MICA HIGH SPEED RADIO STACK

by

Nelson Lee, Philip Levis and Jason Hill

Memorandum No. UCB/ERL M02/34

11 September 2002

MICA HIGH SPEED RADIO STACK

by

Nelson Lee, Philip Levis and Jason Hill

Memorandum No. UCB/ERL M02/34

11 September 2002

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Mica High Speed Radio Stack

Nelson Lee, Philip Levis, Jason Hill

September 11, 2002

Introduction

This document describes the TinyOS networking stack released in TinyOS 1.0. This stack provides variable length packets and data-link level synchronous acknowledgements at a 40Kb data rate; it only works on mica motes. This document assumes the reader is familiar with nesC.

The Old Network Stack

The pre-mica TinyOS networking components used a vertical protocol stack. It roughly had this structure:

```
      Application
           |
           V
      GENERIC_COMM
           |
           V
      AM_STANDARD
           |
           V
   CRCPACKETOBJ_SIGNAL
           |
           V
SEC_DED_RADIO_BYTE_SIGNAL
           |
           V
          RFM
```

This vertical layering made each component dependent on the components directly above and below it, and allowed different components (e.g. a non CRC packet) to be easily interchanged. However, experience has shown that most of the interesting and important functionality had to be encapsulated in SEC_DED_RADIO_BYTE_SIGNAL, as only it could use the bit-level interface to the radio (RFM).

For example, SEC_DED_RADIO_BYTE_SIGNAL was responsible for the MAC layer, packet start symbol detection, and data encoding/decoding: three very separate pieces of functionality.

Introducing the New Radio Stack in nesC

In nesC, configuration files link components together according to the interfaces they use and provide. The hierarchy that links applications to the radio stack is as follows:

```
        Application
            |
            V
        GenericComm (configuration ''/tos/system/'')
            |
            V
        AMStandard (module ''/tos/system'')
            |
            V
        RadioCRCPacket (configuration ''/tos/platform/mica'')
            |
            V
        MicaHighSpeedRadioM (module ''/tos/platform/mica'')
            |
 -----------|----------------------------------------
 |          |         |          |         |        |
 |          |         |          |         |        |
 V          |         V          |         V        |
ChannelMonC.td  |  RadioTimingC.td  |  SlavePinC.td  |
                V                   V                 V
        SpiByteFifoC.td     SecDedEncoding.td  RandomLFSR.td
```

All componenents below RadioCRCPacket, except for RandomLFSR, are implemented in /tos/platform/mica.

A Brief Overview

Several components combine to form the network stack.

- MicaHighSpeedM contains the logic and state at the packet-level, and acts as a central controller for all of the components below it. It does not communicate directly to hardware, instead, it calls on other components to do so.
- ChannelMonC observes the radio at bit-level at 20kbps. When the stack is idle, it samples waiting for the preamble and start symbol. When the stack is sending a packet and is in backoff, ChannelMon monitors the radio and signals idleDetect to MicaHighSpeeedM.
- SpiByteFifo provides a byte-level abstraction to the radio. In essence, it uses the Serial Peripheral Interface (SPI) of the ATmega103 processor to shift out bits to the radio when sending, and shift in bits from the radio when receiving at 40kbps.
- SlavePinC calls HPL functions to flip the SlavePin high and low.
- RadioTiming uses counters on the ATmega103 and input capture to sync a receiver of a packet to the sender.

- `SecDedEncoding` provides a byte-level implementation of encoding/decoding single error correction and double error detection.
- `RandomLFSR` returns a 16 bit random number. This is used by `ChannelMon` to determine the length of the backoff state in radio clock ticks.

Init/Idle

The network stack is initialized by calling init() in `MicaHighSpeedRadioM`. In turn, `RandomLFSR` is initialized and `ChannelMonC` is initialized. `RandomLFSR` initializes the seed from the ID of the mote for the random number generator. `ChannelMonC` sets its `CM_waiting` field to -1, sets the radio hardware to receiving, scales timer2 and compare register2, clears the current counter value and enables timer2's interrupt to go off every 200 clock ticks (200 clock ticks/bit = 4MHz/20kbps).

Every time timer2's interrupt fires, `TOSH_SIGNAL(SIG_OUTPUT_COMPARE2)` is called in `ChannelMonC`. While the entire network stack is idle (`MicaHighSpeedRadioM` has not accepted any packets and its `state` and `send_state` are both `IDLE_STATE`), it shifts in the bit received into a buffer and checks for the preamble. Preamble/start symbol detection will be discussed in further detail below.

The new TOSMsg format

The new structure of the `TOS_Msg` (the struct declaration can be found in ``/tos/system/AM.h'':

```
typedef struct TOS_Msg
{
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[TOSH_DATA_LENGTH];
  uint16_t crc;
  uint16_t strength;
  uint8_t ack;
  uint16_t time;
} TOS_Msg;
```

It consists of an unsigned two byte field `addr`, followed by three unsigned single byte fields `type`, `group`, and `length` `addr` specifies a moteID or the broadcast address (`0xffff`). When the `MicaHighSpeedRadioStackM` receives a packet, the packet is passed to the AM level. If `addr` is not the broadcast address nor the address of the mote receiving the packet, the packet is dropped. The `group` field specifies a channel for motes on a network. If a mote receives a packet sent by a mote with a different group field, the packet is dropped at the AM level. The default group is 0x7d. The `type` field specifies which handler to be called at the AM level when a packet is received. The `length` field specifies the length of the data portion of the `TOS_Msg`. Packets have a maximum payload of 29 bytes.

The next field in the `TOS_Msg` struct is the `data` portion. It consists of an array of 29 bytes (as specified by `TOSH_DATA_LENGTH`). The unsigned two byte field `crc` follows. When sending, the CRC is incrementally calculated as each byte of the packet is transmitted. The maximum length of a transmitted `TOS_Msg` is 36 bytes (`addr`(2 bytes) + `type`(1 bytes) + `group`(1 bytes) + `length`(1 bytes) + data(29 bytes) + crc(2 bytes = 36 bytes)). The strength, ack, and time fields are not transmitted; they are meta-data about the packet.

The last three fields of TOS_Msg are the single unsigned byte ack field, the unsigned two byte strength and unsigned two byte time fields. The ack is sent by the receiver, and set by the sender. This is the mechanism that can provide reliability in the stack. When the network stack finishes sending a packet, it will return the TOS_MsgPtr to the application that issued the send request, with the ack field set to either 1 or 0. If the field is 1, the data link layer received an acknowledgement for the packet. When a packet is received, the data link layer transmits an ack if the receiving mote is a valid destination for the packet: (rec_ptr− >addr == TOS_LOCAL_ADDRESS || rec_ptr− >addr == TOS_BCAST_ADDR). The strength field of TOS_Msg is currently unused, and the time field stores an atomic capture of a 16-bit 4MHz counter.

Sending a Packet

MicaHighSpeedRadioM contains two state variables, send_state and state. When AMStandard hands down a TOS_MsgPtr to send, MicaHighSpeedRadio's state must be IDLE_STATE. If it is IDLE_STATE, then the radio stack accepts the packet. Its state changes and does not return until a packet is completely sent, which includes the reception of an ack.

MicaHighSpeedRadioM then calls macDelay() in ChannelMonC. macDelay sets its CM_waiting field to a random number. CM_waiting specifies the number of ChannelMonC clock ticks (one ChannelMonC clock tick is equal to 200 ATmega clock ticks at 4MHz) to wait for idle over the network. This Backoff state, as described previously, ensures that a sender of a packet in the network will not interfere with the transmission of another sender's packet in the network. The random factor prevents starvation.

Now, since ChannelMonC is waiting for idleness in the network, each call to TOSH_SIGNAL(SIG_OUTPUT_COMPARE2) in ChannelMonC decrements CM_waiting. When CM_waiting is equal to 1, it checks to see if during the past 12 ChannelMonC clock ticks a single 1 bit was not received (checking if CM_search[0] & 0xfff == 0). If so, it sets CM_waiting to -1, disables timer2's interrupt (thereby disabling ChannelMonC) and signals MicaHigSpeedRadioM that idleness was detected on the network and that it may begin sending over the radio. If activity was detected over the network, it sets CM_waiting to another random number and continues waiting for idleness.

It is important to note that while ChannelMonC searches for idleness over the network, it is simultaneously searching for a preamble. If a preamble is detected, ChannelMonC begins search for a start_symbol. This in effect switches the network into receive mode. However, when the network finishes receiving the packet or realizes that it falsely detected a preamble, ChannelMonC will return to IDLE_STATE and resume its detection for an idle network to send the packet it accepted to send.

ChannelMonC signals MicaHighSpeedRadioM via the idleDetect signal handler that the network is idle and ready for transmission. MicaHighSpeedRadioM then calls on SecDedEncoding to encode the first byte of the TOS_Msg. Each byte to be encoded results in three bytes to be sent over the network. Hence, SecDedEncoding signals MicaHighSpeedRadioM three times for each byte called to be encoded. MicaHighSpeedRadioM then activates SpiByteFifoC to send the first byte of the preamble/start symbol (char start[12]), sets the time field of the TOS_Msg to be sent (send_ptr− >time), and begins crc calculation with the first byte of the TOS_Msg to be sent (send_ptr[0]). MicaHighSpeedRadioM's msg_length field is also calculated here. This value corresponds to the number of bytes to be encoded and sent over the network excluding the crc. This calculation proceeds as follows: taking the maximum number of unencoded bytes of a TOS_Msg that can be sent over the network (36), subtracting the maximum length of the data field (29) and the crc (2), adding the length field of the TOS_Msg, which specifies the number of bytes of the data array to be sent, results in the number of bytes to be encoded and sent over the network.

SpiByteFifoC holds at most two bytes at any time; the one that is currently being sent, and the one that is waiting to be sent (uint8_t nextByte). Being in IDLE state corresponds to inactivity in SpiByteFifoC. When its buffer is free, its state is open, and when its buffer is in use, its state is full.

SpiByteFifoC receives a byte to send, and if it is currently in its IDLE state, which in this particular case it will be since it was inactive before receiving the first byte of the start symbol, it will accept the byte and signal to MicaHighSpeedRadioM that data is ready. SpiByteFifoC also initializes the SPI hardware, initializes and sets timer2 (modifying registers TIMSK, TCNT2, OCR2, TCCR2), and sets the radio to transmit.

The hardware shift register used by the SPI is now configured to shift in a bit from the radio every 100 clock ticks (100 ticks/bit = 4MHz/40kbps). After eight bits are shifted out of the SPDR register (data register of the SPI hardware) and sent over the network, TOSH_SIGNAL(SIG_SPI) in SpiByteFifoC is called. The nextByte field of SpiByteFifoC is then output to SPDR and the hardware continues shifting a bit out and sending it over the network at 40kbps (1 bit every 100 clock ticks) for another group of eight bits. This is the primary interface to the radio hardware for sending out bits. Contrary to the old stack, there is no software layer that communicates directly to radio hardware when sending.

To understand the following explanations on the intricacies of MicaHighSpeedRadioM, the distinction between "calling send on a byte", "sending a byte", and "signalling that a byte has been sent" must be fully understood. SpiByteFifo keeps a single byte buffer. Calling send() will place the byte in the buffer; the byte is not immediately sent. SpiByteFifoC can be in one of three states: IDLE, when it not sending a byte, OPEN, when it is sending a byte but its buffer is open and can be used, and FULL, when it is sending a byte and has a byte in its buffer. When a byte has been sent, SpiByteFifoC signals a dataReady() event. As there is a one byte queue, the dataReady() event for a given byte may not be the one immediately following the send() request. The calling component must keep track of the send() and dataReady() counts to know which event is associated with a specific byte.

When MicaHighSpeedRadioM calls send on the first byte of the start symbol, its state changes to TRANSMITTING_START. At each signal of dataReady, it calls send on the next byte of the start symbol. After the tenth byte of the preamble/start symbol has been sent, meaning dataReady is signaled with the tenth byte, MicaHighSpeedRadioM calls send on the twelfth and final byte of the preamble/start symbol and changes its state to TRANSMITTING.

When dataReady is signaled for the eleventh byte of the preamble/start symbol, MicaHighSpeedRadioM calls send on the first encoded byte. MicaHighSpeedRadioM stores encoded bytes in its 4 byte array encoded_buffer. After send is called on two of the three encoded bytes for a single byte of the TOS_Msg, MicaHighSpeedRadioM will call encode on the next byte of the TOS_Msg to be encoded and buffered for sending. Using the field tx_count as an index into send_ptr cast into a char*, the byte pointed to will be the next byte encoded.

The field tx_count corresponds to the index of the next byte to be encoded and buffered for sending. Let's use the application CntToRfm to illustrate how exactly MicaHighSpeedRadioM behaves. The first packet sent by CntToRfm appears as follows:

```
TOS_Msg:                          encoded bytes
        addr    = 0xff             0x9b, 0x55, 0x55
                  0xff             0x9b, 0x55, 0x55
        type    = 0x4              0x52, 0xaa, 0x9a
        group   = 0x7d             0x48, 0x95, 0x59
        length  = 0x4              0x9b, 0x55, 0x55
        data    = 0x1              0x5b, 0xaa, 0x9a
                  0x0              0xa4, 0xaa, 0xaa
                  0x0              0xa4, 0xaa, 0xaa
                  0x0              0xa4, 0xaa, 0xaa
        crc     = 0xd9             0x58, 0x59, 0x69
                  0x2d             0x95, 0xa6, 0x59
```

At each call to SpiByteFifo.dataReady, send is called on the next encoded byte and enc_count is decremented. Therefore, taking the first byte of the TOS_Msg (0xff), the order of operations is as follows:

- tx_count is set to 1, and enc_count equals 3
- SpiByteFifo.dataReady() is signaled. Call SpiByteFifo.send(0x9b) on the first encoded byte, decrement enc_count to 2
- SpiByteFifo.dataReady() is signaled. Call SpiByteFifo.send(0x55) on the second encoded byte, decrement enc_count to 1. To fill up the encoded buffer, call Code.encode(next_data) where next_data is send_ptr[tx_count]. Increment tx_count to 2 and incrementally compute the crc (calc_crc = add_crc_byte(next_data, calc_crc)).
- Code.encodeDone() is signaled. Add the number of encoded bytes (3) to enc_count, to make it 4.
- SpiByteFifo.dataReady() is signaled. Call SpiByteFifo.send(0x55) on the third encoded byte (the final encoded byte of the first data byte of the packet). Decrement enc_count to 3.
- SpiByteFifo.dataReady() is signaled. Call SpiByteFifo.send(0x9b) on the fourth encoded byte (the first encoded byte of the second data byte of the packet). Decrement enc_count to 2.

This cycle repeats itself for each byte of the TOS_Msg that is sent over the radio. In the instance of the dataReady handler that calls send on the second to last byte of the encoded three bytes of the second to last byte of the TOS_Msg to be sent (in this case it would be the fifth to last encoded byte before the crc, 0xaa, refer to CntToRfm example above), tx_count is automatically changed to 34. Therefore, independent of what msg_length or the number of data bytes encoded and sent over the network is, the crc bytes will always be the last two byte encoded and called send on.

After the six bytes of the encoded crc are called send on, MicaHighSpeedRadioM changes its state to SENDING_STRENGTH_PULSE. The time from when MicaHighSpeedRadioM transitions from TRANSMITTING to SENDING_STRENGTH_PULSE, to the time when it transitions from SENDING_STRENGTH_PULSE to WAITING_FOR_ACK, two bytes of 0xff are sent. As the name of the state suggests, a strength pulse is sent. However, currently in the radio stack, the strength pulse is used merely as a timing mechanism.

After the strength pulse is sent, during the transition from SENDING_STRENGTH_PULSE to WAITING_FOR_ACK, SpiByteFifo.phaseShift() is called. phaseShift delays SpiByteFifoC, meaning SpiByteFifoC pauses before resuming shifting in bits from the radio.

Once MicaHighSpeedRadioM enters the WAITING_FOR_ACK state, it transitions the radio to receive mode. SpiByteFifoC continues to signal dataReady to MicaHighSpeedRadioM in 800 clock tick intervals (after 8 bits are shifted in), and the byte signalled (uint8_t data) corresponds to the byte heard over the radio. MicaHighSpeedRadioM listens for four bytes, and on the last one, if the byte is equal to 0x55, then it sets the TOS_Msg ack field to 1, indicating the message sent was properly received. A packetReceived task is then posted, which sets MicaHighSpeedRadioM to IDLE_STATE, sets ChannelMonC to IDLE_STATE and activates it to search for a preamble/start symbol, and passes the sent packet to the AM layer with the ack field and time fields set.

To summarize, the sender's interaction with the radio in the CntToRfm example is as follows:

```
                               bytes sent
                               -------------------
addr    = 0xff                 0x9b, 0x55, 0x55
          0xff                 0x9b, 0x55, 0x55
type    = 0x4                  0x52, 0xaa, 0x9a
group   = 0x7d                 0x48, 0x95, 0x59
length  = 0x4                  0x9b, 0x55, 0x55
data    = 0x1                  0x5b, 0xaa, 0x9a
          0x0                  0xa4, 0xaa, 0xaa
          0x0                  0xa4, 0xaa, 0xaa
          0x0                  0xa4, 0xaa, 0xaa
crc     = 0xd9                 0x58, 0x59, 0x69
          0x2d                 0x95, 0xa6, 0x59
strength                       0xff
pulse                          0xff

---phase shift occurs---
---radio now set to receiving---
                               byte received 0x55
                               byte received 0x55
                               byte received 0x55
                               data = byte received (send_ptr->ack = (data == 0x55))
DONE
```

Receiving a Packet

ChannelMonC initiates the reception of a packet. When the radio stack is initialized, ChannelMon.startSymbolSearch is called. This method initializes ChannelMonC to IDLE_STATE as described earlier in section init/idle. Once ChannelMonC detects a preamble, its state changes into START_SYMBOL_SEARCH, where it will shift in bits in search of a start symbol. If a start symbol was not detected after 30 bits received, it changes its state back to IDLE_STATE. If a start symbol was detected, it signals MicaHigSpeedRadioM startSymDetect.

In the startSymDetect handler, MicaHighSpeedRadioM changes its state to RX_STATE, sets the time field of the packet received to the current time, trivially sets the strength field of the packet to 0, synchronizes the receiver (RadioTiming.getTiming() and startReadBytes(tmp)) to the sender and activates SpiByteFifoC to begin shifting in bits. Synchronization details can be found in section Timing.

SpiByteFifoC is now configured to shift in bits sampled from the radio once every 100 clock ticks, and signals dataReady to MicaHighSpeedRadio after 8 bits have been sampled.

Now, each time dataReady is called in MicaHighSpeedRadioM, SpiByteFifoC will call decode on the byte received and returned by SpiByteFifoC. SecDedEncoding signals decodeDone to MicaHighSpeedRadioM after three bytes have been called to be decoded. Therefore, most of the logic for the receiver resides in the decodeDone handler.

Many constants are used in the decodeDone handler and they are MSG_DATA_SIZE, LENGTH_BYTE_NUMBER and DATA_LENGTH. MSG_DATA_SIZE is equal to 36, the number of bytes of a TOS_Msg up to and including the crc field. LENGTH_BYTE_NUMBER corresponds to the index of the length field of TOS_Msg when it is cast into a (char*). DATA_LENGTH corresponds to the size of the data field of a TOS_Msg, which is currently set to 29.

The logic decodeDone follows is nearly identical to the logic described in the previous section for the sender of the packet. Each time a byte is decoded, it is written into the buffer TOS_Msg (rec_ptr) using the index rec_count. The field msg_length, corresponds to the number of decoded bytes that should be received excluding the crc. The calculation for msg_length is the same as described in the previous section, except that it cannot be calculated until it has received the length field of the packet being sent (if(rec_count == LENGTH_BYTE_NUMBER){...}). For the sender, msg_length can be calculated right away because the length of the packet is passed as a paramter to the AM layer.

Once msg_length bytes have been received and decoded, rec_count is automatically set to 34 (if(rec_count == msg_length){...}). This occurs because the next two bytes decoded will be the crc, and the index of the first byte of the crc of rec_ptr, when cast as a (char*), is 34.

As a note regarding CRC reception and calculation, each byte received excluding the two crc bytes is used to calculate the CRC. After the crc has been received, it is compared with the calculated CRC. If they are the same, the crc field of the TOS_Msg is set to 1 (if(calc_crc == rec_ptr->crc){ rec_ptr->crc = 1; ...}). If not, rec_ptr->crc is set to 0.

If the received crc and calculated CRC match, MicaHighSpeedRadioM checks if the address of the packet was either its own moteID or the broadcast address. If so, it tells SpiByteFifoC to send the ack (0x55) and changes its state to ACK_SEND_STATE. If not, a call to SpiByteFifoC send is not made.

After receiving the last decoded byte of the packet being sent, the receiver will receive the first 0xff byte sent by the receiver during the sender's SENDING_STRENGTH_PULSE state. During this instance of dataReady, MicaHighSpeedRadio will call SpiByteFifo.txMode(), which keeps SpiByteFifoC active but changes the state of the hardware to transmit.

For the next five instances of dataReady, either 0x00 or 0x55 is sent over the wire: 0x00 if packet was corrupted or intended for a different mote, 0x55 if the packet was received properly and addressed to itself.

During the fifth instance of dataReady, MicaHighSpeedRadioM deactivates SpiByteFifo (call SpiByteFifo.idle()), and posts a packetReceived task. The packetReceived task sets the radio stack to IDLE_STATE, signals to the AM layer that the packet was received, and activates ChannelMonC to search for a preamble/start symbol (call ChannelMon.startSymbolSearch). The purpose for this check, "if(tmp != 0) rec_ptr = tmp;" in the packetReceived task is because the AM layer will return a TOS_Msg (tmp), but that TOS_Msg may be an application's buffer and different than the buffer used to receive the packet. Therefore, it is an established convention that the receive signal handler return a free TOS_Msg for the radio stack to use for reception of another packet when a packet was signalled upon reception.

To summarize, the receiver's interaction with the radio in the `CntToRfm` example is as follows:

```
                          bytes received
                          -------------------
addr    = 0xff            0x9b, 0x55, 0x55
          0xff            0x9b, 0x55, 0x55
type    = 0x4             0x52, 0xaa, 0x9a
group   = 0x7d            0x48, 0x95, 0x59
length  = 0x4             0x9b, 0x55, 0x55
data    = 0x1             0x5b, 0xaa, 0x9a
          0x0             0xa4, 0xaa, 0xaa
          0x0             0xa4, 0xaa, 0xaa
          0x0             0xa4, 0xaa, 0xaa
crc     = 0xd9            0x58, 0x59, 0x69
          0x2d            0x95, 0xa6, 0x59
strength pulse            0xff

---radio now set to sending---
                          byte sent 0x55
                          byte sent 0x55
                          byte sent 0x55
                          byte sent 0x55
                          byte sent 0x55
DONE
```

Timing

As discussed previously, there are two components that communicate directly with radio hardware: `SpiByteFifoC` and `ChannelMonC`. `SpiByteFifoC` reads from the radio and is the only component to send to the radio. It samples/outputs to the radio every 100 clock ticks (40kbps). `ChannelMonC` only reads from the radio, and this occurs every 200 clock ticks (20kbps).

When the sender sends the preamble/start symbol, the following bytes are sent over the wire at 40kbps (using `SpiByteFifoC`).

`start`[12] = {0xf0, 0xf0, 0xf0, 0xff, 0x00, 0xff, 0x0f, 0x00, 0xff, 0x0f, 0x0f, 0x0f};

`ChannelMonC` monitors for packet reception by searching for the preamble and start symbol. The following timing diagram illustrates the transmission and reception of the preamble/start symbol.

```
Sender     1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1

Receiver    ^   ^   ^   ^   ^   ^   ^   ^   ^   ^

           | | | | | | | | | | | | | | | | | | | |
           0         500       1000      1500
          (time in clock ticks)
----------------------------------------------------------------------
Sender     0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

Receiver    ^   ^   ^   ^   ^   ^   ^   ^   ^   ^

           | | | | | | | | | | | | | | | | | | | |
           2000      2500      3000      3500
----------------------------------------------------------------------
Sender     1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0

Receiver    ^   ^   ^*  ^   '   ^   '   ^   '   ^

           | | | | | | | | | | | | | | | | | | | |
           4000      4500      5000      5500
----------------------------------------------------------------------
Sender     0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1

Receiver    '   ^   '   ^   '   ^   '   ^   '   ^

           | | | | | | | | | | | | | | | | | | | |
           6000      6500      7000      7500
----------------------------------------------------------------------
Sender     0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 \ \ \ \

Receiver    '   ^   '**

           | | | | | | | | | | | | | | | | | | | |
           8000      8500      9000      9500

^ indicates when the Receiver received a bit using CM_search[0]
' indicates when the Receiver received a bit using CM_search[1]

The sender sends a bit once very 100 clock ticks.
```

ChannelMonC has an unsigned short CM_search[2] that it uses to shift in bits once every 200 clock ticks. When ChannelMonC is in its IDLE_STATE, it shifts in bits into CM_search[0] only. Everytime a bit is received (TOSH_SIGNAL(SIG_OUTPUT_COMPARE2)), it masks CM_search[0] with 0x777 and checks to see if it is equal to 0x707. If so, it changes its state to START_SYMBOL_SEARCH, sets both CM_search[0] and CM_search[1] to 0 and sets CM_startSymBits to 30.

preamble check: 0111 0000 0111
preamble mask in bits: 0111 0111 0111

bits received up to : 110 0110 0110 0111 1000 0111
preamble detection
As shown from the timing diagram above, the last bit received before start symbol detection is the 1 bit ChannelMonC samples right after 4400 clock ticks as indicated by a "*".

During start symbol detection, both CM_search[0] and CM_search[1] are used. Since TOSH_SIGNAL(SIG_OUTPUT_COMPARE2) runs once every 200 clock ticks and the start symbol sent by the sender is actually a 10kbps signal, the bits received in two consecutive instances of TOSH_SIGNAL(SIG_OUTPUT_COMPARE2) go to separate buffers. As shown in the timing diagram above, where there is a "^" the bit was shifted into CM_search[0], and where there is a "" the bit was shifted into CM_search[1].

The contents of CM_search[0] and CM_search[1] after preamble detection are shown below:

start_symbol mask: 0001 1111 1111
start_symbol check: 0001 0011 0101

CM_search[1]: 01 0011 0101
CM_search[0]: 10 1001 1010
In the timing diagram above, CM_search[1] will detect the start symbol before CM_search[0]. The bit received, as marked by the "**" is the last bit received by ChannelMonC. Upon receiving this bit, ChannelMonC disables itself and signals startSymDetect to MicaHighSpeedRadioM.

The next timing issue that needs to be discuessed is the synchronization/input capture the receiver of a packet performs after detecting the preamble/start symbol. In essence, since the sender is sending the packet at 40kbps and the receiver is receiving bits at 40kbps, it is crucial that they are in sync. Since start symbol detection was performed at 10kbps, having the receiver know when to start clocking in bits at 40kbps is critical. This is accomplished through input capture. The receiver loops until a 1 bit is received, and begins clocking in bits for the packet some offset from when the 1 bit was received.

In the timing diagram above, the first bit of the packet is sent at 9500 clock ticks, 100 after the last 1 bit sent at 9400 clock ticks. Seeing that the last bit received for the start symbol occurs sometime between 8400 and 8500 as marked by '**, the receiver synchronizes itself with the sender between 8500 and 9500. The bits over the wire during this time are:

```
1 1 1 0 0 0 0 1 1 1 1 \ \ \ \
| | | | | | | | | | |
8500      9000      9500

each bit separated by 100 clock ticks.
```

As soon as the '** bit is received, RadioTiming's getTiming method is called from MicaHighSpeedRadioM's startSymDetect. The code line "while(TOSH_READ_RFM_RXD_PIN()) { }" will hold the receiver in a spin loop until the 0 at 6800 clock ticks is received. RadioTimingC then enables input capture from the radio and the code line "while((inp(TIFR) & (0x1 ¡¡ ICF1)) == 0) { }" pauses the receiver until the 1 at 9200 is received. RadioTimingC returns the time the input capture occurred to MicaHighSpeedRadioM, and MicaHighSpeedRadioM then calls SpiByteFifo's startReadBytes with the time stamp of when the input capture occurred.

startReadyBytes sets SpiByteFifoC's state to reading, and delays itself based on the timestamp of when the input capture occurred to begin clocking in bits between 9600 and 9700, when the first TOS_Msg packet bit is sent over the network.

The last timing issue that needs to be addressed is the phase shift that occurs when the sender switches its state from SENDING_STRENGTH_PULSE to WAITING_FOR_ACK. Up to this point, the sender and receiver are in perfect sync. The sender sends at 40kbps and the receiver receives at 40kbps. When the sender and receiver switch roles for the transmission and reception of the ack, it is necessary for the sender of the packet, to delay SpiByteFifo so that it remains in sync with the receiver of the packet (the one sending the ack). The timing diagram below illustrates the phase shift.

```
                 <- tx        rx ->
sender:    1    1    1  ###  0    1    0    1

          tx ->
receiver:   1    0    1    0    1    0    1    0

            |    |    |    |    |    |    |    |    |
            0   100  200  300  400  500  600  700  800 (clock ticks)
```

As shown above, the sender is sending the last 3 bits of the strength pulse, 0xff. The ### indicates that the sender shifts its timing, changes its radio hardware to receive so that the next bit SpiByteFifoC shifts in occurs after the 0 bit is transmitted by the receiver of the packet shortly after 300 clock ticks.

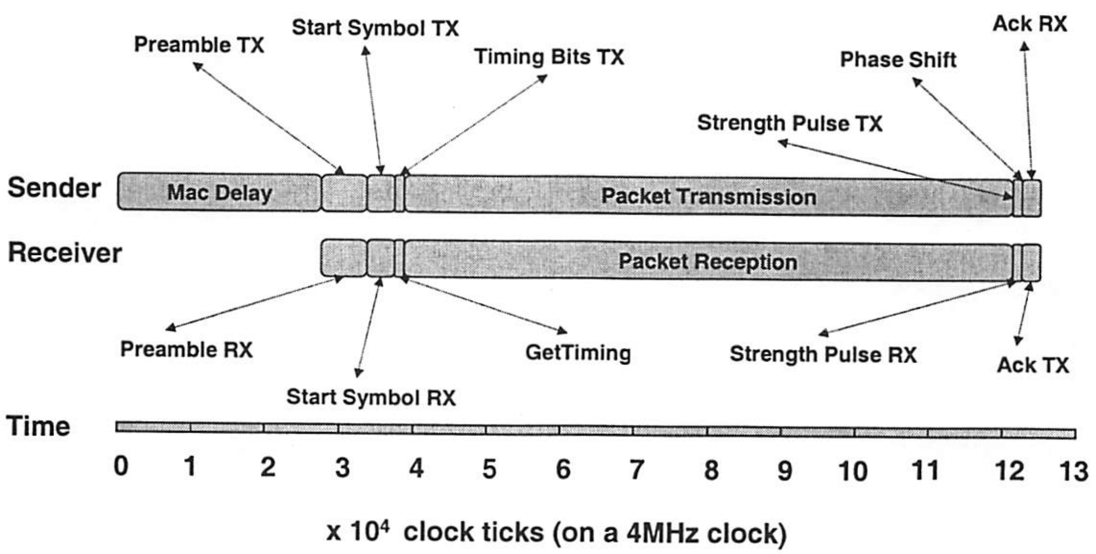

Figure 1: Timing Diagram of Network Send/Receive