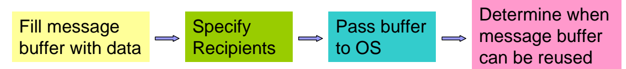
TinyOS
Network Communication

Computer Network Programming

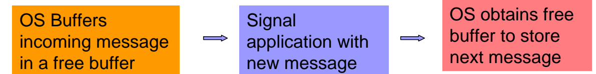
Wenyuan Xu

Inter-Node Communication

- General idea:
  - Sender:



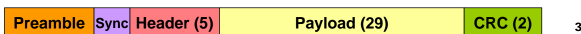
  - Receiver:



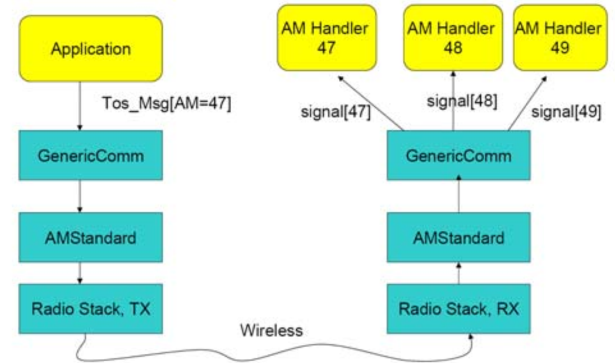
TOS Active Messages

- Message is “active” because it contains the destination address, group ID, and type.
- ‘group’: group IDs create a virtual network
  - an 8 bit value specified in <tos>/apps/Makelocal
- The address is a 16-bit value specified by “make”
  - – make install.<id> mica2
- “length” specifies the size of the message .
- “crc” is the check sum

```
typedef struct TOS_Msg {

// the following are transmitted
        uint16_t addr;
        uint8_t type;
        uint8_t group;
        uint8_t length;
        int8_t data[TOSH_DATA_LENGTH];
        uint16_t crc;

// the following are not transmitted
        uint16_t strength;
        uint8_t ack;
        uint16_t time;
        uint8_t sendSecurityMode;
        uint8_t receiveSecurityMode;

} TOS_Msg;
```



TOS Active Messages (continue)



Receiving a message

- Define the message format
- Define a unique active message number
- *How does TOS know the AM number?*

```
includes Int16Msg;
module ForwarderM {
  //interface declaration
}
implementation {
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m)
  {         call Leds.yellowToggle();
            call SendMsg.send(TOS_BCAST_ADDR,
                    sizeof(IntMsg), m);
            return m;
  }

  event result_t SendMsg.sendDone(TOS_MsgPtr msg, bool success)
  {         call Leds.greenToggle();
            return success;
  }
}
```

Message received

File: Int16Msg.h

```
struct Int16Msg {
   uint16_t val;
};

enum {
  AM_INTMSG = 47
};
```

```
configuration Forwarder { }
implementation
{ …
  ForwarderM.SendMsg -> Comm.SendMsg[AM_INTMSG];
  ForwarderM.ReceiveMsg -> Comm.ReceiveMsg[AM_INTMSG];
}
```


Sending a message

- Define the message format
- Define a unique active message number
- *How does TOS know the AM number?*

```
includes Int16Msg;
module ForwarderM {
  //interface declaration
}
implementation {
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m)
  {         call Leds.yellowToggle();
            call SendMsg.send(TOS_BCAST_ADDR,
                    sizeof(IntMsg), m);
            return m;
  }
  event result_t SendMsg.sendDone(TOS_MsgPtr msg, bool success)
  {         call Leds.greenToggle();
            return success;
  }
}
```

destination

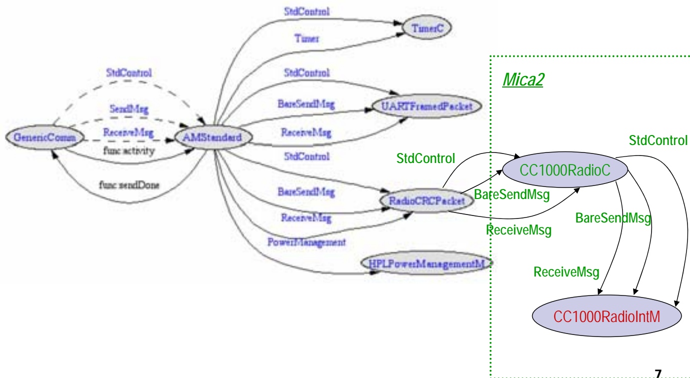
length

File: Int16Msg.h

```
struct Int16Msg {
   uint16_t val;
};

enum {
  AM_INTMSG = 47
};
```

```
configuration Forwarder { }
implementation
{ …
  ForwarderM.SendMsg -> Comm.SendMsg[AM_INTMSG];
  ForwarderM.ReceiveMsg -> Comm.ReceiveMsg[AM_INTMSG];
}
```

Where exactly is the radio stuff?



Spi bus interrupt handler

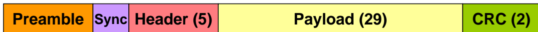
- Connection between Chipcon CC1000 radio and the ATmega128 processor: SPI bus.
- Spibus interrupt handler: SpiByteFifo.dataReady()
- SpiByteFifo.dataReady() will be called every 8 ticks.

```
file:CC1000RadioIntM.nc
async event result_t SpiByteFifo.dataReady(uint8_t data_in) {
   …
   switch (RadioState) {
              case RX_STATE: {...}
              case DISABLED_STATE: {…}
              case IDLE_STATE: {...}
              case PRETX_STATE: {...}
              case SYNC_STATE: {...}
              case RX_STATE: {...}
              return SUCCESS;
}
```

Receiving a message (1)

- IDLE_STATE ➔SYNC_STATE
  - Listen to the preamble, if the enough bytes of preamble are received, entering SYCN_STATE

```
file:CC1000RadioIntM.nc
async event result_t SpiByteFifo.dataReady(uint8_t data_in) {
   …
   switch (RadioState) {
              …
              case IDLE_STATE:
              {    if (((data_in == (0xaa)) || (data_in == (0x55)))) {
                      PreambleCount++;
                      if (PreambleCount > CC1K_ValidPrecursor) {
                        PreambleCount = SOFCount = 0;
                        RxBitOffset = RxByteCnt = 0;
                        usRunningCRC = 0;
                        rxlength = MSG_DATA_SIZE-2;
                        RadioState = SYNC_STATE;
                      }
                  }
              } …
}
```



Receiving a message (2)

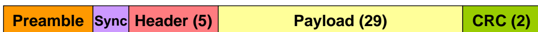
- SYNC_STATE ➔ RX_STATE
  - look for a SYNC_WORD (0x33cc).
  - Save the last received byte and current received byte
  - Use a bit shift compare to find the byte boundary for the sync byte
  - Retain the shift value and use it to collect all of the packet data
- SYNC_STATE ➔ IDLE_STATE
  - didn't find the SYNC_WORD after a reasonable number of tries, so set the radio state back to idle:
    - RadioState = IDLE_STATE;

```
file:CC1000RadioIntM.nc
async event result_t SpiByteFifo.dataReady(uint8_t data_in) {
   …
   switch (RadioState) {
              case SYNC_STATE: …
              {  if ( find SYCN_WORD) {
                   …
                   RadioState = RX_STATE;
                 } else if ( too many preamble) {
                   …
                   RadioState = IDLE_STATE; }
              } …
}
```

Receiving a message (3)

- RX_STATE ➔IDLE_STATE/SENDING_ACK
  - Keep receiving bytes and calculate CRC until the end of the packet.
  - The end of the packet are specified by the length in the packet header
  - Pass the message to the application layer, no matter whether the message passed the CRC check

```
file:CC1000RadioIntM.nc
async event result_t SpiByteFifo.dataReady(uint8_t data_in) { …
   switch (RadioState) {
              …
              case RX_STATE:
              { …RxByteCnt++;
                   if (RxByteCnt <= rxlength) {
                          usRunningCRC = crcByte(usRunningCRC,Byte);
                           if  (RxByteCnt == HEADER_LENGTH_OFFSET) {
                                    rxlength = rxbufptr->length;}
                    } else if (RxByteCnt == rxlength-CRCBYTE_OFFSET) {
                           if (rxbufptr->crc == usRunningCRC) {
                                    rxbufptr->crc = 1;
                           } else {
                                    rxbufptr->crc = 0;
                          } …
                    RadioState = IDLE_STATE;
                    post  PacketRcvd();
               } …
```

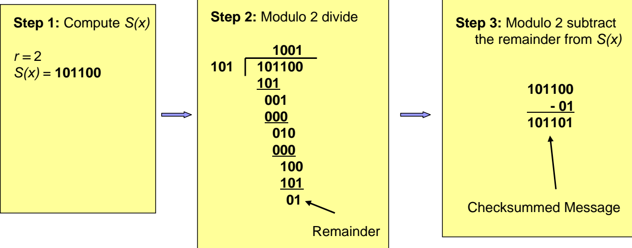


Error Detection – CRC

- CRC – Cyclic Redundancy Check
  - Polynomial codes or checksums
- Procedure:
  1. Let *r* be the degree of the code polynomial. Append *r* zero bits to the end of the transmitted bit string. Call the entire bit string *S(x)*
  2. Divide *S(x)* by the code polynomial using modulo 2 division.
  3. Subtract the remainder from *S(x)* using modulo 2 subtraction.
- The result is the checksummed message

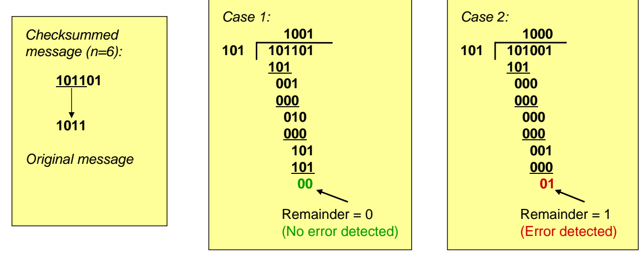
Generating a CRC – example

- **Message**: 1011
  - $1 * x^3 + 0 * x^2 + 1 * x + 1 = x^3 + x + 1$
- **Code Polynomial**: $x^2 + 1$ (101)



Decoding a CRC – example

- Procedure
  1. Let n be the length of the checksummed message in bits
  2. Divide the checksummed message by the code polynomial using modulo 2 division. If the remainder is zero, there is no error detected.



CRC in TinyOS

- Calculate the CRC byte by byte

```
crc=0x0000;
while (more bytes) {
  crc=crc^b<<8;
  calculate the high byte of crc
}
```

- Code Polynomial: CRC-CCITT
  0x1021 = 0001 0000 0010 0001

  $x^{16} + x^{12} + x^5 + 1$

file: system/crc.h

```
uint16_t crcByte(uint16_t crc, uint8_t b)
{         uint8_t i;

          crc = crc ^ b << 8;
          i = 8;
          do
              if (crc & 0x8000)
                    crc = crc << 1 ^ 0x1021;
              else
                    crc = crc << 1;
          while (--i);
          return crc;
}
```