. If `bpush1 0` is passed to `OPbpush1M.nc`, then it returns the lock number for buffer zero, while `bpush1 1` will return the lock number for buffer one.

The full `OPbpush1M.nc` logic:

```
module OPbpush1M {
    ...
    provides interface MateBytecodeLock;
    ...
}

implementation {
    typedef enum {
        BOMB_BUF_LOCK_1_0 = unique("MateLock"),
        BOMB_BUF_LOCK_1_1 = unique("MateLock"),
    } BufLockNames;
    ...
    command int16_t MateBytecodeLock.lockNum(uint8_t instr) {
        if (instr & 1) {
            return BOMB_BUF_LOCK_1_1;
        }
        else {
            return BOMB_BUF_LOCK_1_0;
        }
    }
    ...
}
```

It declares two unique lock numbers with the `nesC` `unique` function. When `lockNum()` is called, it returns the lock number associated with the corresponding buffer. Then, each context has

```
uint8_t heldSet[(BOMB_HEAPSIZE + 7) / 8];
uint8_t releaseSet[(BOMB_HEAPSIZE + 7) / 8];
uint8_t acquireSet[(BOMB_HEAPSIZE + 7) / 8];
```

where `BOMB_HEAPSIZE` is defined to be `uniqueCount("MateLock")`;

5.2.1 Opcode Component Naming Convention

Library function components have the following naming convention:

OP<width><name><operand>.nc

Width and operand are both numbers. Width specifies how many bytes wide the instruction is. If no width is specified, the default is one. Operand specifies how many bytes of embedded operand there are. If no operand is specified, the default is zero. Note that, after considering width and operand, the instruction must have an opcode in its first byte. That is, the instruction cannot be two bytes wide and have no embedded operand; as the Maté scheduler dispatches on the first byte of the opcode, it would not be able to distinguish this instruction from other ones.

Here are a few examples:

Component	Width	Name	Embedded	Description
OPrand	1	rand	0	Generates a random number
OPpushc6	1	pushc	6	Push a constant onto the stack
OP2jumps10	2	jumps	10	Jump to a 10-bit address

Generally, library functions are always one byte wide and never have embedded operands. In the above example, neither `pushc` nor `jumps` are available as library functions; they are actually opcodes that compose part of what TinyScript compiles to. So, all library functions you write should just be of the form `OP< name >`.

The library function component must be a configuration, which wires the actual module to all of its needed services. We'll use `OPrand` as a running example.

The *name* field must be the name of the function as it is called from a scripting language.

5.3 Events

All Maté event components use the interfaces `MateContextSynch`, `MateEngineControl`, `MateCapsuleStore`, and `MateAnalysis`. We describe each in turn.

MateContextSynch: This interface is how the handler component communicates with the Maté scheduler. The only event the component must implement is `makeRunnable()`.

MateEngineControl: This interface is how the handler component interacts with the main execution engine. Using it, a context registers its execution capsule and can ask the VM to reboot (necessary when new code is installed).

MateCapsuleStore: This interface is how handler components obtain references to their current capsule. The component that implements `MateCapsuleStore` is responsible for managing code storage across different program dissemination formats. It signals a `capsuleChanged` event when new code is fully installed.

MateAnalysis: This interface is how the context requests a program analysis to compute lock sets for parallel execution.

The exact working and interactions of all of these interfaces are somewhat complex. If you want to write a new event handler, I recommend taking an existing, simple one (e.g., `ClockContext`), copying it, and modifying it as need be. Cleaning up this aspect of the VM is definitely a thing I plan to do, but not just yet.

Event handler contexts have component naming conventions similar to library functions. There is a configuration, which must be `<name>.Context.nc`, a module, and a context description file (CDF). In addition to a `CONTEXT` entry, a CDF can contain `PRIMITIVE` entries, which represent library functions the handler enables. For example, `Timer1Context.cdf` is so:

```
<CONTEXT name="Timer1" desc="A periodic timer.">
<PRIMITIVE name=SETTIMER1 opcode=settimer1 numparams=1 param1=1 returnval=FALSE
desc="Takes a single parameter, the interval (in tenths of a second) between
timer firings for the Timer 1 context. Calling with a time of zero stops the
timer.">
```

Note that `settimer1()` does not have an ODF file in `lib/opcodes`. In fact, the opcode module makes calls on `Timer1Context.nc`, to change the timer firing rate:

```
configuration OPsettimer1 {
  provides interface MateBytecode;
}

implementation {
  components OPsettimer1M, Timer1Context, MStacksProxy;

  MateBytecode = OPsettimer1M;
  OPsettimer1M.Stacks -> MStacksProxy;
  OPsettimer1M.Types -> MStacksProxy;
  OPsettimer1M.Timer -> Timer1Context;
}
```