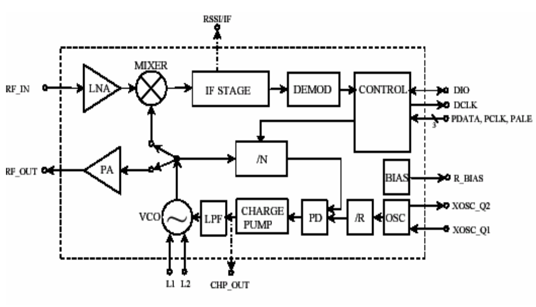
Radio Protocols

- UHF (300-1000Hz)
  - Mote radio
- Bluetooth (2.4GHz)
  - Common in many consumer devices (PDAs, cell phones, etc.)
- Zigbee (850-930MHz)
  - Next generation radio for sensor networks and consumer devices

Mote Radio

- ChipCon CC1000
  - Single-chip RF transceiver
  - Programmable frequency (300-1000 MHz)
  - Very low current consumption (Rx: 7.4 mA, Tx: 10.4 mA)
  - Very few external components required
  - FSK (frequency-shift-key) modulation spectrum shaping
  - Manchester encoded data
  - Low supply voltage (2.1 - 3.6 V)
  - High receiver sensitivity (-110 dBm)
  - RSSI output
  - FSK data rate up to 76.8 kBaud (motes use 38.4Kb)
  - Programmable frequency in 250 Hz steps
  - Suitable for frequency hopping protocols
  - Single-port antenna connection
  - Small 28-pin TSSOP package

Chipcon CC1000 Block Diagram



Radio on Motes

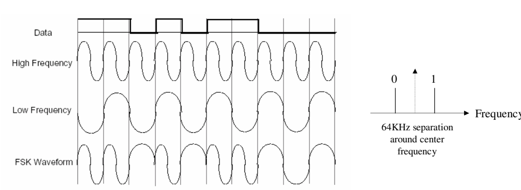
- Integrated CC1000 radio package
- Connection to ATmega microcontroller through SPI bus
  - CC1000 is master
  - Rx is double buffered, Tx is single buffered
- ATmega interfaces at the bit level
  - On transmit, new bits must be provided at the right rate
  - On receive, new bits must be collected at the right rate
  - CC1000 has a byte-wide interface
  - At 38.4Kbps, a new byte every ~8*25μsec = 200μsec
    - Corresponds to approximately 1600 assembly instructions at 8MHz
    - Bounds the length of interrupt service routines
    - Older versions of motes used a bit interface to the radio

RF Frequencies & Channels

- Industrial, scientific, and medical (ISM) bands
  - 868 to 870 in Europe and Asia
  - 902 to 928 MHz in US
  - 433.1 to 434.8 MHz in US and Europe
  - 313.9 to 316.1 MHz in Asia
- Other unregulated frequency bands
  - 2.4GHz (Bluetooth, 802.11b)
  - 5.8GHz (802.11a)
- Mote is manufactured for specific band
  - Discrete components on board set operating frequency

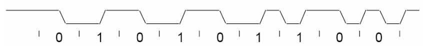
RF Modulation - FSK

- Frequency shift keying
  - One of many possible modulation schemes
  - 0 and 1 represented by two different frequencies slightly offset from a center carrier frequency (average)
  - At 38.4Kbps, ~10,000 periods for one bit



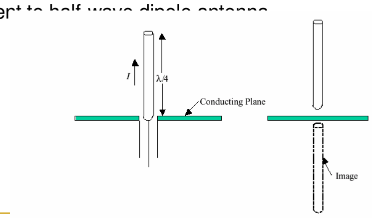
Data Encoding

- Manchester encoding
  - Every bit, whether a 0 or 1, has a transition
  - Guarantees there will never be a run of 0 or 1
  - Ensures stable clock recovery at receiver
  - Recovered clock determines sampling time of data bits
- Implemented in CC1000 hardware
  - Reduced ATMega128 overhead



Radio Antenna

- Simple ¼ wave monopole whip antenna is sufficient for most uses
  - 916Mhz → 3.2" wire length
  - 433Mhz → 6.8" wire length
- Equivalent to half wave dipole antenna

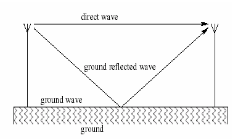


Antennas and Radio Transmission

- Polarization
  - Vertical orientation of all antennas in a system is best
  - 1/10th the distance if some antennas are vertical, some horizontal
- Transmission Near the Ground
  - Mica2 916Mhz, 3' above ground
    ~> 300' line of sight, 30' on the ground
  - Mica2 433 Mhz, 3' above ground
    ~>500' line of sight, 150' on the ground

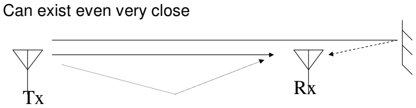
RF Propagation

- Line of sight
  - Direct path from transmitter to receiver
  - Free space attenuation $1/d^2$
    - To double distance, it needs 4x power
- Reflection
  - Off objects large compared to wavelength
    - walls, buildings
- Scattering
  - Off objects smaller than wavelength
    - foliage, chairs



Multi-path Effects

- Path length variations
  - Delayed version of signal arrives at receiver
- Path Attenuation
  - Various signal strengths
- Result looks like distortion or interference at receiver
- Out of phase signal interference can create <u>nulls</u>
  - Zero or severely reduced signal strength
  - Can exist even very close



Indoor Propagation

- Rapid signal attenuation closer to $1/d^3$
- People moving around cuts range by 1/3
- Concrete/steel flooring by 1/4
- Metallic tinted windows by 1/3

Dynamic Fading Effects

- People moving, doors opening and closing (esp. if metal)
  - E.g., closing doors in lab changes strong (green) RF regions to weak (blue) RF regions

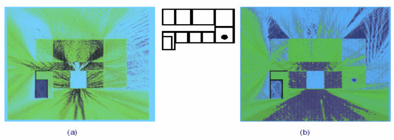


Fig. 4.1. Fluctuations of the electric field in the building that houses the faculty of Electrical Engineering. 2D FDTD code was used in these calculations which include the following: (a) all doors are open with no object present; (b) doors are closed with one object placed in one of the rooms. Green color denotes high electric field values; blue color represents low signal amplitudes.

Common RF Link Problems

- Signal strength
  - Weak, overload
- Collisions
  - Other motes (independent of GroupID)
- Interference from other sources
  - Cross-talk from adjacent RF channels
  - Other devices on same frequency
    - e.g., cordless phones at 900MHz
- Multi-path
  - Nulls are especially problematic as they can shift location

RF Link Metrics

- Packet loss
  - Determined at application layer (your code)
- Bit error rate
  - Determined at the link layer – incorrect checksums (TinyOS)
- Low RSSI – received signal strength indicator
  - Determined at the physical layer (CC1000)

RF Solutions for Signal Power & Wavelength

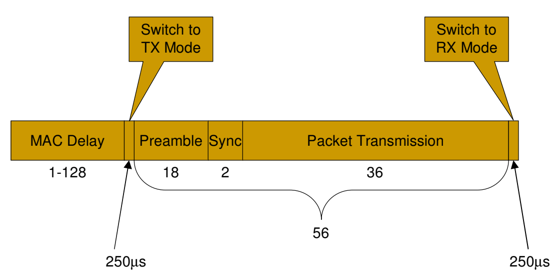
- Transmitter Power Level
- Antenna Efficiency
  - Antenna orientation
  - Antenna placement
    - Ground plane or off-ground
- RF Band Choice
  - Installation of 433MHz vs. 916MHz
- RF Channel selection default
  - Static vs. compile time
- Frequency Hopping
  - Dynamic / run time

Radio Data Packets in TinyOS

- Data transport – packetized data
  - 18 byte preamble – alternating 1010… pattern for clock recovery
  - 2 byte frame sync – indicates start of data packet
  - 36 bytes of TinyOS packet – data payload including CRC

noise	preamble	sync	data payload	noise

Data Packets on the Mica2 Platform



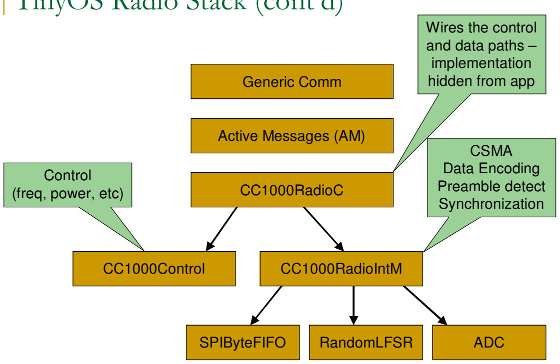
TinyOS Message Structure

- Header
  - Address (2bytes)
  - Active Message Type (1byte)
    - Indicates which handler to use to process message
  - Group ID (1 byte)
    - Adds to address space but provides a way to broadcast to a group
  - Payload Length (1 byte)
- Payload
  - 29 bytes user/application defined data
- CRC
  - 2 bytes

TinyOS Radio Stack

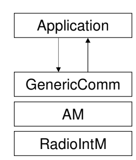
- Stack Layers
  - Application
  - Routing
  - Active Message
  - Packet
  - Byte
- Lowest-level radio interface
  - Data/noise streams in from CC1000 at 19.2Kb/s rate
  - SPI Input Interrupt every 416µsec (8bits)
  - CC1000 handles the serialization and physical layers

TinyOS Radio Stack (cont'd)



TinyOS Packet Reception

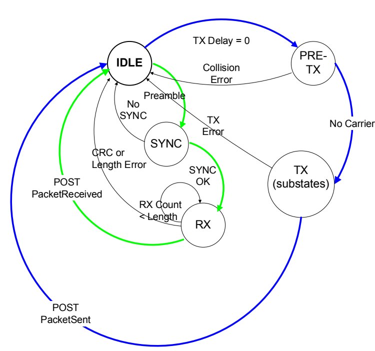
- RadioIntM.nc SPI Port Interrupt Handler
  - Search for preamble pattern (10101…)
  - Wait for frame sync word (two bytes)
- Assemble packet
- Check CRC – reject if bad
- Route to active message handler
- Check Group ID – reject if not member
- Signal spplication with ReceivedMsg event



TinyOS Packet Transmission

- Packet is routed
  - GenericComm
  - AM Handler – RF or UART
  - TinyOS CC1000RadioIntM
- Random delay (0-15 packet times)
- Check for carrier
- Turn on transmitter
- Send
  - 18-byte preamble (10101… pattern) & frame SYNC
  - Packet (34 bytes) – address, GroupID, type, length, and data payload
  - CRC (2 bytes)
- Turn off transmitter
- Signal TxDone event to application

CC1000Radio States



SYNC State

- Shift RX Byte bit-wise into Word Buffer
  - Word Buffer == SYNC PATTERN?
    - Byte Align = Bit Shift Count
    - Next State = RX
- RX Byte Count > MAX LENGTH ?
  - Next State = IDLE

Word Buffer (Previous RX Byte) Incoming RX Byte

10011001	11100110	←	01100001
00110011	11001100		

SYNC PATTERN

RX State

- RXBuffer[RXCount] = RX Data
- RXCount++
- Compute CRC(RXByte)
- RXCount == RXBuffer[Length]+Header?
- CRC = RXBuffer[CRC]?
  - Post PACKETRECEIVED
  - Next state = IDLE
- Error?
  - Next state = IDLE

TinyOS Radio Controls

- Frequency
  - Frequency Band / RF Channel Choices

```
#define CC1K_433_002_MHZ        0x00
```

  - Specify CC1K_DEFAULT_FREQ in makefile

```
CFLAGS -d:CC1K_DEFAULT_FREQ  CC1K_433_002_MHZ
```

- Power on/off
  - Sleep ~2µA
  - Radio signal strength (RSSI valid) ~20µsec
  - Receiver packet acquire time ~3msec
  - Re-tune radio after a power off/on cycle

```
command result_t Tune(uint8_t freq);
```

- RF Power Level
  - 0xFF is 5dBm
  - 0x80 is 0 dBm (1mW)
  - 0x09 is –10dBm

```
command result_t SetRFPower(uint8_t power);
```


Important RF Issues

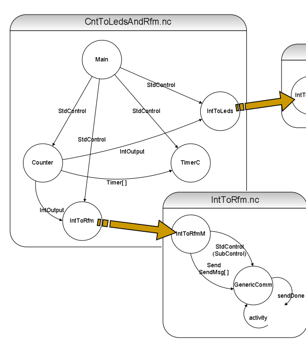
- Re-tune after sleep or temperature changes
- Remember multi-path effects can occur
- Different GroupIDs do NOT prevent RF interference
- Radio Debugging Hints
  - Correct Radio Frequency?
    - CC1K_DEFAULT_FREQ
  - Correct GroupID?
  - GenericBase Hangup
    - Press RESET button
  - RF Null Location?
    - Move mote to different location (+/- 1m)
  - RF Overload
    - Separation >2m

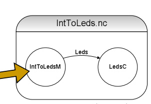
Example

- CntToLedsAndRfm
  - Display a number on LEDs
  - Send it to another mote over the radio

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl     -> Counter.StdControl;
  Main.StdControl     -> IntToLeds.StdControl;
  Main.StdControl     -> IntToRfm.StdControl;
  Main.StdControl     -> TimerC.StdControl;
  Counter.Timer       -> TimerC.Timer[unique("Timer")].Timer;
  IntToLeds.IntOutput <- Counter.IntOutput;
  Counter.IntOutput   -> IntToRfm.IntOutput;
}
```

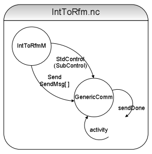

CntToLedsAndRfm Components






```
configuration IntToLeds
{
  provides interface IntOutput;
  provides interface StdControl;
}
implementation
{
  components IntToLedsM, LedsC;

  IntOutput = IntToLedsM.IntOutput;
  StdControl = IntToLedsM.StdControl;
  IntToLedsM.Leds -> LedsC.Leds;
}
```

IntToRfm.nc



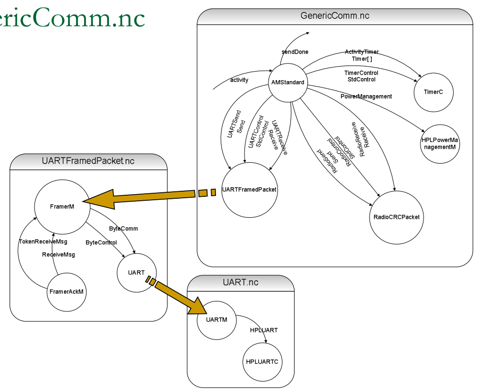

```
includes IntMsg;

configuration IntToRfm
{
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  components IntToRfmM, GenericComm as Comm;

  IntOutput = IntToRfmM;
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.SubControl -> Comm;
}
```


GenericComm.nc

```
configuration GenericComm
{
  provides {
    interface StdControl as Control;

    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    command uint16_t activity();
  }
  uses {
    event result_t sendDone();
  }
}
```

```
implementation
{
  components
    AMStandard,
    RadioCRCPacket as RadioPacket,
    UARTFramedPacket as UARTPacket,
    NoLeds as Leds,
    TimerC, HPLPowerManagementM;

  Control = AMStandard.Control;
  SendMsg = AMStandard.SendMsg;
  ReceiveMsg = AMStandard.ReceiveMsg;
  sendDone = AMStandard.sendDone;

  activity = AMStandard.activity;
  AMStandard.TimerControl -> TimerC.StdControl;
  AMStandard.ActivityTimer -> TimerC.Timer[unique("Timer")];

  AMStandard.UARTControl -> UARTPacket.Control;
  AMStandard.UARTSend -> UARTPacket.Send;
  AMStandard.UARTReceive -> UARTPacket.Receive;

  AMStandard.RadioControl -> RadioPacket.Control;
  AMStandard.RadioSend -> RadioPacket.Send;
  AMStandard.RadioReceive -> RadioPacket.Receive;
  AMStandard.PowerManagement -> HPLPowerManagementM.PowerManagement;

}
```


GenericComm.nc



UARTM.nc (abbreviated)

```
module UARTM {
  provides {
    interface ByteComm;
    interface StdControl as Control;
  }
  uses {
    interface HPLUART;
  }
}
implementation
{
  bool state;

  command result_t Control.init() {
  command result_t Control.start() {
  command result_t Control.stop() {
  async event result_t HPLUART.get(uint8_t data) {
  async event result_t HPLUART.putDone() {
  async command result_t ByteComm.txByte(uint8_t data) {

}
```


HPLUARTC.nc

```
configuration HPLUARTC {
  provides interface HPLUART as UART;
}
implementation
{
  components HPLUART0M;

  UART = HPLUART0M;

}
```

```
module HPLUART0M {
  provides interface HPLUART as UART;

}
implementation
{
  async command result_t UART.init() {

    // UART will run at:
    // 115kbps, N-8-1

    // Set 57.6 KBps
    outp(0,UBRR0H);
    outp(15, UBRR0L);

    // Set UART double speed
    outp((1<<U2X),UCSR0A);

    // Set frame format: 8 data-bits, 1 stop-bit
    outp(((1 << UCSZ1) | (1 << UCSZ0)) , UCSR0C);

    // Enable reciever and transmitter and their interrupts
    outp(((1 << RXCIE) | (1 << TXCIE) | (1 << RXEN) | (1 << TXEN)) ,UCSR0B);

    return SUCCESS;
  }

  async command result_t UART.stop() {
    outp(0x00, UCSR0A);
    outp(0x00, UCSR0B);
    outp(0x00, UCSR0C);
    return SUCCESS;
  }

  . . .
```


GenericComm.nc

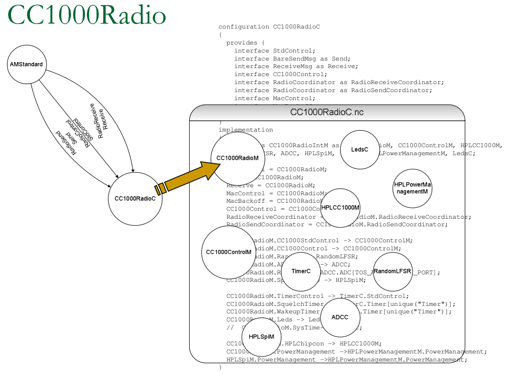
```
configuration GenericComm
{
  provides {
    interface StdControl as Control;

    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    command uint16_t activity();
  }
  uses {
    event result_t sendDone();
  }
}
```

```
implementation
{
  components
    AMStandard,
    RadioCRCPacket as RadioPacket,
    UARTFramedPacket as UARTPacket,
    NoLeds as Leds,
    TimerC, HPLPowerManagementM;

  Control = AMStandard.Control;
  SendMsg = AMStandard.SendMsg;
  ReceiveMsg = AMStandard.ReceiveMsg;
  sendDone = AMStandard.sendDone;

  activity = AMStandard.activity;
  AMStandard.TimerControl -> TimerC.StdControl;
  AMStandard.ActivityTimer -> TimerC.Timer[unique("Timer")];

  AMStandard.UARTControl -> UARTPacket.Control;
  AMStandard.UARTSend -> UARTPacket.Send;
  AMStandard.UARTReceive -> UARTPacket.Receive;

  AMStandard.RadioControl -> RadioPacket.Control;
  AMStandard.RadioSend -> RadioPacket.Send;
  AMStandard.RadioReceive -> RadioPacket.Receive;
  AMStandard.PowerManagement -> HPLPowerManagementM.PowerManage

}
```

in /platform/mica2

```
configuration RadioCRCPacket
{
  provides {
    interface StdControl as Control;
    interface BareSendMsg as Send;
    interface ReceiveMsg as Receive;
  }
}
implementation
{
  //components RadioCRCPacketM, RFCommC;
  components CC1000RadioC as RadioCRCPacketM;

  Control = RadioCRCPacketM;
  Send = RadioCRCPacketM.Send;
  Receive = RadioCRCPacketM.Receive;

  //RadioCRCPacketM.RFComm -> RFCommC;
}
```

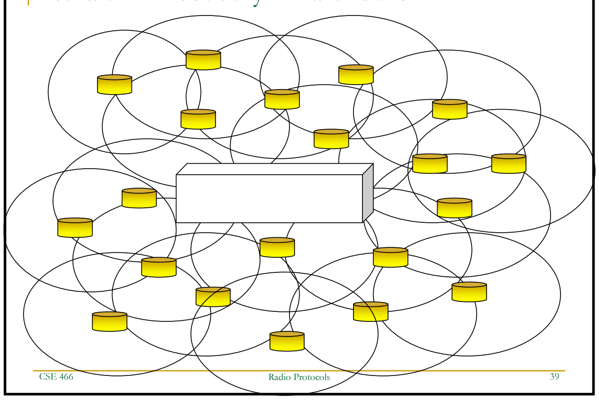
CC1000Radio



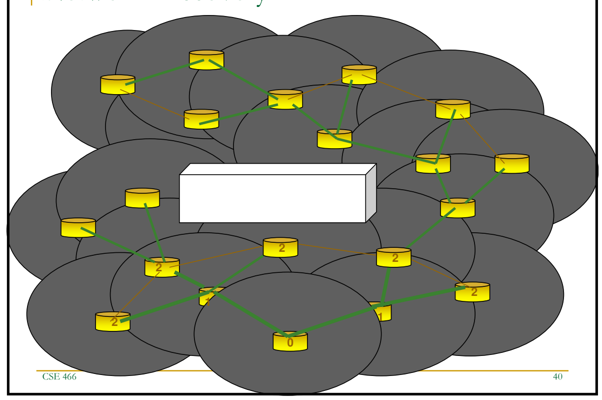
Another Application

- Surge
  - Forms a multi-hop ad-hoc network of nodes
  - Each node takes light readings and sends them to a base station
  - Each node also forwards messages of other nodes
  - Designed to be used in conjunction with the Surge java tool
  - The node also responds to broadcast commands from the base

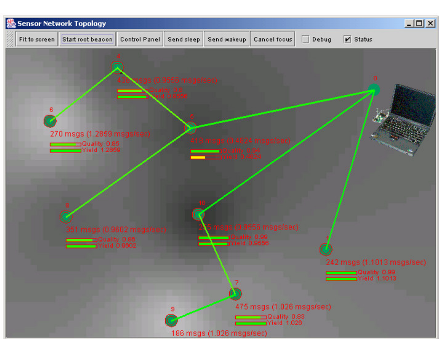
Network Discovery: Radio Cells



Network Discovery



Surge Multihop Routing



Surge.nc

```
configuration Surge {
}
implementation {
  components Main, SurgeM, TimerC, LedsC, Photo, RandomLFSR,
    GenericCommPromiscuous as Comm, Bcast,
    MultiHopRouter as multihopM, QueuedSend, Sounder;

  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> Photo;
  Main.StdControl -> Bcast.StdControl;
  Main.StdControl -> multihopM.StdControl;
  Main.StdControl -> QueuedSend.StdControl;
  Main.StdControl -> TimerC;
  Main.StdControl -> Comm;

  SurgeM.ADC      -> Photo;
  SurgeM.Timer    -> TimerC.Timer[unique("Timer")];
  SurgeM.Leds     -> LedsC;
  SurgeM.Sounder  -> Sounder;

  SurgeM.Bcast -> Bcast.Receive[AM_SURGECMDMSG];
  Bcast.ReceiveMsg[AM_SURGECMDMSG] -> Comm.ReceiveMsg[AM_SURGECMDMSG];

  SurgeM.RouteControl -> multihopM;
  SurgeM.Send -> multihopM.Send[AM_SURGEMSG];
  multihopM.ReceiveMsg[AM_SURGEMSG] -> Comm.ReceiveMsg[AM_SURGEMSG];
}
```

SurgeM.nc

```
  provides {
    interface StdControl;
  }
  uses {
    interface ADC;
    interface Timer;
    interface Leds;
    interface StdControl as Sounder;
    interface Send;
    interface Receive as Bcast;
    interface RouteControl;
  }
}
```

```
  command result_t StdControl.init()
    timer_rate = INITIAL_TIMER_RATE;
    atomic gfSendBusy = FALSE;
    sleeping = FALSE;
    rebroadcast_adc_packet = FALSE;
    focused = FALSE;
    return SUCCESS;
  }

  command result_t StdControl.start()
    return call Timer.start(TIMER_REP
    return SUCCESS;
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }
```

```
event result_t Timer.fired() {
  timer_ticks++;
  if (timer_ticks % TIMER_GETADC_COUNT == 0) {
    call ADC.getData();
  }
  // If we're the focused node, chirp
  if (focused && timer_ticks % TIMER_CHIRP_COUNT == 0) {
    call Sounder.start();
  }
  // If we're the focused node, chirp
  if (focused && timer_ticks % TIMER_CHIRP_COUNT == 1) {
    call Sounder.stop();
  }
  return SUCCESS;
}

async event result_t ADC.dataReady(uint16_t data) {
  atomic {
    if (!gfSendBusy) {
        gfSendBusy = TRUE;
        gSensorData = data;
        post SendData();
    }
  }
  return SUCCESS;
}
```


SurgeM.nc (cont'd)

```
task void SendData() {
  SurgeMsg *pReading;
  uint16_t Len;
  dbg(DBG_USR1, "SurgeM: Sending sensor reading\n");

  if (pReading = (SurgeMsg *)call Send.getBuffer(&gMsgBuffer,&Len)) {
    pReading->type = SURGE_TYPE_SENSORREADING;
    pReading->parentaddr = call RouteControl.getParent();
    pReading->reading = gSensorData;

    if ((call Send.send(&gMsgBuffer,sizeof(SurgeMsg))) != SUCCESS)
        atomic gfSendBusy = FALSE;
  }

}

event result_t Send.sendDone(TOS_MsgPtr pMsg, result_t success) {
  atomic gfSendBusy = FALSE;
  return SUCCESS;
}
```


SurgeM.nc (cont'd)

```
event TOS_MsgPtr Bcast.receive(TOS_MsgPtr pMsg, void* payload, uint16_t payloadLen) {
  SurgeCmdMsg *pCmdMsg = (SurgeCmdMsg *) payload;
  if (pCmdMsg->type == SURGE_TYPE_SETRATE) {       // Set timer rate
    timer_rate = pCmdMsg->args.newrate;
    call Timer.stop(); call Timer.start(TIMER_REPEAT, timer_rate);
  } else if (pCmdMsg->type == SURGE_TYPE_SLEEP) {
    sleeping = TRUE;
    call Timer.stop();
    call Leds.greenOff(); call Leds.yellowOff();
  } else if (pCmdMsg->type == SURGE_TYPE_WAKEUP) {
    if (sleeping) {
        initialize();
        call Timer.start(TIMER_REPEAT, timer_rate);
        sleeping = FALSE;
    }
  } else if (pCmdMsg->type == SURGE_TYPE_FOCUS) {
    if (pCmdMsg->args.focusaddr == TOS_LOCAL_ADDRESS) {
        focused = TRUE;
        call Sounder.init();
        call Timer.stop(); call Timer.start(TIMER_REPEAT, FOCUS_TIMER_RATE);
    } else {
        call Timer.stop(); call Timer.start(TIMER_REPEAT, FOCUS_NOTME_TIMER_RATE);
    }
  } else if (pCmdMsg->type == SURGE_TYPE_UNFOCUS) {
    focused = FALSE;
    call Sounder.stop();
    call Timer.stop(); call Timer.start(TIMER_REPEAT, timer_rate);
  }
  return pMsg;
}
```


TinyOS Active Messages (Sending)

- Sending using AMStandard
  - Get a region in memory for packet buffer
  - Form packet in the buffer
  - Assign active message type for proper handling
  - Request transmission
  - Handle completion signal

```
call SendMsg.send(TOS_BCAST_ADDR, 14, &data)
```


AMStandard.c (send)

```
 command result_t SendMsg.send[uint8_t id]
        (uint16_t addr, uint8_t length, TOS_MsgPtr data) {
   if (!state) {
     state = TRUE;
     if (length > DATA_LENGTH) {
       dbg(DBG_AM, "AM: Send length too long: %i. Fail.\n", (int)length);
       state = FALSE;
       return FAIL;
     } if (!(post sendTask())) {
       dbg(DBG_AM, "AM: post sendTask failed.\n");
       state = FALSE;
       return FAIL;
     } else {
       buffer = data;
       data->length = length;
       data->addr = addr;
       data->type = id;
       buffer->group = TOS_AM_GROUP;
       dbg(DBG_AM, "Sending message: %hx, %hhx\n\t", addr, id);
       dbgPacket(data);
     }
     return SUCCESS;
   }
   return FAIL;
}
```


TinyOS Active Messages (Receiving)

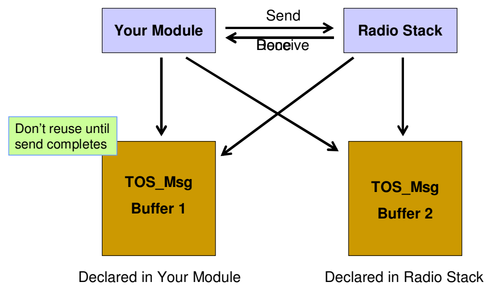
- Receiving using AMStandard
  - Declare a handler to perform action on message event
  - Active message automatically dispatched to associated handler
    - Known format
    - No run-time parsing
  - Buffer management
    - Must return free buffer to the system for the next packet reception
    - Typically the incoming buffer once processing is complete

```
TOS_MsgPtr buf;
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
 TOS_MsgPtr tmp;
 tmp = buf;
 buf = m;
 post receiveTask();
 return tmp;
}
```

AMStandard.c (receive)

```
TOS_MsgPtr received(TOS_MsgPtr packet) {
  uint16_t addr = TOS_LOCAL_ADDRESS;
  counter++;
  if (packet->crc == 1 &&
       packet->group == TOS_AM_GROUP &&
      (packet->addr == TOS_BCAST_ADDR ||
       packet->addr == addr))
    {
      uint8_t type = packet->type;
      TOS_MsgPtr tmp;
      tmp = signal ReceiveMsg.receive[type](packet);
      if (tmp) packet = tmp;
    }
  return packet;
}
default event TOS_MsgPtr ReceiveMsg.receive[uint8_t id](TOS_MsgPtr msg) {
  return msg;
}
event TOS_MsgPtr UARTReceive.receive(TOS_MsgPtr packet) {
  packet->group = TOS_AM_GROUP;
  return received(packet);
}
event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr packet) {
  return received(packet);
}
}
```


Packet Buffers



Platform Folder

- Location of details of the Hardware Layer
  - Most files have the HPL prefix
- Each type of platform has its own subfolder where platform specific files are pulled from.
  - (e.g. HPLUARTM, CC1000RadioC, HPLADCM)
- '.platform' file in platform directory
  - lists common platforms
  - allows compiler to pull from those platform directories second.
- 'hardware.h' is where the pins are mapped
- 'avrhardware.h" is where the macro's are defined

Pin Assignments

- Macros used to declare pins
  - `TOSH_ASSIGN_PIN(RED_LED, A, 2);`
- This gives a set of macro's that can be called
  - `TOSH_SET_RED_LED_PIN()`
  - `TOSH_CLR_RED_LED_PIN()`
  - `TOSH_MAKE_RED_LED_OUTPUT()`
  - `TOSH_MAKE_RED_LED_INPUT()`

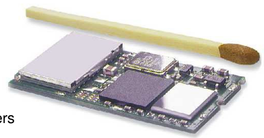
hardware.h

```
// LED assignments
TOSH_ASSIGN_PIN(RED_LED, A, 2);

// ChipCon control assignments
TOSH_ASSIGN_PIN(CC_CHP_OUT, E, 7);    // chipcon CHP_OUT
TOSH_ASSIGN_PIN(CC_PDATA, D, 7);      // chipcon PDATA
TOSH_ASSIGN_PIN(CC_PCLK, D, 6);       // chipcon PCLK
TOSH_ASSIGN_PIN(CC_PALE, D, 5);       // chipcon PALE

// PWM assignments
TOSH_ASSIGN_PIN(PWM1B, B, 6);
```

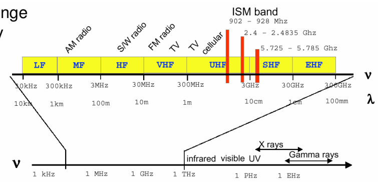

avrhardware.h

```
#define TOSH_ASSIGN_PIN(name, port, bit)
static inline void TOSH_SET_##name##_PIN() {sbi(PORT##port , bit);}
static inline void TOSH_CLR_##name##_PIN() {cbi(PORT##port , bit);}
static inline int TOSH_READ_##name##_PIN()
  {return (inp(PIN##port) & (1 << bit)) != 0;}
static inline void TOSH_MAKE_##name##_OUTPUT() {sbi(DDR##port , bit);}
static inline void TOSH_MAKE_##name##_INPUT() {cbi(DDR##port , bit);}

For

TOSH_ASSIGN_PIN(PWM1B, B, 6);

Yields:

static inline void TOSH_SET_PWM1B_PIN() {sbi(PORTB , 6);}
static inline void TOSH_CLR_PWM1B_PIN() {cbi(PORTB , 6);}
static inline int TOSH_READ_PWM1B_PIN()
  {return (inp(PINB) & (1 << 6)) != 0;}
static inline void TOSH_MAKE_PWM1B_OUTPUT() {sbi(DDRB , 6t);}
static inline void TOSH_MAKE_PWM1B_INPUT() {cbi(DDRB , 6);}
```

Bluetooth



- Short-range radio at 2.4GHz
  - Available globally for unlicensed users
  - Low-power
  - Low-cost
  - Cable replacement
  - Devices within 10m can share up to 1Mb/sec – 700Kb/sec effective
  - Universal short-range wireless capability



Bluetooth Application Areas

- Data and voice access points
  - Real-time voice and data transmissions
  - Cordless headsets
  - Three-in-one phones: cell, cordless, walkie-talkie
- Cable replacement
  - Eliminates need for numerous cable attachments for connection
  - Automatic synchronization when devices within range
- Ad hoc networking
  - Can establish connections between devices in range
  - Devices can “imprint” on each other so that authentication is not required for each instance of communication
  - Support for object exchange (files, calendar entries, business cards)

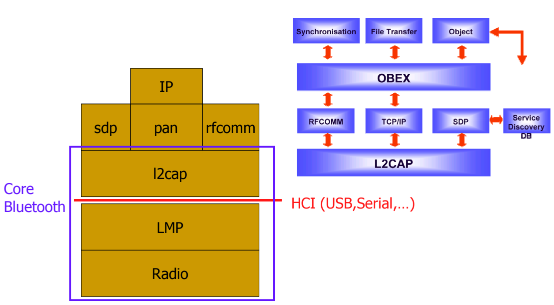
Bluetooth Standards Documents

- Core specifications
  - Details of various layers of Bluetooth protocol architecture
  - Emphasis on physical and transport layers
- Profile specifications
  - Use of Bluetooth technology to support various applications
  - Examples include point-to-point audio and local area network

Protocol Architecture

- Bluetooth is a layered protocol architecture
  - Core protocols
  - Cable replacement and telephony control protocols
  - Adopted protocols
- Core protocols
  - Radio
  - Baseband
  - Link manager protocol (LMP)
  - Logical link control and adaptation protocol (L2CAP)
  - Service discovery protocol (SDP)

Bluetooth Stack Overview



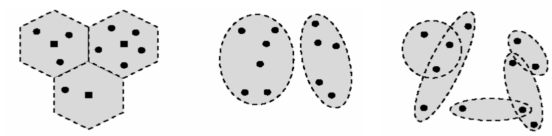
Protocol Architecture

- Cable replacement protocol
  - RFCOMM
- Telephony control protocol
  - Telephony control specification – binary (TCS BIN)
- Adopted protocols
  - PPP
  - TCP/UDP/IP
  - OBEX
  - WAP
- Profiles – vertical slide through the protocol stack
  - Basis of interoperability
  - Each device supports at least one profile
  - Defined based on usage models
    - e.g., headset, camera, personal server, etc.

Piconets and Scatternets

- Piconet
  - Basic unit of Bluetooth networking
  - Master and up to 7 slave devices
  - Master determines channel and phase
- Scatternet
  - Device in one piconet may exist as master or slave in another piconet
  - Allows many devices to share same area
  - Makes efficient use of bandwidth

Wireless Network Configurations



(a) Cellular system (squares represent stationary base stations) (b) Conventional ad hoc systems (c) Scatternets

Radio Specification

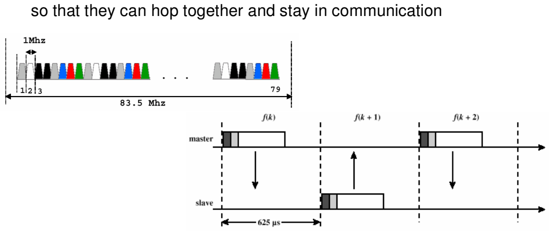
- Classes of transmitters
  - Class 1: Outputs 100 mW for maximum range
    - Power control mandatory
    - Provides greatest distance
  - Class 2: Outputs 2.4 mW at maximum
    - Power control optional
  - Class 3: Nominal output is 1 mW
    - Lowest power

Frequency Hopping in Bluetooth

- Provides resistance to interference and multipath effects
- Provides a form of multiple access among co-located devices in different piconets

Frequency Hopping

- Total bandwidth divided into 1MHz physical channels
- Frequency hopping occurs by moving transmitter/receiver from one channel to another in a pseudo-random sequence
- Hopping sequence shared with all devices in the same piconet so that they can hop together and stay in communication



Physical Links between Master - Slave

- Synchronous connection oriented (SCO)
  - Allocates fixed bandwidth between point-to-point connection of master and slave
  - Master maintains link using reserved slots
  - Master can support three simultaneous links
- Asynchronous connectionless (ACL)
  - Point-to-multipoint link between master and all slaves
  - Only single ACL link can exist

Bluetooth Packet Fields

- Access code
  - timing synchronization, offset compensation, paging, and inquiry
- Header
  - identify packet type and carry protocol control information
- Payload
  - contains user voice or data and payload header, if present

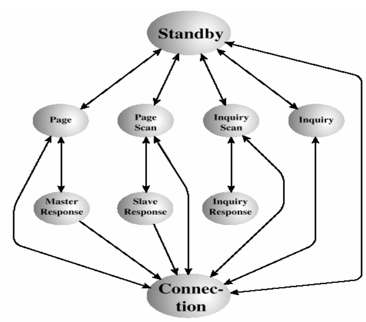
Channel Control

- States of operation of a piconet during link establishment and maintenance
- Major states
  - Standby – default state
  - Connection – device connected

Channel Control

- Interim substates for adding new slaves
  - Page – device issued a page (used by master)
  - Page scan – device is listening for a page
  - Master response – master receives a page response from slave
  - Slave response – slave responds to a page from master
  - Inquiry – device has issued an inquiry for identity of devices within range
  - Inquiry scan – device is listening for an inquiry
  - Inquiry response – device receives an inquiry response

State Transition Diagram



Scenario steps

- Master device (e.g., PDA) pages for nearby devices
- Receives response from 0, 1, or more devices
  - Slave device (e.g., headphone) responds to page
- Determines which it “knows” – established connections
- L2CAP establishes Bluetooth connection assigning paging device to be master
- Devices exchange profiles they both support
- Agree upon profile (e.g., audio streaming)
- Master sends audio data
  - Two devices synchronize their frequency hopping
- Keep-alive packets used to maintain connections
- Connections dropped if keep-alive packets are not acknowledged

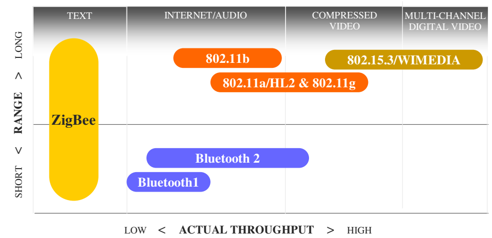
Limitations/Issues

- Discovery time on the order of 10sec for unknown devices
- Interaction with user required to connect to unknown devices or if multiple masters
- Can connect 8 devices at a time, more need to be multiplexed radically lowering throughput
- Doesn’t support simple broadcast – need to be on same frequency hopping schedule
- Effective bandwidth closer to 500Kbps (within one scatternet, order of magnitude lower if between two)

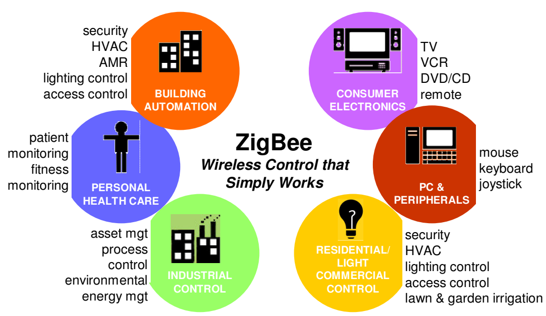
Zigbee (adapted from www.zigbee.org)

- Simpler protocol
- Broadcast support
- Network support (rather than point-to-point)
- Very low power (batteries that last years)
- Consumer device networks
  - Remote monitoring and control
  - Low-cost, low-complexity
  - Support ad-hoc and mesh networking
- Industry consortium
- Builds on IEEE standard 802.15.4 physical radio standard – OQSK encoding (offset quadrature phase shift keyed)
  - Adds logical network, security and application software
- 250Kb/sec bandwidth – 128Kb/sec effective, 30m range

The Wireless Market

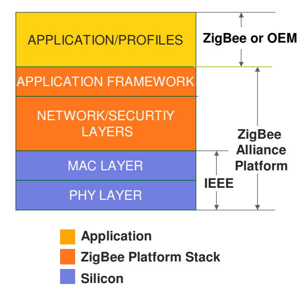


Applications



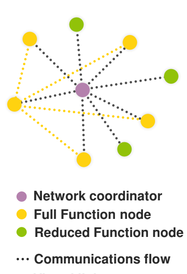
Protocol Stack Features

- 8-bit microcontroller
- Compact protocol stack <32KB
- Supports even simpler slave-only stack <4KB
- Coordinator requires extra memory for storing association tables

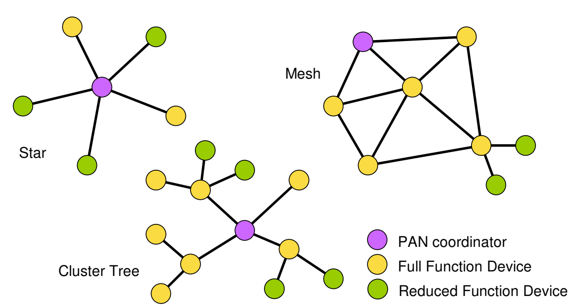


Zigbee Networks

- 65,536 network (client) nodes
- Optimized for timing-critical applications
  - Network join time: 30 ms (typ)
  - Sleeping slave changing to active: 15 ms (typ)
  - Active slave channel access time: 15 ms (typ)
- Traffic types
  - Periodic data (e.g., sensor)
  - Intermittent data, event (e.g., light switch)
  - Low-latency, slotted (e.g., mouse)



Zigbee Networks (cont'd)



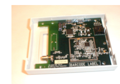
Lighting Control

[Philips Lighting]

- Advance Transformer
  - Wireless lighting control
    - Dimmable ballasts
    - Light switches anywhere
    - Customizable lighting schemes
    - Energy savings on bright days
    - Dali [or other] interface to BMS
  - Extendable networks
    - Additional sensors
    - Other networks

HVAC Energy Management

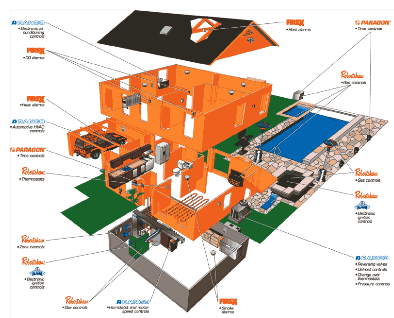
- Hotel energy management
  - Major operating expense for hotel
    - Centralized HVAC management allow hotel operator to make sure empty rooms are not cooled
  - Retrofit capabilities
  - Battery operated thermostats can be placed for convenience
  - Personalized room settings at check-in



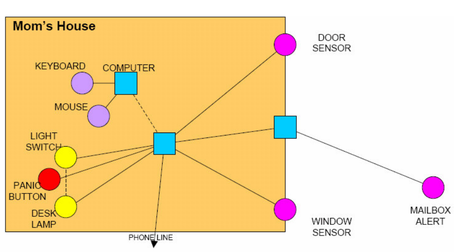
Asset Management

- Within each container, sensors form a mesh network.
- Multiple containers in a ship form a mesh to report sensor data
- Increased security through on-truck and on-ship tamper detection
- Faster container processing. Manifest data and sensor data are known before ship docks at port.

Residential Control



Residential Example



Comparison of Complementary Protocols

Feature(s)	IEEE 802.11b	Bluetooth	ZigBee
Power Profile	Hours	Days	Years
Complexity	Very Complex	Complex	Simple
Nodes/Master	32	7	64000
Latency	Enumeration upto 3 seconds	Enumeration upto 10 seconds	Enumeration 30ms
Range	100 m	10m	70m-300m
Extendability	Roaming possible	No	YES
Data Rate	11Mbps	1Mbps	250Kbps
Security	Authentication Service Set ID (SSID)	64 bit, 128 bit	128 bit AES and Application Layer user defined

HVAC control in building automation

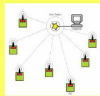
Wireless Network Evolution

Point to Point

- Simple wire replacement
- Direct Connection between devices
- Limited communication

Point to Multi-Point

- Centralized routing and control point
- Examples include: Wi-Fi, GSM, Bluetooth
- All data must flow through "base station"



Multi-hop/Mesh

- Full RF redundancy, with multiple data paths
- Self Configuring / Self Healing
- Distributed Intelligence