Targeting PowerTOSSIM for the SensorCubes and Online Energy Management Schemes

by
Alexandru Emilian Şuşu
Andrea Acquaviva
David Attienza

TOSSIM and PowerTOSSIM

TOSSIM ([TOSSIM-Sensys03, TOSSIM-Manual]) is the original simulator for TinyOS. We use it in order to assess the behavior of MAC and routing protocols, and TinyOS applications.

The simulator can be used for prototyping networks with many sensor nodes (e.g.,one can prototype networks with more nodes than the available number of physical nodes). Thus the simulator can be used to assess the scalability of WSN applications.

Another advantage of TOSSIM is that it is able to run TinyOS applications, without any need to change them. Also, TOSSIM does emulation of the code, i.e. the simulation code is executed natively on a PC. These are advantages to many of the other existing emulators/simulators for Wireless sensor networks – see [SIM] for simulator survey.

TOSSIM was originally written for the mica platform which has a 40kbit RFM radio transceiver.

In the [TOSSIM-Manual] paper they present in chapter 3 the "Radio Models". Here are some extracts:

- "collision is modeled as a logical or; there is no cancellation. This means that distance does not effect signal strength; if mote B is very close to mote A, it cannot cut through the signal from far-away mote C. This makes interference in TOSSIM generally worse than expected real world behavior."
- there are 2 radio models:
  - simple where we have no signal loss and
  - lossy where we describe the topology using a graph where we have edges between nodes x and y if y can hear x. Edges have probabilities of flipping the bits - there is a Java tool LossyBuilder that builds loss rates based on the physical topology.
- "The lossy model models interference and corruption, but it does not model noise; if no mote transmits, every mote will hear a perfectly clear channel."

The radio model is implemented in tinyos-1.x\tos\platform\pc\CC1000Radio\rfm_model.c . There it is read the file with the loss rates (default filename "lossy.nss") that is normally generated by the Java tool LossyBuilder.

PowerTOSSIM ([PowerTOSSIM]) is an extension of TOSSIM that adds energy consumption estimation for the Mica2 platform. The Mica2 platform is composed of:

- 7.3 MHz ATmega128L processor
- 128KB of code memory, 512KB EEPROM
- 4KB of data memory
- ChipCon CC1000 radio capable of transmitting at 38.4 Kbps with an outdoor transmission range of approximately 300m
  - to use the CC1000 radio stack (which includes the BMAC protocol [BMAC]) you have to write in the Makefile of the application you want to compile the following line: –PFLAGS += -I%T/platform/pc/CC1000Radio (see http://www.eecs.harvard.edu/~shnayder/ptossim/install.html for all details)

PowerTOSSIM is included in TOSSIM version 1.1.9 and beyond. This means basically that if you download an up-to-date version of TinyOS you will have TOSSIM with its power extension, PowerTOSSIM. For example, we use TinyOS 1.1.13 (the latest stable version, that we installed with CygWin) which contains **almost** the latest version of PowerTOSSIM - I compared with the latest version of PowerTOSSIM from CVS.

It is important to mention that PowerTOSSIM as distributed with TinyOS only generates log messages for the various important events for a wireless mote (e.g., mote has radio on at this CPU cycle, radio starts transmitting, the sensor is on, etc). Basically, during the emulation with PowerTOSSIM there is no status of the battery energy level, of how much energy is consumed in a CPU cycle, etc. This means that PowerTOSSIM does not allow to build online power management/energy-aware strategies.

In order to infer the energy consumption out of the log messages generated by running the emulation/compiled-simulation one has to run the offline postprocessing tool postprocess.py Python script on the resulting traces.

Simple Example of Using PowerTOSSIM

Recompile your app:

```
$ make pc
```

Make sure DBG includes POWER. If you don't need any other debugging messages, this reduces to (for the appropriate shell):

```
bash$ export DBG=power
tcsh% setenv DBG power
```

Run main.exe with the -p flag and save the output to a file. For example

```
./build/pc/main.exe -t=60 -p 10 > myapp.trace
```

The trace will contain log message as these:

```
SIM: Random seed is 812500
0: POWER: Mote 0 ADC ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE ON at 645564
0: POWER: Mote 0 RADIO_STATE TX at 672964
0: POWER: Mote 0 RADIO_STATE RX at 724964
```

In order to infer the energy consumption out of the log messages generated by running the emulation/compiled-simulation one has to run the postprocess.py Python script on the resulting trace:

```
$TOSROOT/tools/scripts/PowerTOSSIM/postprocess.py -sb=0 --em
$TOSROOT/tools/scripts/PowerTOSSIM/mica2_energy_model.txt myapp.trace
```

The -sb parameter specifies whether to assume that the motes have a sensor board attached. The -em parameter specifies the energy model. Run postprocess.py -help for details on other options.

By default, the postprocessor prints the total energy used by each component on each mote, like the following output:

```
Mote 0, cpu total: 143.495863
Mote 0, radio total: 2579.446498
Mote 0, adc total: 93.272311
Mote 0, leds total: 0.000000
Mote 0, sensor total: 0.000000
Mote 0, eeprom total: 0.000000
Mote 0, cpu_cycle total: 0.000000
Mote 0, Total energy: 2816.214672
```

The -detail flag can be used to generate current vs time data files, one per simulated mote

$ /opt/tinyos-1.x/tools/scripts/PowerTOSSIM/postprocess.py -help

```
USAGE: postprocess.py [-help] [-debug] [-nosummary][-detail[=basename]]
[-maxmotes N][-simple] [-sb={0|1}] --em file trace_file
    -help: print this help message
    -debug: turn on debugging output
    -nosummary: avoid printing the summary to stdout
    -detail[=basename]: for each mote, print a list of `time\tcurrent' pairs to the
file basename$moteid.dat (default basename='mote')
    -em file: use the energy model in file
    -sb={0|1}: Whether the motes have a sensor board or not. (default: 0)
    -maxmotes: The maximum of number of motes to support. 1000 by default
    -simple: Use a simple output format, suitable for machine parsing
```

By default, PowerTOSSIM uses the energy model from energy_model.txt in the current directory.

TinyVIZ

An example application that can be used to be run on the TOSSIM platform is the TinyVIZ application (see http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson5.html for details)

Note: If you receive the following error message when running the TinyVIZ:

java.lang.ClassNotFoundException: net.tinyos.message.avrmote.TOSMsg

then the solution is described here:
www.cs.virginia.edu/~jwang/STIL_files/Implementation_documents/nescProgrammingNotes.doc

Note: Details about how to build the Surge application: http://www.tinyos.net/tinyos-1.x/doc/multihop/multihop_routing.html.

Enhancing PowerTOSSIM with Online Power Consumption

For this I got inspired from how the log messages are treated in order to infer the energy consumption. This is done in the Python script $TOSROOT\tools\scripts\PowerTOSSIM\postprocess.py that posprocesses the log data obtained from running the TOSSIM-targetted application. I have changed the following .nc files:

Reverse engineering the postprocess.py script:

The postprocess.py script does basically the following:

- keeps track for every mote the time and the state of the components of the previous event

Files that were changed/added:

- powermod.h
- PowerState_Alex_include.h
- Nido.nc
- PowerStateM.nc

Check for "Alex" comments to see where I have added new code.

Basically what I've done was:

- added all the constants defined in the energy model file (tinyos-1.x\tools\scripts\PowerTOSSIM\sensorcube_energy_model.txt) in the PowerState_Alex_include.h header file.
- for each mote, defined variables (see powermod.h):
  - batteryEnergy
  - radioState, radioCrt, totalRadioEng
  - CPUState, CPUCrt, totalCPUEng
  - PrevCPUCycles – to keep track how many CPU cycles have elapsed from the last log message to the current one
  - variables for "plugging in" the emulation the energy received from a scavenger: scavengerCrt, etc
- in nido.nc added:
  - in nido_start_mote() - the initialization of the various variables mentioned above
  - in main() - a last update of the total energy variables
- in PowerStateM.nc added immediately after generating log messages for the events regarding the radio, and CPU components the computation of the new energy levels (see for functions MACRO_UpdateTotalRadioEnergy and MACRO_UpdateTotalCPUEnergy).

Targeting PowerTOSSIM for the SensorCubes

This task is basically composed of several sub-tasks:

- changing the values of current consumption of the important components. For this we got the current consumption values from Tom Torfs and:
  - we have created another energy model file (SensorCube_energy_model.txt) that is used by postprocess.py
  - used the values directly in our modified version of PowerTOSSIM, as we have already explained.
- changing the way the simulation is performed due to the behavioral differences between the mica/mica2 motes and the SensorCubes. We focused here only on the following components, which are the most power consuming:
  - CPU – instead of the Atmega128L microcontroller we have to model the TI MSP430 microcontroller
  - Radio – instead of the CC1000 or RFM we have to model the nRF2401 transceiver

We attempted to create a very good modeling of the TI MSP430 microcontroller by trying to compile the C file generated from the NesC file using the GCC compiler for the TI MSP430. This is so because the machine code for the Atmega128L microcontroller will have a different size than the TI MSP430 microcontroller, thus resulting in different number of executed cycles.
In the end, we agreed with Bert (also Victor Shnayder) not to care about this issue, since the CPU is anyway not the most power consuming component. It would be good to pursue further this issue, when we have the time. We also faced some technical issues, for example:

- The reason I can't run the script "compile.pl" (that instruments the NesC application with basic block counters to provide a good estimate of the energy consumed by the CPU) is that on Windows the C:\tinyos\cygwin\opt\tinyos-1.x\tools\scripts\PowerTOSSIM\cilly.asm.exe is not a Win32 executable (but ELF32, strange enough). So I tried to recompile the executable as explained in http://www2.uic.edu/~tcanli1/cpucyclecountingcode.htm, but I wasn't able to do it because the compilation of CIL is requiring libcurses and I wasn't able to find libcurses for CygWin (although I found libncurses...).
- Then I tried to run "compile.pl" on Linux (where I already had installed tinyos, avr-gcc), since here the ELF-executable should have worked, but I still receive errors - now in the .s file generated from the C file.

We focused on the modeling of the radio.

CC1000 Radio Abstraction (in PowerTOSSIM and for the real implementation - Mica2)

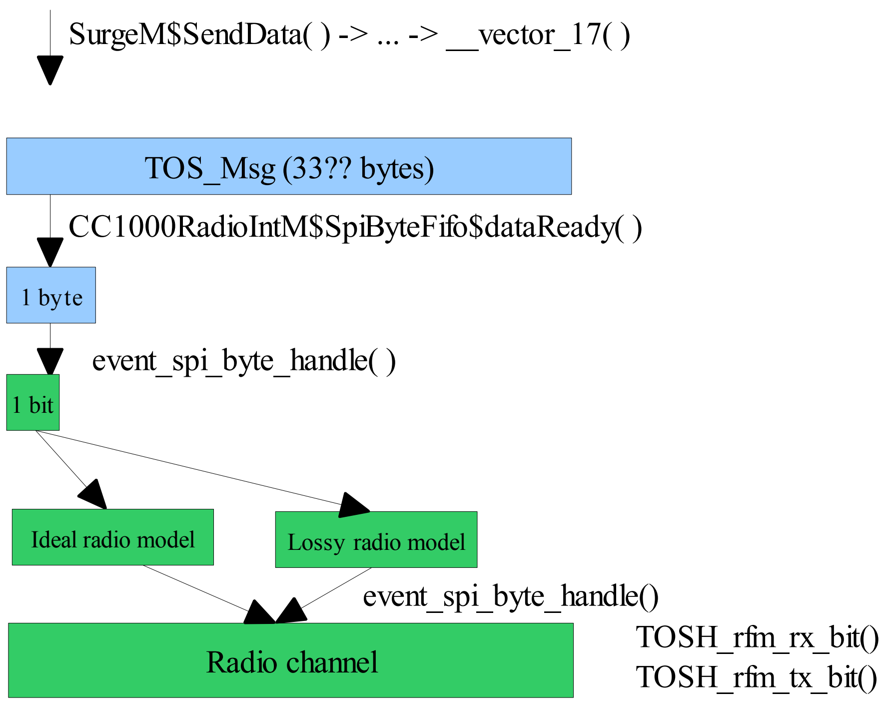
To understand how is modeled the radio in PowerTOSSIM and for the real Mica2 we reversed engineered the source code of the radio stack.

The best way to understand the source code is to understand the generated C file by the NesC compiler: for example, if you build the SurgeImec application, you get the apps\SurgeImec\build\pc\app.c source file. This file is preprocessed and thus it is self-contained - there is no reference to external header/source files (Note: to compile the SurgeImec application with CC1000 you have to add in the makefile the following line: PFLAGS += -I%T/platform/pc/CC1000Radio) .

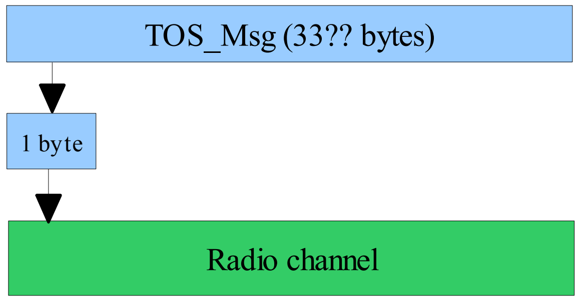
Sending a packet with CC1000

The difference in abstraction between the radio stack implemented on the real Mica2 platform and in PowerTOSSIM is the following.

For the radio stack implemented for the PowerTOSSIM we have the following flow of a TOS_Msg packet that is sent:



For the radio stack implemented for the real Mica2 platform we have the following flow of a TOS_Msg packet that is sent (the packet is transmitted by the controller to a memory location read and written by the transceiver – see function __vector_17()):



CC1000RadioIntM$txbufptr of type TOS_MsgPtr found in:

- CC1000RadioIntM$Send$send
- void CC1000RadioIntM$PacketSent(void)
- CC1000RadioIntM$SpiByteFifo$dataReady

The message is of type TOS_MsgPtr= * TOS_Msg. TOS_Msg is a struct of size 33 bytes.

```
typedef struct TOS_Msg {
         uint16_t addr;
         uint16_t s_addr;
         uint8_t type;
         uint8_t group;
         uint8_t length;
         uint8_t seqNo;
         int8_t data[18];

         uint16_t crc;
         uint16_t strength;
         uint8_t ack;
         uint16_t time;
       } TOS_Msg;
```

The SurgeMsg has the following definition:

```
typedef struct SurgeMsg {
  uint8_t type;
  uint16_t reading;
  uint16_t parentaddr;
} __attribute((packed))  SurgeMsg;
```

Let's take a look how SurgeImec is sending a packet - the "stream" of calls is the following:

```
SurgeM$SendData() -> SurgeM$Send$send() -> MultiHopEngineM$Send$send() ->
MultiHopEngineM$SendMsg$send() -> QueuedSendM$SerialSendMsg$sendDone() ->
QueuedSendM$QueueServiceTask() -> QueuedSendM$SerialSendMsg$send() ->
AMPromiscuous$SendMsg$send() -> AMPromiscuous$sendTask() -> AMPromiscuous$RadioSend$send()
-> CC1000RadioIntM$Send$send() -> CC1000RadioIntM$SpiByteFifo$enableIntr() ->
HPLSpiM$SpiByteFifo$enableIntr -> sets a call to __vector_17 ->
HPLSpiM$SpiByteFifo$dataReady() -> CC1000RadioIntM$SpiByteFifo$dataReady() ->
CC1000RadioIntM$PacketSent ->  CC1000RadioIntM$Send$sendDone ->
AMPromiscuous$RadioSend$sendDone -> AMPromiscuous$reportSendDone ->
QueuedSendM$SerialSendMsg$sendDone -> CC1000RadioIntM$Send$send ->
CC1000RadioIntM$Send$sendDone -> ...
```

Also we have the following calls from `CC1000RadioIntM$SpiByteFifo$dataReady()` -> `CC1000RadioIntM$SpiByteFifo$writeByte() -> HPLSpiM$SpiByteFifo$writeByte()`.

From HPLSpiM$SpiByteFifo$writeByte() we don't have calls, but we have data flow:

data --(see HPLSpiM$SpiByteFifo$writeByte())--> HPLSpiM$OutgoingByte[tos_state.current_node] --(see SIG_SPI_signal())--> HPLSpiM$spdr[tos_state.current_node] --(see SIG_SPI_signal())--> temp

SIG_SPI_signal() -> HPLSpiM$SpiByteFifo$dataReady() -> CC1000RadioIntM$SpiByteFifo$dataReady

In event_spi_byte_create() we assign event_spi_byte_handle() to fevent->handle. This is used in queue_handle_next_event().

queue_handle_next_event() --indirectly_through_function_pointer--> event_spi_byte_handle() -> SIG_SPI_signal(void) and TOSH_rfm_tx_bit() -> **lossy_transmit()** (or **simple_transmit()** - it depends on the type of Radio model used - see function main() that parses the command line parameters to choose simple or lossy model).

Basically what happens during this long tree of calls is that the packet of data gets split into bytes in CC1000RadioIntM$Send$send(TOS_MsgPtr pMsg) (actually function CC1000RadioIntM$SpiByteFifo$enableIntr() calls HPLSpiM$SpiByteFifo$enableIntr() which calls indirectly HPLSpiM$SpiByteFifo$dataReady()) and then every byte gets split into bits (see event_spi_byte_handle()) and then every bit gets sent - see TOSH_rfm_tx_bit() and lossy_transmit().

CC1000RadioIntM$SpiByteFifo$dataReady is an important function. It is written in an FSM-like style. If it has to send a packet it sends it byte by byte, calling the CC1000RadioIntM$SpiByteFifo$writeByte() function, and after sending the last byte it calls CC1000RadioIntM$PacketSent.

When a byte is sent through CC1000RadioIntM$SpiByteFifo$writeByte() then, because we have SIG_SPI_signal() that is called periodically, indirectly by queue_handle_next_event() we ensure the data flow of the current byte. Also queue_handle_next_event() calls indirectly event_spi_byte_handle() that splits the byte into bits.

For the sake of completeness you can find all the definitions of the functions that I mentioned in Appendix A.

<u>Receiving a packet with CC1000</u>

Depending on the radio model used:

- create_simple_model() does model->hears = simple_hears
- create_lossy_model() does model->hears = lossy_hears

event_spi_byte_handle() -> TOSH_rfm_rx_bit() -> hears()

The Nordic nRF2401 Transceiver

One fundamental difference between the CC1000 and the nRF2401 transceiver is that nRF2401 uses the ShockBurst buffering technique **when transmitting**. The limit of the ShockBurst frame length is 256 bits. [Master, Chapter 2.1.3]

In the app.c source code obtained from compiling the SurgeImec application with the nRF2401 ALOHA stack we can find the following “stream” of calls responsible for sending a packet:

SurgeM$SendData() -> **SurgeM$Send$send()** -> **MultiHopEngineM$Send$send()** -> MultiHopEngineM$SendMsg$send() -> QueuedSendM$SerialSendMsg$sendDone() -> QueuedSendM$QueueServiceTask() -> QueuedSendM$SerialSendMsg$send() -> AMPromiscuous$SendMsg$send() -> AMPromiscuous$sendTask() -> AMPromiscuous$RadioSend$send() -> **nRF2401RadioM$Send$send()** -> **nRF2401RadioM$MacTableTx$processOutPacket()**

Here it is not clear anymore where the stream of calls get. Because we have an interrupt (event) type of implementation that mimics well the FSM design of the MAC protocol (timers for duty-cycling and backoff period are used to send data) I got lost in the stream of function calls. Anyway, I assume the stream of calls will reach at a certain point here:

nRF2401RadioM$retransmission_control(void) -> n**RF2401RadioM$SendToSPI(void).** From here we take 2 call branches:

- the 1st branch cares about sending to the Shockburst buffer of nRF2401 every byte of data of the packet that needs to be sent:
  - nRF2401RadioM$SpiByte$write() -> HPLSpiM$SpiByte$write() -> HPLSpiM$USARTControl$tx() -> HPLUSART0M$USARTControl$tx() -> dataflow: HPLUSART0M$U0TXBUF = data -> ...
- the 2nd branch takes care after sending every byte to the buffer to actually send the data via wireless using the ShockBurst mode (basically the important part is function **TOSH_CLR_RF_CE_PIN()** that executes an ASM instruction that instructs nRF2401 to send the data in the buffer via wireless).
  - **nRF2401RadioM$nRF2401Control$SBurstSend(void)** -> **nRF2401ControlM$nRF2401Control$SBurstSend(void)** -> nRF2401ControlM$HPLNordic$SBurstSend(void) -> HPLnRF2401M$HPLnRF2401$SBurstSend() -> HPLnRF2401M$HPLnRF2401$SBurstSend(void) -> **TOSH_CLR_RF_CE_PIN()**

So we can see in practice the difference between the ShockBurst mode of nRF2401 and the unbuffered byte-by-byte transmission of CC1000.

For the sake of completeness you can find all the definitions of the functions that I mentioned in Appendix C.

For receiving a packet we have the following stream of calls:

sig_PORT2_VECTOR -> MSP430InterruptM$Port27$fired -> HPLnRF2401M$DataReady1$fired -> HPLnRF2401M$HPLnRF2401RxCH1$UveGotPckt -> nRF2401ControlM$HPLNordicRxCH1$UveGotPckt -> nRF2401ControlM$HPLNordicRxCH1$read_byte ->

HPLnRF2401M$HPLnRF2401RxCH1$read_byte -> TOSH_READ_RF_DATA_PIN
Also from nRF2401ControlM$HPLNordicRxCH1$UveGotPckt ->
nRF2401ControlM$nRF2401Control$UveGotPckt ->
nRF2401ControlM$nRF2401Control$PktRX -> nRF2401ControlM$nRF2401Control$PktRX ->
nRF2401RadioM$nRF2401Control$PktRX -> nRF2401RadioM$nRF2401Control$SelectChannel ->
nRF2401ControlM$nRF2401Control$SelectChannel and
nRF2401ControlM$nRF2401Control$RxMode->
nRF2401ControlM$HPLNordic$setRXShockBurstMode ->
HPLnRF2401M$HPLnRF2401$setRXShockBurstMode ->
HPLnRF2401M$HPLnRF2401$setRXShockBurstMode ->
HPLUSART0M$USARTControl$enableRxIntr ->

The most important function when reading a packet is HPLnRF2401M$HPLnRF2401RxCH1$read_byte that reads bit-by-bit the received byte.

Appendix A1

```
#line 41
static inline
# 74 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/mica2/HPLSpiM.nc"
result_t HPLSpiM$SpiByteFifo$enableIntr(void)
#line 74
{

  * (volatile unsigned char *)(unsigned int )& * (volatile unsigned char *)(0x0D + 0x20) =
0xC0;
  * (volatile unsigned char *)(unsigned int )& * (volatile unsigned char *)(0x17 + 0x20) &=
~(1 << 0);
  HPLSpiM$PowerManagement$adjustPower();
  return SUCCESS;
}

*********************

#line 67
static inline
# 286 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/mica2/CC1000RadioIntM.nc"
void CC1000RadioIntM$PacketSent(void)
#line 286
{
  TOS_MsgPtr pBuf;

#line 288
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 288
    {
      CC1000RadioIntM$txbufptr->time = 0;
      pBuf = CC1000RadioIntM$txbufptr;
    }
#line 291
    __nesc_atomic_end(__nesc_atomic); }
  CC1000RadioIntM$Send$sendDone((TOS_MsgPtr )pBuf, SUCCESS);
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 293
    CC1000RadioIntM$bTxBusy = FALSE;
#line 293
    __nesc_atomic_end(__nesc_atomic); }
}

# 67 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/BareSendMsg.nc"
inline static  result_t CC1000RadioIntM$Send$sendDone(TOS_MsgPtr arg_0xdb6f10, result_t
arg_0xdb7060){
#line 67
  unsigned char result;
#line 67

#line 67
  result = AMPromiscuous$RadioSend$sendDone(arg_0xdb6f10, arg_0xdb7060);
#line 67

#line 67
  return result;
#line 67
}

#line 80
static inline
# 239 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
result_t AMPromiscuous$RadioSend$sendDone(TOS_MsgPtr msg, result_t success)
```

```
#line 239
{
  return AMPromiscuous$reportSendDone(msg, success);
}

static
# 170 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
result_t AMPromiscuous$reportSendDone(TOS_MsgPtr msg, result_t success)
#line 170
{
  dbg(DBG_AM, "AM report send done for message to 0x%x, type %d.\n", msg->addr, msg->type);
  AMPromiscuous$state[tos_state.current_node] = FALSE;
  AMPromiscuous$SendMsg$sendDone(msg->type, msg, success);
  AMPromiscuous$sendDone();

  return SUCCESS;
}

# 49 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/SendMsg.nc"
inline static  result_t AMPromiscuous$SendMsg$sendDone(uint8_t arg_0xfc6e68, TOS_MsgPtr
arg_0xfc82f8, result_t arg_0xfc8448){
#line 49
  unsigned char result;
#line 49

#line 49
  result = QueuedSendM$SerialSendMsg$sendDone(arg_0xfc6e68, arg_0xfc82f8, arg_0xfc8448);
#line 49

#line 49
  return result;
#line 49
}


#line 81
static inline
# 188 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Queue/QueuedSendM.nc"
result_t QueuedSendM$SerialSendMsg$sendDone(uint8_t id, TOS_MsgPtr msg, result_t success)
#line 188
{
  if (msg !=
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].pMsg) {
      return FAIL;
    }


  if ((!QueuedSendM$retransmit[tos_state.current_node] || msg->ack != 0) ||
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].address == TOS_UART_ADDR) {

      QueuedSendM$QueueSendMsg$sendDone(id, msg, success);
      QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.curren
t_node]].pMsg = (void *)0;
      dbg(DBG_USR2, "qent %d dequeued.\n",
QueuedSendM$dequeue_next[tos_state.current_node]);
      QueuedSendM$dequeue_next[tos_state.current_node]++;
#line 199
      QueuedSendM$dequeue_next[tos_state.current_node] %= QueuedSendM$MESSAGE_QUEUE_SIZE;
    }
  else
```

```
    {
      QueuedSendM$Leds$redToggle();
      if (++
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].xmit_count > QueuedSendM$MAX_RETRANSMIT_COUNT) {


          QueuedSendM$QueueSendMsg$sendDone(id, msg, FAIL);
          QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.cu
rrent_node]].pMsg = (void *)0;
          QueuedSendM$dequeue_next[tos_state.current_node]++;
#line 216
          QueuedSendM$dequeue_next[tos_state.current_node] %=
QueuedSendM$MESSAGE_QUEUE_SIZE;
        }
    }


  TOS_post(QueuedSendM$QueueServiceTask);

  return SUCCESS;
}



***********************************

# 58 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/BareSendMsg.nc"
inline static  result_t AMPromiscuous$RadioSend$send(TOS_MsgPtr arg_0xdb69f8){
#line 58
  unsigned char result;
#line 58

#line 58
  result = CC1000RadioIntM$Send$send(arg_0xdb69f8);
#line 58

#line 58
  return result;
#line 58
}


#line 67
static inline
# 174 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/CC1000RadioIntM.nc"
void CC1000RadioIntM$PacketSent(void)
#line 174
{
  RadioMsgSentEvent ev;
  TOS_MsgPtr pBuf;

#line 177
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 177
    {
      CC1000RadioIntM$txbufptr[tos_state.current_node]->time = 0;
      pBuf = CC1000RadioIntM$txbufptr[tos_state.current_node];
    }
#line 180
    __nesc_atomic_end(__nesc_atomic); }

  nmemcpy(& ev.message, pBuf, sizeof  ev.message);
```

```
  ev.message.crc = 1;
  sendTossimEvent(tos_state.current_node, AM_RADIOMSGSENTEVENT, tos_state.tos_time, &ev);
  CC1000RadioIntM$Send$sendDone((TOS_MsgPtr )pBuf, SUCCESS);
  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 186
    CC1000RadioIntM$bTxBusy[tos_state.current_node] = FALSE;
#line 186
    __nesc_atomic_end(__nesc_atomic); }
}

# 67 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/BareSendMsg.nc"
inline static  result_t CC1000RadioIntM$Send$sendDone(TOS_MsgPtr arg_0xdb6f10, result_t
arg_0xdb7060){
#line 67
  unsigned char result;
#line 67

#line 67
  result = AMPromiscuous$RadioSend$sendDone(arg_0xdb6f10, arg_0xdb7060);
#line 67

#line 67
  return result;
#line 67
}

#line 80
static inline
# 239 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
result_t AMPromiscuous$RadioSend$sendDone(TOS_MsgPtr msg, result_t success)
#line 239
{
  return AMPromiscuous$reportSendDone(msg, success);
}

static
# 170 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
result_t AMPromiscuous$reportSendDone(TOS_MsgPtr msg, result_t success)
#line 170
{
  dbg(DBG_AM, "AM report send done for message to 0x%x, type %d.\n", msg->addr, msg->type);
  AMPromiscuous$state[tos_state.current_node] = FALSE;
  AMPromiscuous$SendMsg$sendDone(msg->type, msg, success);
  AMPromiscuous$sendDone();

  return SUCCESS;
}

# 49 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/SendMsg.nc"
inline static  result_t AMPromiscuous$SendMsg$sendDone(uint8_t arg_0xfc6e68, TOS_MsgPtr
arg_0xfc82f8, result_t arg_0xfc8448){
#line 49
  unsigned char result;
#line 49

#line 49
  result = QueuedSendM$SerialSendMsg$sendDone(arg_0xfc6e68, arg_0xfc82f8, arg_0xfc8448);
#line 49

#line 49
  return result;
#line 49
}
```

Appendix A – C Source Code of CC1000 **PowerTOSSIM** Targeted Surge Application

```
#line 106
static inline # 88 "SurgeM.nc"
void SurgeM$SendData(void)
#line 88
{
 SurgeMsg *pReading;
 uint16_t Len;

#line 91
 dbg(DBG_USR1, "SurgeM: Sending sensor reading\n");

 if ((pReading = (SurgeMsg
*)SurgeM$Send$getBuffer(&SurgeM$gMsgBuffer[tos_state.current_node], &Len)) != (void *)0) {
     pReading->type = SURGE_TYPE_SENSORREADING;
     pReading->parentaddr = SurgeM$RouteControl$getParent();
     pReading->reading = SurgeM$gSensorData[tos_state.current_node];

      if (SurgeM$Send$send(&SurgeM$gMsgBuffer[tos_state.current_node], sizeof(SurgeMsg )) !=
SUCCESS) {
        { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 103
         SurgeM$gfSendBusy[tos_state.current_node] = FALSE;
#line 103
         __nesc_atomic_end(__nesc_atomic); }
       }
   }
}

************************

# 83 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/Send.nc"
inline static  result_t SurgeM$Send$send(TOS_MsgPtr arg_0xec46b0, uint16_t arg_0xec4800){
#line 83
 unsigned char result;
#line 83

#line 83
  result = MultiHopEngineM$Send$send(AM_SURGEMSG, arg_0xec46b0, arg_0xec4800);
#line 83

#line 83
 return result;
#line 83
}

************************

#line 86
static inline # 124 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Route/MultiHopEngineM.nc"
result_t MultiHopEngineM$Send$send(uint8_t id, TOS_MsgPtr pMsg, uint16_t PayloadLen)
#line 124
{

 uint16_t usMHLength = (size_t )& ((TOS_MHopMsg *)0)->data + PayloadLen;

 if (usMHLength > 29) {
     return FAIL;
   }
 MultiHopEngineM$RouteSelect$initializeFields(pMsg, id);
```

```
 if (MultiHopEngineM$RouteSelect$selectRoute(pMsg, id) != SUCCESS) {
     return FAIL;
   }
 if (MultiHopEngineM$SendMsg$send(id, pMsg->addr, usMHLength, pMsg) != SUCCESS) {
     return FAIL;
   }

 return SUCCESS;
}

********************************

# 48 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/SendMsg.nc"
inline static  result_t MultiHopEngineM$SendMsg$send(uint8_t arg_0x116c158, uint16_t
arg_0xff9bc8, uint8_t arg_0xff9d10, TOS_MsgPtr arg_0xff9e60){
#line 48
 unsigned char result;
#line 48

#line 48
  result = QueuedSendM$QueueSendMsg$send(arg_0x116c158, arg_0xff9bc8, arg_0xff9d10,
arg_0xff9e60);
#line 48

#line 48
 return result;
#line 48
}
***************************
#line 81
static inline # 188 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Queue/QueuedSendM.nc"
result_t QueuedSendM$SerialSendMsg$sendDone(uint8_t id, TOS_MsgPtr msg, result_t success)
#line 188
{
 if (msg !=
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].pMsg) {
     return FAIL;
   }

 if ((!QueuedSendM$retransmit[tos_state.current_node] || msg->ack != 0) ||
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].address == TOS_UART_ADDR) {

     QueuedSendM$QueueSendMsg$sendDone(id, msg, success);

QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].pMsg = (void *)0;
     dbg(DBG_USR2, "qent %d dequeued.\n", QueuedSendM$dequeue_next[tos_state.current_node]);

     QueuedSendM$dequeue_next[tos_state.current_node]++;
#line 199
     QueuedSendM$dequeue_next[tos_state.current_node] %= QueuedSendM$MESSAGE_QUEUE_SIZE;
   }
 else
   {
     QueuedSendM$Leds$redToggle();
     if (++
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].xmit_count > QueuedSendM$MAX_RETRANSMIT_COUNT) {


         QueuedSendM$QueueSendMsg$sendDone(id, msg, FAIL);

QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
```

```
]].pMsg = (void *)0;
         QueuedSendM$dequeue_next[tos_state.current_node]++;
#line 216
         QueuedSendM$dequeue_next[tos_state.current_node] %= QueuedSendM$MESSAGE_QUEUE_SIZE;

       }
   }


  TOS_post(QueuedSendM$QueueServiceTask);

 return SUCCESS;
}

********************************

# 49 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/SendMsg.nc"
inline static  result_t QueuedSendM$QueueSendMsg$sendDone(uint8_t arg_0x111ba10, TOS_MsgPtr
arg_0xfc82f8, result_t arg_0xfc8448){
#line 49
  unsigned char result;
#line 49

#line 49
  result = MultiHopEngineM$SendMsg$sendDone(arg_0x111ba10, arg_0xfc82f8, arg_0xfc8448);
#line 49
  result = rcombine(result, BcastM$SendMsg$sendDone(arg_0x111ba10, arg_0xfc82f8,
arg_0xfc8448));
#line 49
  switch (arg_0x111ba10) {
#line 49
    case AM_MULTIHOPMSG:
#line 49
      result = rcombine(result, MultiHopLEPSM$SendMsg$sendDone(arg_0xfc82f8, arg_0xfc8448));
#line 49
      break;
#line 49
  }
#line 49

#line 49
  return result;
#line 49
}

********************************

static
# 205 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Route/MultiHopEngineM.nc"
result_t MultiHopEngineM$SendMsg$sendDone(uint8_t id, TOS_MsgPtr pMsg, result_t success)
#line 205
{

  if (pMsg ==
MultiHopEngineM$FwdBufList[tos_state.current_node][MultiHopEngineM$iFwdBufTail[tos_state.cur
rent_node]]) {
      MultiHopEngineM$iFwdBufTail[tos_state.current_node]++;
#line 208
      MultiHopEngineM$iFwdBufTail[tos_state.current_node] %= MultiHopEngineM$FWD_QUEUE_SIZE;
    }
  else
#line 209
    {
      MultiHopEngineM$Send$sendDone(id, pMsg, success);
    }
  return SUCCESS;
```

```
}

*******************************

# 119 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/Send.nc"
inline static  result_t MultiHopEngineM$Send$sendDone(uint8_t arg_0x116e450, TOS_MsgPtr
arg_0xec57c8, result_t arg_0xec5918){
#line 119
  unsigned char result;
#line 119

#line 119
  switch (arg_0x116e450) {
#line 119
    case AM_SURGEMSG:
#line 119
      result = SurgeM$Send$sendDone(arg_0xec57c8, arg_0xec5918);
#line 119
      break;
#line 119
    default:
#line 119
      result = MultiHopEngineM$Send$default$sendDone(arg_0x116e450, arg_0xec57c8,
arg_0xec5918);
#line 119
    }
#line 119

#line 119
  return result;
#line 119
}

*******************************
#line 81
static inline
# 166 "SurgeM.nc"
result_t SurgeM$Send$sendDone(TOS_MsgPtr pMsg, result_t success)
#line 166
{
  dbg(DBG_USR2, "SurgeM: output complete 0x%x\n", success);
  SurgeM$Leds$redToggle();


  { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 171
    SurgeM$gfSendBusy[tos_state.current_node] = FALSE;
#line 171
    __nesc_atomic_end(__nesc_atomic); }
  return SUCCESS;
}

*******************************

static inline
# 246 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Route/MultiHopEngineM.nc"
result_t MultiHopEngineM$Send$default$sendDone(uint8_t id, TOS_MsgPtr pMsg, result_t
success)
#line 246
{
  return SUCCESS;
}

*******************************
*******************************
*******************************
```

```
*******************************
*******************************

static # 121 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/lib/Queue/QueuedSendM.nc"
void QueuedSendM$QueueServiceTask(void)
#line 121
{
 uint8_t id;

 if
(QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_nod
e]].pMsg != (void *)0) {
     QueuedSendM$Leds$greenToggle();
     dbg(DBG_USR2, "QueuedSend: sending msg (0x%x)\n",
QueuedSendM$dequeue_next[tos_state.current_node]);
     id =
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].id;

      if (!QueuedSendM$SerialSendMsg$send(id,
QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].address,

QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].length,

QueuedSendM$msgqueue[tos_state.current_node][QueuedSendM$dequeue_next[tos_state.current_node
]].pMsg)) {



         dbg(DBG_USR2, "QueuedSend: send request failed. stuck in queue\n");
       }
   }
 else {
     QueuedSendM$fQueueIdle[tos_state.current_node] = TRUE;
   }
}

*******************************

# 48 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/SendMsg.nc"
inline static result_t QueuedSendM$SerialSendMsg$send(uint8_t arg_0x11523c8, uint16_t
arg_0xff9bc8, uint8_t arg_0xff9d10, TOS_MsgPtr arg_0xff9e60){
#line 48
 unsigned char result;
#line 48

#line 48
  result = AMPromiscuous$SendMsg$send(arg_0x11523c8, arg_0xff9bc8, arg_0xff9d10,
arg_0xff9e60);
#line 48

#line 48
 return result;
#line 48
}

*******************************
#line 106
static inline # 206 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
result_t AMPromiscuous$SendMsg$send(uint8_t id, uint16_t addr, uint8_t length, TOS_MsgPtr
data)
#line 206
{
 if (!AMPromiscuous$state[tos_state.current_node]) {
```

```
    AMPromiscuous$state[tos_state.current_node] = TRUE;
    AMPromiscuous$Leds$greenToggle();

    if (length > DATA_LENGTH) {
        dbg(DBG_AM, "AM: Send length too long: %i. Fail.\n", (int )length);
        AMPromiscuous$state[tos_state.current_node] = FALSE;
        return FAIL;
      }
    if (!TOS_post(AMPromiscuous$sendTask)) {
        dbg(DBG_AM, "AM: post sendTask failed.\n");
        AMPromiscuous$state[tos_state.current_node] = FALSE;
        return FAIL;
      }
    else {
        AMPromiscuous$buffer[tos_state.current_node] = data;
        data->length = length;
        data->addr = addr;
        data->type = id;
        AMPromiscuous$buffer[tos_state.current_node]->group = TOS_AM_GROUP;
        AMPromiscuous$dbgPacket(data);
        dbg(DBG_AM, "Sending message: %hx, %hhx\n\t", addr, id);
      }
    return SUCCESS;
  }

 return FAIL;
}

*****************
#line 58
static inline # 193 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/system/AMPromiscuous.nc"
void AMPromiscuous$sendTask(void)
#line 193
{
 result_t ok;

 if (AMPromiscuous$buffer[tos_state.current_node]->addr == TOS_UART_ADDR) {
   ok = AMPromiscuous$UARTSend$send(AMPromiscuous$buffer[tos_state.current_node]);
   }
 else {
#line 199
   ok = *AMPromiscuous$RadioSend$send(AMPromiscuous$buffer[tos_state.current_node]);*
   }
 if (ok == FAIL) {
   AMPromiscuous$reportSendDone(AMPromiscuous$buffer[tos_state.current_node], FAIL);
   }
}

************************

# 58 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/BareSendMsg.nc"
inline static  result_t AMPromiscuous$RadioSend$send(TOS_MsgPtr arg_0xdb69f8){
#line 58
 unsigned char result;
#line 58

#line 58
  result = CC1000RadioIntM$Send$send(arg_0xdb69f8);
#line 58

#line 58
 return result;
#line 58
}

***************
```

```
#line 63
static inline *# 426 "C:/tinyos/cygwin/opt/tinyos-
1.x/tos/platform/pc/CC1000Radio/CC1000RadioIntM.nc"
result_t CC1000RadioIntM$Send$send(TOS_MsgPtr pMsg)
#line 426
{
 result_t Result = SUCCESS;

 { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 429
   {
     if (CC1000RadioIntM$bTxBusy[tos_state.current_node]) {
         Result = FAIL;
       }
     else {
         CC1000RadioIntM$bTxBusy[tos_state.current_node] = TRUE;
         CC1000RadioIntM$txbufptr[tos_state.current_node] = pMsg;
         CC1000RadioIntM$txlength[tos_state.current_node] = pMsg->length + (MSG_DATA_SIZE -
DATA_LENGTH - 2);

         CC1000RadioIntM$sMacDelay[tos_state.current_node] = MSG_DATA_SIZE +
(CC1000RadioIntM$Random$rand() & 0x7F);
         CC1000RadioIntM$bTxPending[tos_state.current_node] = TRUE;
       }
   }
#line 441
   __nesc_atomic_end(__nesc_atomic); }

 if (Result) {
     uint8_t tmpState;

#line 445
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 445
       tmpState = CC1000RadioIntM$RadioState[tos_state.current_node];
#line 445
       __nesc_atomic_end(__nesc_atomic); }

     if (tmpState == CC1000RadioIntM$POWER_DOWN_STATE) {

         CC1000RadioIntM$WakeupTimer$stop();
         CC1000RadioIntM$CC1000StdControl$start();
         CC1000RadioIntM$CC1000Control$BIASOn();
         CC1000RadioIntM$CC1000Control$RxMode();
         CC1000RadioIntM$SpiByteFifo$rxMode();
         CC1000RadioIntM$SpiByteFifo$enableIntr();
         CC1000RadioIntM$WakeupTimer$start(TIMER_ONE_SHOT, 16 * 2);
         { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 456
           CC1000RadioIntM$RadioState[tos_state.current_node] = CC1000RadioIntM$IDLE_STATE;
#line 456
           __nesc_atomic_end(__nesc_atomic); }
       }
   }

 return Result;
}

********************************************
# 36 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/SpiByteFifo.nc"
inline static   result_t CC1000RadioIntM$SpiByteFifo$enableIntr(void){
#line 36
  unsigned char result;
#line 36
```

```
#line 36
  result = HPLSpiM$SpiByteFifo$enableIntr();
#line 36

#line 36
  return result;
#line 36
}
**********************


#line 41
static inline
# 93 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/HPLSpiM.nc"
result_t HPLSpiM$SpiByteFifo$enableIntr(void)
#line 93
{
  HPLSpiM$intrstate[tos_state.current_node] = HPLSpiM$INTR_ENABLED; //Alex: see
event_spi_byte_handle() - here it treats the HPLSpiM$INTR_ENABLED value
  HPLSpiM$PowerManagement$adjustPower(); //Alex: just returns IDLE - doesn't do anything
useful
  return SUCCESS;
}

**********************


#line 45
static inline  *# 477 "C:/tinyos/cygwin/opt/tinyos-
1.x/tos/platform/pc/CC1000Radio/CC1000RadioIntM.nc"
result_t CC1000RadioIntM$SpiByteFifo$dataReady(uint8_t data_in)
#line 477
{

 if (CC1000RadioIntM$bInvertRxData[tos_state.current_node]) {
   switch (CC1000RadioIntM$RadioState[tos_state.current_node]) {

       case CC1000RadioIntM$TX_STATE:
         {

CC1000RadioIntM$SpiByteFifo$writeByte(CC1000RadioIntM$NextTxByte[tos_state.current_node]);
            CC1000RadioIntM$TxByteCnt[tos_state.current_node]++;


         ........

                case CC1000RadioIntM$TXSTATE_DATA:
                  if ((uint8_t )CC1000RadioIntM$TxByteCnt[tos_state.current_node] <
CC1000RadioIntM$txlength[tos_state.current_node]) {
                      CC1000RadioIntM$NextTxByte[tos_state.current_node] = ((uint8_t
*)CC1000RadioIntM$txbufptr[tos_state.current_node])[CC1000RadioIntM$TxByteCnt[tos_state.curr
ent_node]];
                      CC1000RadioIntM$usRunningCRC[tos_state.current_node] =
crcByte(CC1000RadioIntM$usRunningCRC[tos_state.current_node],
CC1000RadioIntM$NextTxByte[tos_state.current_node]);
                      CC1000RadioIntM$RadioSendCoordinator$byte(CC1000RadioIntM$txbufptr[tos
_state.current_node], (uint8_t )CC1000RadioIntM$TxByteCnt[tos_state.current_node]);
                    }
                  else {
                      CC1000RadioIntM$NextTxByte[tos_state.current_node] = (uint8_t )
CC1000RadioIntM$usRunningCRC[tos_state.current_node];
                      CC1000RadioIntM$RadioTxState[tos_state.current_node] =
CC1000RadioIntM$TXSTATE_CRC;
                    }
                break;
```

```
      break;

      default:
        break;
    }
  }
 return SUCCESS;
}

**************************************

# 33 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/SpiByteFifo.nc"
inline static result_t CC1000RadioIntM$SpiByteFifo$writeByte(uint8_t arg_0x10264d8){
#line 33
 unsigned char result;
#line 33

#line 33
  result = HPLSpiM$SpiByteFifo$writeByte(arg_0x10264d8);
#line 33

#line 33
 return result;
#line 33
}

********************************************

#line 81
static inline # 74 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/HPLSpiM.nc"
result_t HPLSpiM$SpiByteFifo$writeByte(uint8_t data)
#line 74
{
  HPLSpiM$OutgoingByte[tos_state.current_node] = data;
  return SUCCESS;
}

***************************************

#line 42
static inline
# 68 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/HPLSpiM.nc"
void SIG_SPI_signal(void)
#line 68
{
 uint8_t temp = HPLSpiM$spdr[tos_state.current_node];

#line 70
  HPLSpiM$spdr[tos_state.current_node] = HPLSpiM$OutgoingByte[tos_state.current_node];
  HPLSpiM$SpiByteFifo$dataReady(temp);
}


******************************************

# 175 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/CC1000Radio/HPLSpiM.nc"
void event_spi_byte_handle(event_t *fevent,
struct TOS_state *fstate)
#line 176
{
 event_queue_t *queue = & fstate->queue;
 spi_byte_data_t *data = (spi_byte_data_t *)fevent->data;
 uint8_t temp;

#line 180
 radioWaitingState[tos_state.current_node] = NOT_WAITING;
```

```
 if (data->ending) {
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 182
       {
         spiByteEvents[tos_state.current_node] = (void *)0;
       }
#line 184
       __nesc_atomic_end(__nesc_atomic); }
     tos_state.rfm->stop_transmit(tos_state.current_node);
     dbg(DBG_RADIO, "RADIO: Spi Byte event ending for mote %i at %lli discarded.\n", data-
>mote, fevent->time);
     event_cleanup(fevent);
   }
 else {
   if (data->valid) {
       tos_state.rfm->stop_transmit(tos_state.current_node);
       if (dbg_active(DBG_RADIO)) {
           char ttime[128];

#line 194
           ttime[0] = 0;
           printTime(ttime, 128);
           dbg(DBG_RADIO, "RADIO: Spi Byte event handled for mote %i at %s with interval of
%i.\n", fevent->mote, ttime, data->interval);
         }



        if (HPLSpiM$state[tos_state.current_node] == HPLSpiM$RX_STATE) {
           temp = TOSH_rfm_rx_bit();
           temp &= 0x01;
            HPLSpiM$spdr[tos_state.current_node] <<= 1;
           HPLSpiM$spdr[tos_state.current_node] |= temp;
         }
       else {
          if (HPLSpiM$state[tos_state.current_node] == HPLSpiM$TX_STATE) {
             temp = (HPLSpiM$spdr[tos_state.current_node] >> 0x7) & 0x1;
             TOSH_rfm_tx_bit(temp);
             HPLSpiM$spdr[tos_state.current_node] <<= 1;
           }
          else {
             dbg(DBG_ERROR, "SpiByteFifo is seriously wacked\n");
           }
         }

       if (data->count == 7) {

           if (HPLSpiM$intrstate[tos_state.current_node] == HPLSpiM$INTR_ENABLED) {
             SIG_SPI_signal();
             }
         }
       data->count = (data->count + 1) & 0x07;
       fevent->time = fevent->time + data->interval;
       queue_insert_event(queue, fevent);
     }
   else
     {
       dbg(DBG_RADIO, "RADIO: invalid Spi Byte event for mote %i at %lli discarded.\n",
data->mote, fevent->time);

       event_cleanup(fevent);
     }
   }
}
```

****

```
static inline
# 398 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/hpl.c"
void TOSH_rfm_tx_bit(uint8_t data)
{
  tos_state.rfm->transmit(tos_state.current_node, (char )(data & 0x01));
  dbg(DBG_RADIO, "RFM: Mote %i sent bit %x\n", tos_state.current_node, data & 0x01);
}
```

```
***************************
See function main() for assignment of tos_state.rfm:

# 122 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/pc/Nido.nc"
int main(int argc, char **argv)
{
....

  if (model_name == (void *)0 || strcmp(model_name, "simple") == 0) {
     tos_state.rfm = create_simple_model();
     tos_state.radioModel = TOSSIM_RADIO_MODEL_SIMPLE;
   }
 else {
#line 261
   if (strcmp(model_name, "lossy") == 0) {
       tos_state.rfm = create_lossy_model(lossy_file);
       tos_state.radioModel = TOSSIM_RADIO_MODEL_LOSSY;

............
}
```

```
*****************************
static inline
#line 342
void lossy_transmit(int moteID, char bit)
#line 342
{
 link_t *current_link;

 pthread_mutex_lock(&radioConnectivityLock);
 current_link = radio_connectivity[moteID];
 transmitting[moteID] = bit;
 while (current_link) {
      int r = rand() % 100000;
     double prob = (double )r / 100000.0;
      int tmp_bit = bit;

#line 352
     if (prob < current_link->data) {
         tmp_bit = tmp_bit ? 0 : 1;
       }
     radio_active[current_link->mote] += tmp_bit;
     radio_idle_state[current_link->mote] = 0;
     current_link->bit = tmp_bit;
     current_link = current_link->next_link;
   }
 pthread_mutex_unlock(&radioConnectivityLock);
}
```

Appendix B – C Source Code for the Real CC1000 (Mica2 Platform) Implementation

```
#line 42
# 53 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/mica2/HPLSpiM.nc"
void __attribute((signal))   __vector_17(void)
#line 53
{
  register uint8_t temp = * (volatile unsigned char *)(unsigned int )& * (volatile unsigned
char *)(0x0F + 0x20);

#line 55
  * (volatile unsigned char *)(unsigned int )& * (volatile unsigned char *)(0x0F + 0x20) =
HPLSpiM$OutgoingByte;
  HPLSpiM$SpiByteFifo$dataReady(temp);
}
```

Appendix C – C Source Code of the nRF2410 Targetted Surge Application

```
# 58 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/interfaces/BareSendMsg.nc"
inline static result_t AMPromiscuous$RadioSend$send(TOS_MsgPtr arg_0x1053918){
#line 58
 unsigned char result;
#line 58

#line 58
  result = nRF2401RadioM$Send$send(arg_0x1053918);
#line 58

#line 58
 return result;
#line 58
}

********************
#line 88
static inline # 432 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/nRF2401RadioM.nc"
result_t nRF2401RadioM$Send$send(TOS_MsgPtr pMsg)
#line 432
{

 result_t Result = SUCCESS;

 { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 436
   {
     if (nRF2401RadioM$bTxBusy) {
         Result = FAIL;
       }
     else {
         nRF2401RadioM$bTxBusy = TRUE;

         nRF2401RadioM$txMsgptr = pMsg;
         nRF2401RadioM$txlength = pMsg->length + (MSG_DATA_SIZE - DATA_LENGTH);*

         if (nRF2401RadioM$bAckEnable) {
              switch (nRF2401RadioM$MacTableTx$processOutPacket(pMsg->addr)) {


                 case TX_TABLE_FULL:
                   {
                     nRF2401RadioM$bTxBusy = FALSE;
                     Result = FAIL;
                   }
                 break;

                 case TX_ENTRY_AVAILABLE:

                   case TX_EXPIRED_ENTRY:

                     nRF2401RadioM$MacTableTx$addEntry(pMsg->addr, 0, TX_TABLE_POSITION);
                 pMsg->seqNo = 0;
                 break;

                 case TX_VALID_ENTRY:

                   pMsg->seqNo = nRF2401RadioM$MacTableTx$updateEntry(pMsg->addr,
TX_TABLE_POSITION);
               }
```

```
          }


         if (Result != FAIL) {

             pMsg->s_addr = TOS_LOCAL_ADDRESS;

             nRF2401RadioM$BackOffState = nRF2401RadioM$BO_SENSE;
             if (nRF2401RadioM$RadioDCState != nRF2401RadioM$DC_LISTEN) {
               nRF2401RadioM$nRF2401Control$RxMode();
               }
#line 479
             nRF2401RadioM$sMacDelay = 80;
             nRF2401RadioM$BackOffTimer$start(TIMER_ONE_SHOT, nRF2401RadioM$sMacDelay);
           }
       }
   }
#line 483
   __nesc_atomic_end(__nesc_atomic); }

 if (Result) {

     uint8_t tmpState;

#line 488
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 488
       tmpState = nRF2401RadioM$RadioState;
#line 488
       __nesc_atomic_end(__nesc_atomic); }

     if (tmpState == nRF2401RadioM$POWER_DOWN_STATE) {
         nRF2401RadioM$nRF2401StdControl$start();
         nRF2401RadioM$nRF2401Control$RxMode();
         { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 493
           nRF2401RadioM$RadioState = nRF2401RadioM$IDLE_STATE;
#line 493
           __nesc_atomic_end(__nesc_atomic); }
       }
   }

 return Result;
}
********************
# 88 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/MacTable.nc"
inline static   TX_PKT_PROCESSING_OUTCOME nRF2401RadioM$MacTableTx$processOutPacket(uint16_t
arg_0x10d1e68){
#line 88
 enum __nesc_unnamed4265 result;
#line 88

#line 88
 result = MacTableM$MacTable$processOutPacket(arg_0x10d1e68);
#line 88

#line 88
 return result;
#line 88
}

************************
************************
******GOT LOST!!!!*******
************************
************************
```

```
************************

static
# 204 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/nRF2401RadioM.nc"
result_t nRF2401RadioM$SendToSPI(void)
#line 204
{

 uint16_t TxBCnt;
 uint8_t AddrCnt;
 uint8_t NextTxByte;
 uint8_t AddrSize;
 uint8_t aux_MAC_TxCounter;
 bool auxbAckEnable;

 uint16_t aux_txlength;

 { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 215
   {
     aux_MAC_TxCounter = nRF2401RadioM$MAC_TxCounter;
      nRF2401RadioM$TxByteCnt++;
     TxBCnt = nRF2401RadioM$TxByteCnt;
      aux_txlength = nRF2401RadioM$txlength;
     auxbAckEnable = nRF2401RadioM$bAckEnable;
     AddrCnt = 0;
   }
#line 222
   __nesc_atomic_end(__nesc_atomic); }



 AddrSize = nRF_ADDRWCRC[1 - 1] >> 5;

 while (AddrCnt < AddrSize) {
     nRF2401RadioM$SpiByte$write(nRF_ADDR1[5 - AddrSize + AddrCnt]);
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 230
       AddrCnt++;
#line 230
       __nesc_atomic_end(__nesc_atomic); }
   }





 while ((uint8_t )TxBCnt < aux_txlength)
   {
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 239
       NextTxByte = ((uint8_t *)nRF2401RadioM$txMsgptr)[TxBCnt];
#line 239
       __nesc_atomic_end(__nesc_atomic); }

     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 241
       {
         TxBCnt++;
         nRF2401RadioM$TxByteCnt++;
       }
#line 244
       __nesc_atomic_end(__nesc_atomic); }

     nRF2401RadioM$SpiByte$write(NextTxByte);
   }
```

```
 while ((uint8_t )(TxBCnt * 8) < nRF_DATA1_W[0])
   {
     nRF2401RadioM$SpiByte$write(0xFF);
     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 252
       TxBCnt++;
#line 252
       __nesc_atomic_end(__nesc_atomic); }
   }

 { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 255
   nRF2401RadioM$num_rtx++;
#line 255
   __nesc_atomic_end(__nesc_atomic); }

  nRF2401RadioM$nRF2401Control$SBurstSend();

 if (nRF2401RadioM$txMsgptr->addr != TOS_BCAST_ADDR && auxbAckEnable) {


     { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
#line 262
       nRF2401RadioM$bWaitingAck = TRUE;
#line 262
       __nesc_atomic_end(__nesc_atomic); }

     nRF2401RadioM$nRF2401Control$RxMode();

     TOSH_uwait(4000);

     nRF2401RadioM$retransmission_control();
   }
 else {
   nRF2401RadioM$retransmission_control();
   }
 return SUCCESS;
}

****************
#line 67
# 53 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/msp430/SpiByte.nc"
inline static   uint8_t nRF2401RadioM$SpiByte$write(uint8_t arg_0x10dc0a0){
#line 53
 unsigned char result;
#line 53

#line 53
  result = HPLSpiM$SpiByte$write(arg_0x10dc0a0);
#line 53

#line 53
 return result;
#line 53
}

****************
static # 43 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/msp430/HPLSpiM.nc"
uint8_t HPLSpiM$SpiByte$write(uint8_t data)
#line 43
{
 uint8_t retdata;

#line 45
 { __nesc_atomic_t __nesc_atomic = __nesc_atomic_start();
```

```
#line 45
   {
      HPLSpiM$USARTControl$tx(data);
      while (!HPLSpiM$USARTControl$isRxIntrPending()) ;
     retdata = HPLSpiM$USARTControl$rx();
   }
#line 49
   __nesc_atomic_end(__nesc_atomic); }
 return retdata;
}
****************
# 202 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/msp430/HPLUSARTControl.nc"
inline static   result_t HPLSpiM$USARTControl$tx(uint8_t arg_0x112d6e8){
#line 202
 unsigned char result;
#line 202

#line 202
  result = HPLUSART0M$USARTControl$tx(arg_0x112d6e8);
#line 202

#line 202
 return result;
#line 202
}

****************
static inline  # 473 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/msp430/HPLUSART0M.nc"
result_t HPLUSART0M$USARTControl$tx(uint8_t data)
#line 473
{
 HPLUSART0M$U0TXBUF = data;
 return SUCCESS;
}

****************
#line 88
inline static   result_t nRF2401RadioM$nRF2401Control$SBurstSend(void){
#line 88
 unsigned char result;
#line 88

#line 88
  result = nRF2401ControlM$nRF2401Control$SBurstSend();
#line 88

#line 88
 return result;
#line 88
}
********************
static # 217 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/nRF2401ControlM.nc"
result_t nRF2401ControlM$nRF2401Control$SBurstSend(void)
#line 217
{

 if (nRF2401ControlM$RadioState != nRF2401ControlM$RADIO_TX) {
#line 219
   return FAIL;
   }
#line 220
  nRF2401ControlM$HPLNordic$SBurstSend();
  TOSH_uwait(195);
 TOSH_uwait(1100);
 nRF2401ControlM$RadioState = nRF2401ControlM$RADIO_ST_BY;
 return SUCCESS;
```

```
}

*******************

# 81 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/HPLnRF2401.nc"
inline static   result_t nRF2401ControlM$HPLNordic$SBurstSend(void){
#line 81
 unsigned char result;
#line 81

#line 81
  result = HPLnRF2401M$HPLnRF2401$SBurstSend();
#line 81

#line 81
 return result;
#line 81
}
******************

#line 53
static inline   # 305 "C:/tinyos/cygwin/opt/tinyos-1.x/tos/platform/imec/HPLnRF2401M.nc"
result_t HPLnRF2401M$HPLnRF2401$SBurstSend(void)
#line 305
{

  TOSH_CLR_RF_CE_PIN();
 return SUCCESS;
}

****************

static inline
#line 52
void TOSH_CLR_RF_CE_PIN(void)
#line 52
{

#line 52
 static volatile uint8_t r __asm ("0x001D");

#line 52
 r &= ~(1 << 5);
}
```

References:

[B-MAC] J. Polastre, J. Hill, D. Culler, Versatile Low Power Media Access for Wireless Sensor Networks, Proc. of SenSys 2004

[SIM] David Curren - "A Survey of Simulation in Sensor Networks", 2004??

[TOSSIM-Sensys03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications"

[TOSSIM-Manual] Philip Levis and Nelson Lee. "TOSSIM: A Simulator for TinyOS Networks"

[PowerTOSSIM] Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh. "Simulating the Power Consumption of LargeScale Sensor Network Applications"

[Master] Ioannis Daskalopoulos, Hamadoun Diall, Kishore Raja. "Power-efficient and Reliable MAC for Routing in Wireless Sensor Networks", Master Thesis from University College London