A Quick User's Guide to Maté

Philip Levis

March 28, 2002

Introduction

This document is a simple overview of the Maté virtual machine for TinyOS. It is intended for people who want to write simple clock-driven Maté applications. Writing ad-hoc routing algorithms in Maté is beyond the scope of this document. This document explains how to put Maté on a mote, how to write Maté programs, how to upload those programs onto a Maté mote, and gives a brief description of some basic Maté instructions.

Maté Overview

Maté is a bytecode interpreter that runs on TinyOS. It presents a virtual stack-based architecture to programs. Code is contained in capsules, which are at most 24 instructions and fit in a single TinyOS packet. There are capsules that are run in response to an event (e.g. the clock timer) and capsules that can be called from other capsules (subroutines).

The complete Maté instruction set can be found in `nest/tos/include/tos-vm2.h`. The complete Maté source can be found in `nest/tos/system/VM2.{c,comp}`. Maté has functionality beyond what is described in this document. A Java Maté assembler is included in the current TinyOS release: `net.tinyos.asm.Assembler.java`. It also has a GUI interface that sends capsules out over the serial port: `net.tinyos.vmGUI.MainWindow`. `MainWindow` uses COM2 by default – changing this requires changing the source file. By connecting a `generic_base` mote to the serial port, you can send Maté capsules.

Maté Architecture and Execution Model

Maté is a two-stack architecture: each execution context has an operand and a return address stack. The former is used for data operations, while the latter is used to keep track of subroutine invocations. The operand stack has a maximum depth of 16, while the return address stack as a maximum depth of 8. There is also a single variable heap that persists across all executions. There are three types of variables: values, sensor readings, and messages.

All Maté code exists in capsules, which hold up to 24 instructions. Every instruction is a single byte long. Every capsule has a numeric ID; capsules 0-3 are the four subroutine capsules, while capsule 4 is the clock capsule that is called in response to a clock event. Every capsule also has a version number, which Maté uses to determine whether it should install the capsule or not.

Maté processes data by pushing operands onto its operand stack then executing instructions which consume them. For example, the program

```
pushc 1
pushc 2
add
halt
```

will result in a 3 being on the top of the operand stack. Besides pushing constants with `pushc`, sensor readings can be read with `sense` and the message buffer can be pushed onto the stack with `pushm`.

The clock context's operand stack persists across invocations. For example, if the above program were the clock capsule, then each invocation would push 3 onto the operand stack until it reached its maximum depth of 16, after which pushed values would be ignored. When a new clock capsule is installed by Maté, the clock operand stack is cleared and a single value variable of zero is pushed onto it – this provides an easy way to keep track of a counter. The clock context is also reset in this manner whenever a new subroutine capsule is installed.

If a clock event goes off while the context is still executing, then Maté clears the return address stack and forces the context to jump to the first instruction of the clock capsule. This prevents a context from entering an infinite loop. The rate at which clock events arrive can be manipulated with the `clockc` and `clockf` instructions.

Subroutine capsules are invoked with the `call` instruction, and should return with the `return` instruction. Call consumes a single value operand, which indicates which subroutine to call. If a subroutine issues the halt instruction the context running it halts.

Branches are made with the `blez` instruction, which branches if the operand on top of the stack (which is consumed) is less than or equal to zero. A message operand will always cause `blez` to branch. Like `pushc`, blez has an embedded operand stating which instruction (absolute, not relative) to jump to. For example

```
pushc 0
blez 3
pushc 1
halt
```

will never push 1 on the operand stack.

All capsules must have an explicit halt at their end.

Maté Capsules and Capsule Propagation

Maté capsules can be written with the `MainWindow` program. On the right is a small assembler window where programs can be entered. Pressing "Create" will generate a capsule with that program. Each capsule written is placed in the payload of a new AMPacket – one can flip through the generated capsules with the tabbed pane. The first two byes of the AM payload are the capsule's meta-information – it's capsule type (byte 0) and version (byte 1). Types 0-3 are the four subroutines, while type 4 is the clock capsule.

Maté will only install capsules that have a more recent version number than the capsule currently running. More recent is defined to either have a version number which is up to 64 greater or more than 128 less (for 8 bit wrap-around).

For example, if a Maté mote is running a clock capsule with version 23, then after writing a program and clicking the create button, one would have to type 4 in the first byte of the packet then 24 in the second before sending it – otherwise it will possibly be installed as the wrong type of capsule or ignored because of its version.

Sending capsules from a generic base is not very useful when there's a very wide area sensor network. Therefore, capsules can forward themselves. When a capsule issues the `forw` instruction, it broadcasts a packet containing itself; motes that hear it may then install the capsule. Capsules can also issue the `forwo` instruction to forward an arbitrary capsule – `forwo` consumes a value single operand off the stack to determine which capsule to forward (0-4). For example,

```
pushc 2
forwo
```

will forward subroutine capsule 2.

Maté Message Buffers

Maté assumes that an ad-hoc routing algorithm will use eight bytes of header in a 30-byte AM payload. Therefore, the message data buffer is 22 bytes long. One byte of this is used to store how many values are in the payload, and a second byte is used to give the lower 8 bits of the sender's ID (it would have been padding otherwise). The remaining 20 bytes contain up to 5 variables, each of which takes up 4 bytes. One byte specifies what kind of variable it is (value or sensor reading); if it's a sensor reading, the second byte indicates which type. The third and fourth bytes store a 16-bit data value.

Header0	Header1	Header2	Header3
Header4	Header5	Header6	Header7
Num	ID	Type0	Sensor Type0?
Data Value0		Type1	Sensor Type1?
Data Value1		Type2	Sensor Type2?
Data Value2		Type3	Sensor Type3?
Data Value3		Type4	Sensor Type4?
Data Value4			

Maté Instructions

The Maté instruction set has three classes of instructions: basic, s-class, and x-class. The s-class instructions are beyond the scope of this document – they only apply to ad-hoc networking. Basic instructions and x-class differ only in that x-class instructions (`blez` and `pushc`) have embedded operands. Basic instructions obtain all of their operands from the operand stack.

Sample Instruction

Effect: Brief description of instruction
Class: instruction class (basic or x-class)
Operands: Input operands – first operand is top of stack, next is 2nd on stack, etc. Type (value, sense, msg, any) is specified.
Results: Output operands – first operand is top of stack, etc. Type is specified.
Bytecode: Hexidecimal opcode
Description: Detailed description.

Production Instructions

pushc

Effect: Push a constant on the operand stack
Class: basic
Operands: none
Results: value
Bytecode: 0xC0-0xff
Description: When this instruction is issued, its embedded six bit operand is pushed onto the operand stack as an unsigned value variable.

sense

Effect: Read a sensor
Class: basic
Operands: value
Results: sensor
Bytecode: 0x0D
Description: The operand specifies which sensor. Currently, only two are supported: sensor 1 is light and sensor 2 is temperature.

id

Effect: Push mote's ID on the operand stack
Class: basic
Operands: none
Results: value
Bytecode: 0x09
Description:

rand

Effect: Push a 16-bit random value on the operand stack
Class: basic
Operands: none
Results: value
Bytecode: 0x1c
Description:

pushm

Effect: Push context's message buffer
Class: basic
Operands: none
Results: msg
Bytecode: 0x11
Description: The context has a single message buffer, which is placed on the operand stack with this instruction. Issuing this instruction multiple times will place the same buffer on the operand stack multiple times.

Data Instructions

and

Effect: Binary and
Class: basic
Operands: value1,value2,...
Results: value
Bytecode: 0x02
Description: Push (value1 & value2).

or

Effect: Binary or
Class: basic
Operands: value1,value2,...
Results: value
Bytecode: 0x03
Description: Push (value1 | value2).

shiftr

Effect: Bitshift variable right
Class: basic
Operands: value1,value2,...
Results: value
Bytecode: 0x04
Description: Shift the second operand right a number of bits specified by the first operand and push the result on the stack.

shiftl

Effect: Bitshift variable left
Class: basic
Operands: value1,value2,...
Results: value
Bytecode: 0x05
Description: Shift the second operand left a number of bits specified by the first operand and push the result on the stack.

add

Effect: Add two variables (polymorphic)
Class: basic
Operands: any,any,...
Results: any
Bytecode: 0x06
Description: The add instruction operates on all three types. Two values added will result in a value. A value added to a sensor will result in a sensor reading adjusted by the value. Two sensor readings of the same type can be added, but different types (e.g. light, temperature) cannot. Adding a value or a sensor reading to a message buffer will append the variable if there is space, do nothing if there is not, and result in a buffer. Adding two message buffers will append the variables of the second onto the first and result in a buffer.

inv

Effect: Invert top of operand stack
Class: basic
Operands: value
Results: value
Bytecode: 0x0a
Description: Push the operand multiplied by -1.

not

Effect: Binary not
Class: basic
Operands: value1,value2,...
Results: value
Bytecode: 0x16
Description: Push ~(value1).

Control Instructions

call

Effect: Call subroutine
Class: basic
Operands: value
Results: none
Bytecode: 0x1f
Description: Call subroutine specified by operand. Must be 0-4.

return

Effect: Return from subroutine
Class: basic
Operands: none
Results: none
Bytecode: 0x3f
Description: Return from subroutine to execute instruction after call.

halt

Effect: Halt execution
Class: basic
Operands: none
Results: none
Bytecode: 0x00
Description: Halt execution of the context.

blez

Effect: Branch if less than or equal to zero
Class: x-class
Operands: value
Results: none
Bytecode: 0x80-0xBF
Description: Branch to absolute instruction address specified by embedded operand if value on top of operand stack is less then or equal to zero.

Code Propagation Instructions

forw

Effect: Forward current capsule
Class: basic
Operands: none
Results: none
Bytecode: 0x00
Description: Broadcast the current capsule for others to possibly install.

forwo

Effect: Forward other capsule
Class: basic
Operands: value
Results: none
Bytecode: 0x00
Description: Broadcast capsule specified by operand for others to possibly install. Value operand must be 0-4.

Other Instructions

putled

Effect: Control LEDs
Class: basic
Operands: value
Results: none
Bytecode: 0x09
Description: `putled` takes a five bit operand. The bottom three bits specify the LEDS: red, yellow, and green. The fourth and fifth bits specify the operation to perform. 0 sets the LEDs; 1 turns the specified LEDs on (leaving the others alone); 2 turns the specified LEDS off (leaving the others alone); 3 toggles the specified LEDS. For example, 0x02 will turn on off the red and green LEDS and turn on the yellow, 0x19 will toggle the red led, and 0x13 will turn off the yellow and red leds while leaving the green one alone.

send

Effect: Send packet
Class: basic
Operands: msg
Results: none
Bytecode: 0x09
Description: Send the data contained in the message buffer with Maté's built-in ad-hoc routing component. By default, this is BLESS. You must therefore have a BLESS base station for packets to be sent (otherwise BLESS will not be able to find a route). You can redefine the ad-hoc routing algorithm used by changing the corresponding application `.desc` file.

clear

Effect: Clear variable
Class: basic
Operands: any
Results: operand
Bytecode: 0x13
Description: Clear the variable on top of the operand stack, leaving it there. This is especially useful for the message buffer, as you want to clear out variables that have been previously put in it, as it persists across invocations. Issuing clear on a sensor reading or a value will set it to be zero.

log

Effect: Write variable to non-volatile storage
Class: basic
Operands: any
Results: none
Bytecode: 0x09
Description: If the operand is a message buffer with 5 variables in it, only the first four are written. Maté buffers variables until it has enough for a full log line (4 variables), then writes it to EEPROM. Therefore, if a program logs three sensor values, they will not be written to EEPROM until a fourth variable is logged.

logr

Effect: Read log line into buffer
Class: basic
Operands: msg,value1,...
Results: msg
Bytecode: 0x09
Description: Read log line specified by value1 into the message buffer.

sets

Effect: Set shared variable
Class: basic
Operands: any
Results: none
Bytecode: 0x09
Description:

gets

Effect: Get shared variable
Class: basic
Operands: none
Results: any
Bytecode: 0x09
Description:

swap

Effect: Swap top two operands on stack
Class: basic
Operands: any1,any2,...
Results: any2,any1,...
Bytecode: 0x09
Description:

clockc

Effect: Set clock counter for capsule invocation
Class: basic
Operands: value
Results: none
Bytecode: 0x3f
Description: `clockc` sets how many clock ticks must occur before the clock context is triggered. For example, if the clock tick frequency is once per second and the clock counter is 4, the clock capsule will run every 4 seconds. If the clock tick frequency were changed to eight per second (with `clockf`), the capsule would start running twice per second.

clockf

Effect: Set clock tick frequency in $\frac{1}{32}$s of seconds
Class: basic
Operands: value
Results: none
Bytecode: 0x3e
Description: `clockf` sets how often the Maté clock ticks. The number of ticks that must occur for the clock context to run is specified by `clockc`. The operand for `clockf` must be within the range 1-255, and represents the frequency of the clock ticks in thirty-seconds of a second. For example, `clockf` with a parameter of 160 will make the Maté clock tick once every five seconds.

Sample Maté Programs

`cnt`$_t o_l eds$

Clock capsule:

```
pushc 1
add      # Add 1 to counter on top of stack
copy
pushc 7
and      # Take bottom 3 bits of copy
putled   # Set LEDs to bits
halt
```

`cnt`$_t o_l eds, whereincrementisinasubroutine$

Clock capsule:

```
pushc 0
call
copy
pushc 7
and      # Take bottom 3 bits of copy
putled   # Set LEDs to bits
halt
```

Subroutine 0:

```
pushc 1
add
copy
```

`sens`$_t o_l eds$

Clock capsule:

```
pushc 1
sense         # Read light value
push 7
shiftr        # Only want top 3 bits of reading
putled
halt
```

Sending a Sensor Reading in a Packet

Clock capsule:

```
pushc 1
sense
pushm
clear        # Clear out the message buffer
add          # Put reading in message buffer
send         # Send buffer
halt
```

Saving 4 readings on operand stack, averaging them, sending

Clock capsule:

```
pushc 1
sense       # Push a light reading on top of operand stack
swap        # Keep counter on top of operand stack
pushc 1
add         # Increment counter
copy
inv
pushc 4
add
blez 11     # If counter >= 4, jump past halt
halt
pop         # Jump-to point; pop counter
add
add
add         # Add up 4 readings
pushc 2
shiftr      # Divide by 4 (bitshift 2 right
pushm
clear       # Clear out a message buffer
add         # Put averaged reading in buffer
send        # Send packet
pushc 0     # Reset counter to 0
halt
```