

# Agenda

- Introduction
- **TinyOS Programming**
- Power Issues
- Multi-hop routing (Surge)
- In-network reprogramming (XNP)
- TinyDB/TinySQL/TASK
- TinyOS Virtual Machine

# TinyOS Installation

- TinyOS can be downloaded from

<http://webs.cs.berkeley.edu/tos/>

- If you do a default installation, TinyOS will be at:

`C:\tinyos\cygwin\opt\tinyos-1.x`

We will refer to this directory as `<tos>`

# Directory Structure

- Within **<tos>** is:

```
/apps
  /OscilloscopeRF
/contrib
/doc
/tools
  /java
/tos
  /interfaces
  /lib
  /platform
    /mica
    /mica2
    /mica2dot
  /sensorboard
    /micasb
  /system
  /types
```

# Conveniences

- Aliases: Add to **C:\tinyos\cygwin\etc\profile**

```
alias cdjava="cd /opt/tinyos-1.x/tools/java"  
alias cdtos="cd /opt/tinyos-1.x"  
alias cdapps="cd /opt/tinyos-1.x/apps"
```

- Create **<tos>\apps\Makelocal** with the following inside:

```
PFLAGS += -DCC1K_DEF_FREQ=433002000  
#PFLAGS += -DCC1K_DEF_FREQ=868918800  
#PFLAGS += -DCC1K_DEF_FREQ=915998000  
#PFLAGS += -DCC1K_DEF_FREQ=914077000  
DEFAULT_LOCAL_GROUP=0x01  
MIB510=/dev/ttyS8
```

# TinyOS make command

- **From within the application's directory:**
- **make (re)install.<node id> <platform>**
  - <node id> is an integer between 0 and 255
  - <platform> may be mica2, mica2dot, or all
- **make clean**
- **make docs**
  - Generates documentation in <tos>\doc\nesdoc\mica2
- **make pc**
  - Generates an executable that can be run a pc for simulation

# Build Tool Chain

Convert NesC into C  
and compile to exec

Modify exec with  
platform-specific  
options

Set the mote ID

Reprogram the  
mote

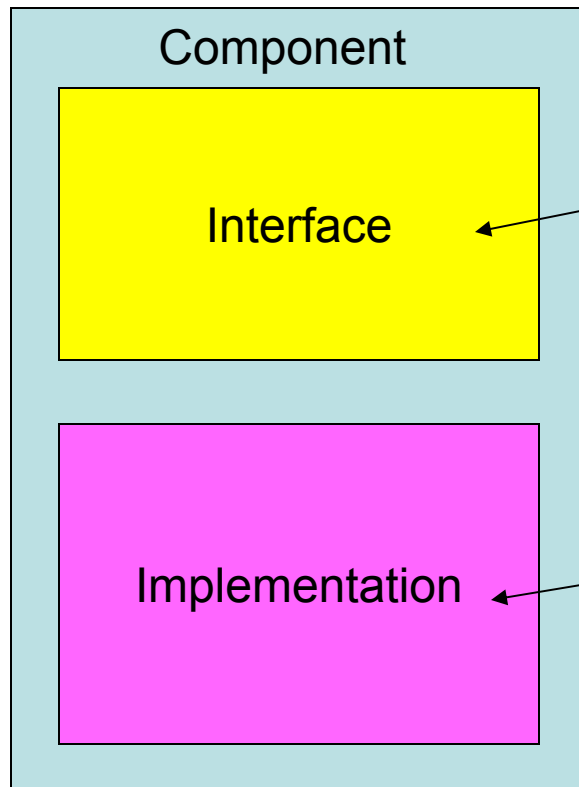
```
liang@pluto /opt/tinyos-1.x/apps/Blink
$ make install.0 mica2
   compiling Blink to a mica2 binary
ncc -o build/mica2/main.exe -Os -board=micasb -target=mica2 -DCC1K_DEF_FREQ=4330
02000 -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x01 -Wnesc-all -finline-limit=100000 -f
nesc-cfile=build/mica2/app.c Blink.nc -lm
   compiled Blink to build/mica2/main.exe
           1428 bytes in ROM
           44 bytes in RAM
avr-objcopy --output-target=srec build/mica2/main.exe build/mica2/main.srec
make mica2 reinstall.0 PROGRAMMER="STK" PROGRAMMER_FLAGS="-dprog=mib510 -dserial
=/dev/ttyS8 -dpart=ATmega128 --wr_fuse_e=ff "
make[11]: Entering directory `/opt/tinyos-1.x/apps/Blink'
   installing mica2 binary
set-mote-id build/mica2/main.srec build/mica2/main.srec.0.out `echo reinstall.0
|perl -pe 's/^reinstall.//; $_=hex if /^0x/i;'`
Could not find symbol TOS_LOCAL_ADDRESS in build/mica2/main.exe, ignoring symbol
.
uisp -dprog=mib510 -dserial=/dev/ttyS8 -dpart=ATmega128 --wr_fuse_e=ff --erase
--upload if=build/mica2/main.srec.0.out
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash

Fuse Extended Byte set to 0xff
make[11]: Leaving directory `/opt/tinyos-1.x/apps/Blink'

liang@pluto /opt/tinyos-1.x/apps/Blink
$
```

# Review of TinyOS Components

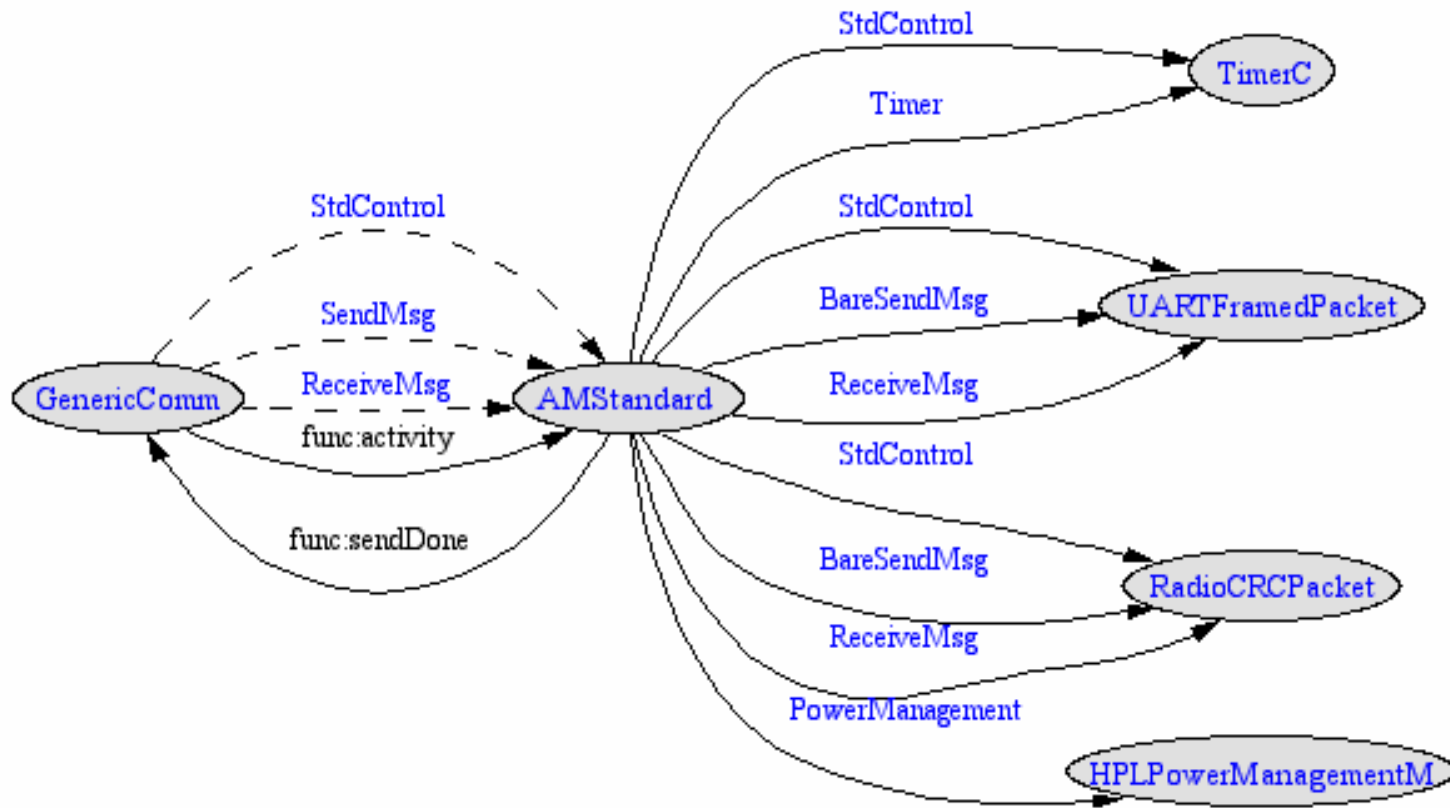
- A TinyOS program is composed of **components**



Declares the type of component (module or configuration), and the **interfaces, commands, and events** it uses and provides.

A module contains C-like code.  
A configuration contains wires.

# Example Components: GenericComm and AMStandard



This graphic is created using **make docs mica2** in  
<tos>/apps/OscilloscopeRF, which uses AT&T Graphviz

# Interface Syntax

- Look in <tos>\tos\interfaces\SendMsg.nc

```
includes AM; // includes AM.h located in <tos>\tos\types\  
  
interface SendMsg {  
    // send a message  
    command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg);  
  
    // an event indicating the previous message was sent  
    event result_t sendDone(TOS_MsgPtr msg, result_t success);  
}
```

- Multiple components may **provide** and **use** this interface
  - Specified by their interface
- A user can call commands on a provider, a provider can generate events which are handled by a user.

# Interface StdControl

- Look in `<tos>\tos\interfaces\StdControl.nc`

```
interface StdControl {  
  
    // Initialize the component and its subcomponents.  
    command result_t init();  
  
    // Start the component and its subcomponents.  
    command result_t start();  
  
    // Stop the component and pertinent subcomponents  
    command result_t stop();  
}
```

- Every component *\*should\** **provide** this interface
  - This is just good programming technique, it is not a language specification
  - Question: Who should call a component's `init()` method?

# Module Syntax (Interface)

- Look in <tos>\tos\system\AMStandard.nc

```
module AMStandard {
  provides {
    interface StdControl as Control;    // init(), start(), stop()
    interface SendMsg[uint8_t id];     // parameterized by AM ID
    command uint16_t activity(); // # of packets sent in past second
    ...
  }
  uses {
    event result_t sendDone();
    interface StdControl as UARTControl;
    ...
  }
  implementation {
    ...// code implementing all provided commands and used events
  }
}
```

Component  
Interface

# Module Syntax (Implementation)

```
module AMStandard {
  provides { interface SendMsg[uint8_t id]; ... }
  uses {event result_t sendDone(); ... }
}
implementation {
  task void sendTask() {
    ...
    signal sendDone(); signal SendMsg.SendDone(...);
  }
  command result_t SendMsg.send[uint8_t id](uint16_t addr,
    uint8_t length, TOS_MsgPtr data) {
    ...
    post sendTask(); // not shown: call UsedComponent.CmdName(...)
    ...
    return SUCCESS;
  }
  default event result_t sendDone() { return SUCCESS; }
}
```

# Async and Atomic

- Commands and events caused by a hardware interrupt must be declared with **async** keyword
  - E.g., `async command result_t Component.CmdName(...)`
- If a variable is accessed by both a task and **async** command or event, all accesses to it should be protected by **atomic{...}**
  - You can omit the atomic statement by declaring the shared variable with **norace** keyword (use at your own risk!!)
  - There are lots of examples of async and atomic usage in HPL\*.nc components in <tos>/tos/platform (e.g., HPLClock.nc)

# Configuration Syntax (Interface)

- Look in <tos>\tos\system\GenericComm.nc

```
configuration GenericComm {  
  provides {  
    interface StdControl as Control;  
    interface SendMsg[uint8_t id]; //parameterized by active message id  
    interface ReceiveMsg[uint8_t id];  
    command uint16_t activity();  
  }  
  uses { event result_t sendDone();}  
}  
implementation {  
  components AMStandard, RadioCRCPacket as RadioPacket, TimerC,  
    NoLeds as Leds, UARTFramedPacket as UARTPacket,  
    HPLPowerManagementM;  
  ... // code wiring the components together  
}
```

Component  
Interface

Component  
Selection

# Configuration Wires

- A configuration can bind an interface user to a provider using `->` or `<-`
  - `User.interface -> Provider.interface`
  - `Provider.interface <- User.interface`
- A component can handover responsibilities using `=`
  - `User1.interface = User2.interface`
  - `Provider1.interface = Provider2.interface`
- The interface may be implicit if there is no ambiguity
  - e.g., `User.interface -> Provider` is the same as `User.interface -> Provider.interface`

# Configuration Syntax (Wiring)

- Going back to `<tos>\tos\system\GenericComm.nc`

```
configuration GenericComm {
  provides {
    interface StdControl as Control;
    interface SendMsg[uint8_t id]; //parameterized by active message id
    command uint16_t activity(); ...
  }
  uses {event result_t sendDone(); ...}
}
implementation {
  components AMStandard, TimerC, ...;
  Control = AMStandard.Control;
  SendMsg = AMStandard.SendMsg;
  activity = AMStandard.activity;
  AMStandard.TimerControl -> TimerC.StdControl;
  AMStandard.ActivityTimer -> TimerC.Timer[unique("Timer")]; ...
}
```

# Fan-Out and Fan-In

- A user can be mapped to multiple providers (fan-out)
  - Open <tos>\apps\CntToLedsAndRfm\CntToLedsAndRfm.nc

```
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")];  
  IntToLeds <- Counter.IntOutput;  
  Counter.IntOutput -> IntToRfm;  
}
```

- A provider can be mapped to multiple users (fan-in)

# Top-Level Configuration

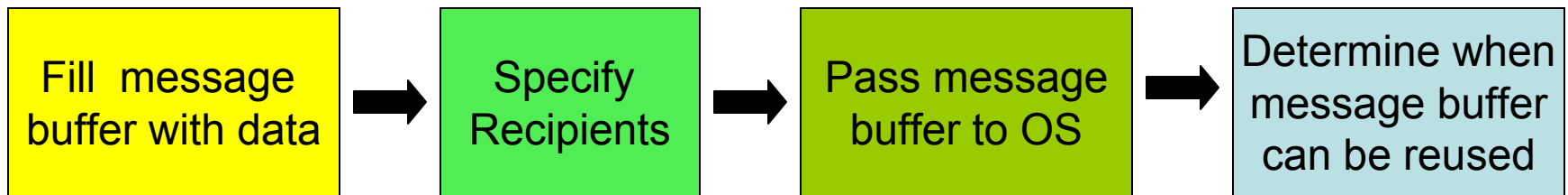
- All applications must contain a top-level configuration that uses **Main.StdControl**
  - This is the TOS Kernel
  - Open <tos>/apps/BlinkTask/BlinkTask.nc

```
configuration BlinkTask {  
  implementation {  
    components Main, BlinkTaskM, SingleTimer, LedsC;  
  
    Main.StdControl -> BlinkTaskM.StdControl;  
    Main.StdControl -> SingleTimer;  
  
    BlinkTaskM.Timer -> SingleTimer;  
    BlinkTaskM.Leds -> LedsC;  
  }  
}
```

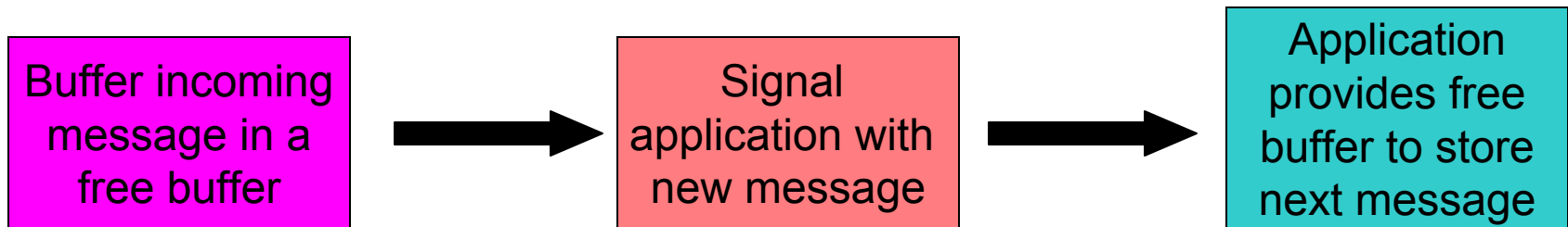
# Inter-Node Communication

- General idea:

- Sender:



- Receiver:



# Group IDs and Addresses

- Nodes can only communicate with other nodes in the same group ID
  - Group ID is an 8 bit value specified in **<tos>/apps/Makelocal**
  - **DEFAULT\_LOCAL\_GROUP=0x01**
- A node's address is a 16-bit value specified by the make command
  - make install.<id> mica2
  - <id> may be any 16-bit value except:
    - 0x007E - reserved for UART (TOS\_UART\_ADDR)
    - 0xFFFF - reserved for broadcast (TOS\_BCAST\_ADDR)

# TOS Active Messages

- TOS uses active messages as defined in `<tos>\system\types\AM.h`
- Message is “active” because it contains the destination address, group ID, and type (handler ID)
- `TOSH_DATA_LENGTH = 29 bytes`

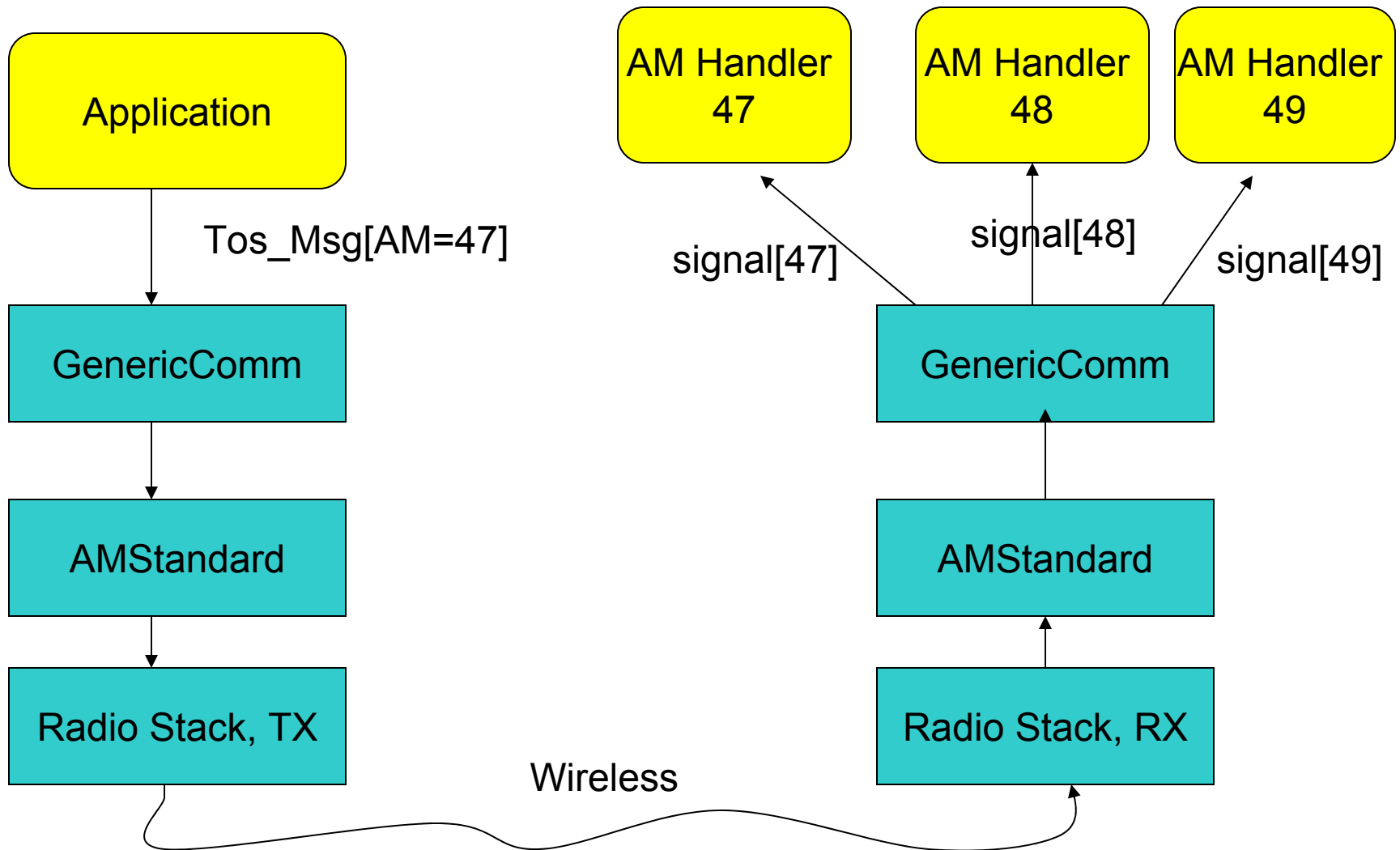
```
typedef struct TOS_Msg {  
    // the following are transmitted  
    uint16_t addr;  
    uint8_t type;  
    uint8_t group;  
    uint8_t length;  
    int8_t data[TOSH_DATA_LENGTH];  
    uint16_t crc;  
    // the following are not transmitted  
    uint16_t strength;  
    uint8_t ack;  
    uint16_t time;  
    uint8_t sendSecurityMode;  
    uint8_t receiveSecurityMode;  
} TOS_Msg;
```

Header (5)

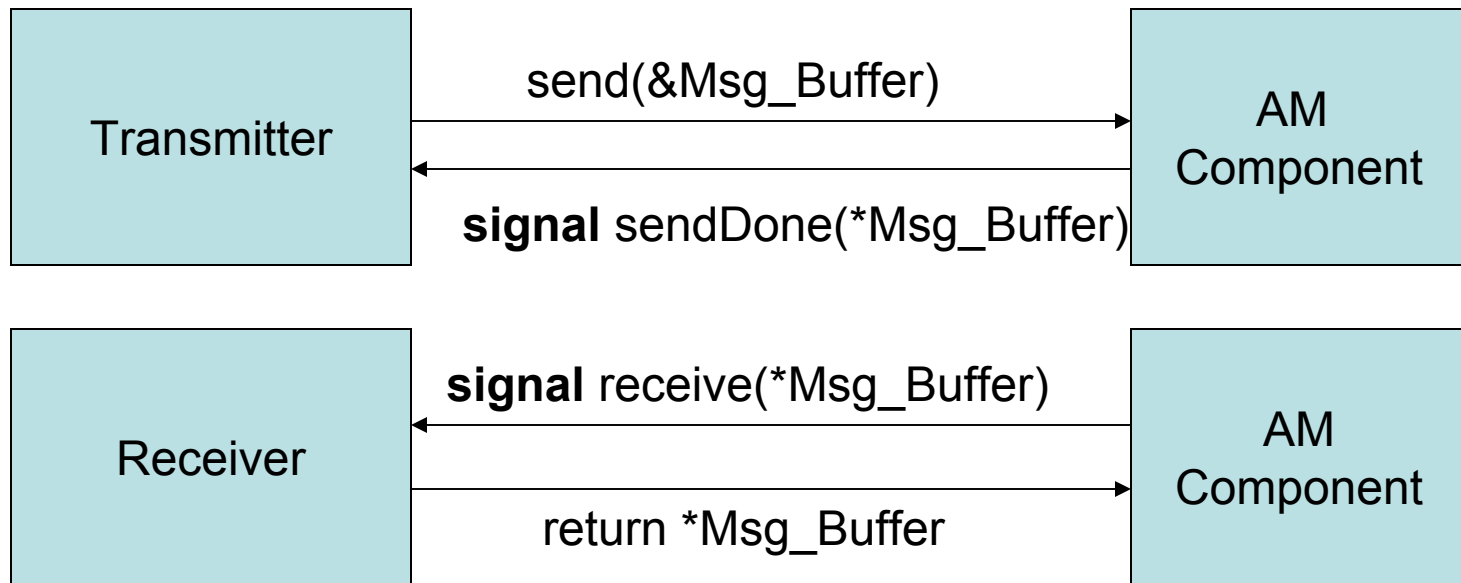
Payload (29)

CRC

# Active Messaging (Cont.)



# Message Buffer Ownership



- Transmission: AM gains ownership of the buffer until `sendDone(...)` is signaled
- Reception: Event handler on receiver gains ownership of the buffer, it must return another buffer to fill with the next message

# Sending a message (1 of 3)

- First define a .h file with a struct defining the message data format, and a unique active message number
  - Open <tos>/apps/Oscilloscope/OscopeMsg.h

```
struct OscopeMsg
{
    uint16_t sourceMotelID;
    uint16_t lastSampleNumber;
    uint16_t channel;
    uint16_t data[BUFFER_SIZE];
};
```

```
struct OscopeResetMsg
{
    /* Empty payload! */
};

enum {
    AM_OSCOPEMSG = 10,
    AM_OSCOPERESETMSG = 32
};
```

# Sending a Message (2 of 3)

```
module OscilloscopeM { ...
  uses interface SendMsg as DataMsg; ...
}
implementation{
  TOS_Msg msg; ...

  task void dataTask() {
    struct OscopeMsg *pack = (struct OscopeMsg *)msg.data;
    ... // fill up the message
    if (call DataMsg.send(TOS_BCAST_ADDR, sizeof(struct OscopeMsg),
                        &msg[currentMsg])) {...}
  }

  event result_t DataMsg.sendDone(TOS_MsgPtr sent, result_t success) {
    return SUCCESS;
  }
}
```

**Question:** How does TOS know the AM number?

Credit: © Crossbow Technologies January mote programming workshop

# Sending a Message (3 of 3)

- The AM number is determined by the configuration file
  - Open <tos>/apps/OscilloscopeRF/Oscilloscope.nc

```
configuration Oscilloscope { }  
implementation {  
  components Main, OscilloscopeM, GenericComm as Comm, ...;  
  ...  
  OscilloscopeM.DataMsg -> Comm.SendMsg[AM_OSCOPEMSG];  
}
```

# Receiving a Message

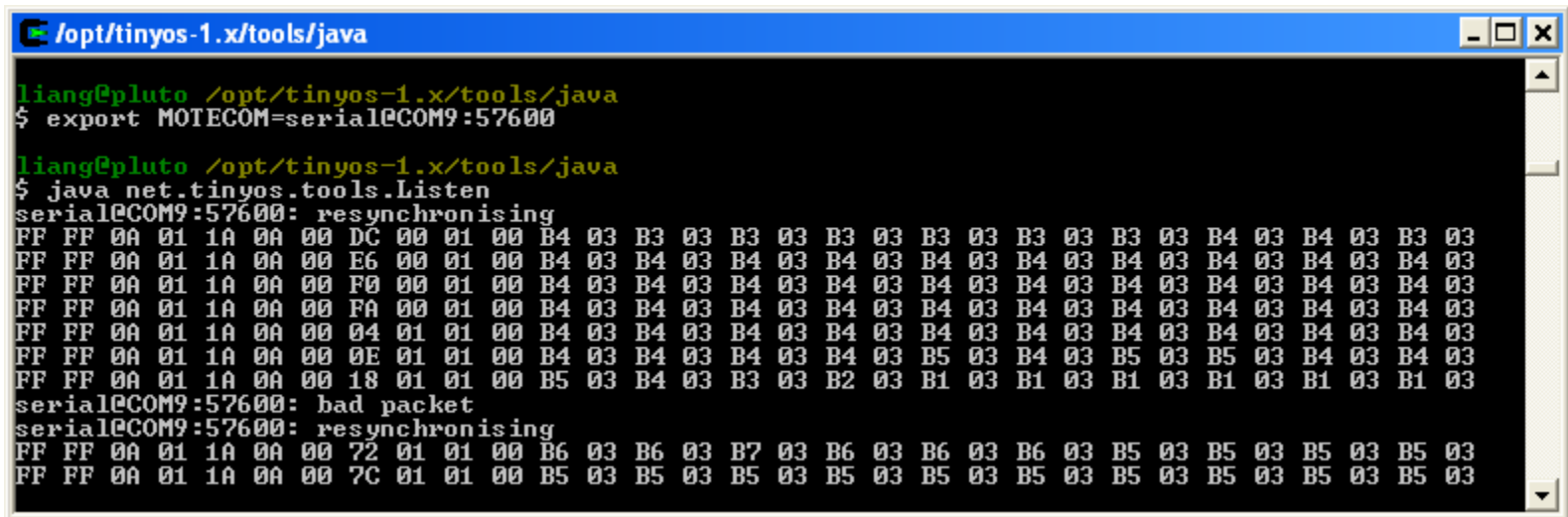
```
module OscilloscopeM {  
  uses interface ReceiveMsg as ResetCounterMsg; ...  
}  
implementation {  
  uint16_t readingNumber;  
  event TOS_MsgPtr ResetCounterMsg.receive(TOS_MsgPtr m) {  
    atomic { readingNumber = 0; }  
    return m;  
  }  
}  
  
configuration Oscilloscope { }  
implementation {  
  components Main, OscilloscopeM, UARTComm as Comm, ....;  
  ...  
  OscilloscopeM.ResetCounterMsg ->  
    Comm.ReceiveMsg[AM_OSCOPERESETMSG];  
}
```

# Sending Data to a Laptop

- A mote attached to the programming board can send data through the UART port to the laptop
- There are several applications that bridge between the wireless network and UART port
  - **<tos>/apps/TOSBase** – forwards messages with correct GroupID
  - **<tos>/apps/TransparentBase** – ignores GroupID
  - **<tos>/apps/GenericBase** – legacy support
- LED status:
  - Green = good packet received and forwarded to UART
  - Yellow = bad packet received (failed CRC)
  - Red = transmitted message from UART

# Displaying Received Data

- Java application: net.tinyos.tools.Listen
  - Located in `<tos>/tools/java/`
  - Relies on MOTECOM environment variable
    - Export MOTECOM=serial@COM9:57600



```
liang@pluto /opt/tinyos-1.x/tools/java
$ export MOTECOM=serial@COM9:57600

liang@pluto /opt/tinyos-1.x/tools/java
$ java net.tinyos.tools.Listen
serial@COM9:57600: resynchronising
FF FF 0A 01 1A 0A 00 DC 00 01 00 B4 03 B3 03 B3 03 B3 03 B3 03 B3 03 B4 03 B4 03 B3 03
FF FF 0A 01 1A 0A 00 E6 00 01 00 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03
FF FF 0A 01 1A 0A 00 F0 00 01 00 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03
FF FF 0A 01 1A 0A 00 FA 00 01 00 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03
FF FF 0A 01 1A 0A 00 04 01 01 00 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03 B4 03
FF FF 0A 01 1A 0A 00 0E 01 01 00 B4 03 B4 03 B4 03 B4 03 B5 03 B4 03 B5 03 B5 03 B4 03 B4 03
FF FF 0A 01 1A 0A 00 18 01 01 00 B5 03 B4 03 B3 03 B2 03 B1 03 B1 03 B1 03 B1 03 B1 03 B1 03
serial@COM9:57600: bad packet
serial@COM9:57600: resynchronising
FF FF 0A 01 1A 0A 00 72 01 01 00 B6 03 B6 03 B7 03 B6 03 B6 03 B6 03 B6 03 B5 03 B5 03 B5 03 B5 03
FF FF 0A 01 1A 0A 00 7C 01 01 00 B5 03 B5 03 B5 03 B5 03 B5 03 B5 03 B5 03 B5 03 B5 03 B5 03
```

5 byte header

OscopeMsg data payload (Big Endian)

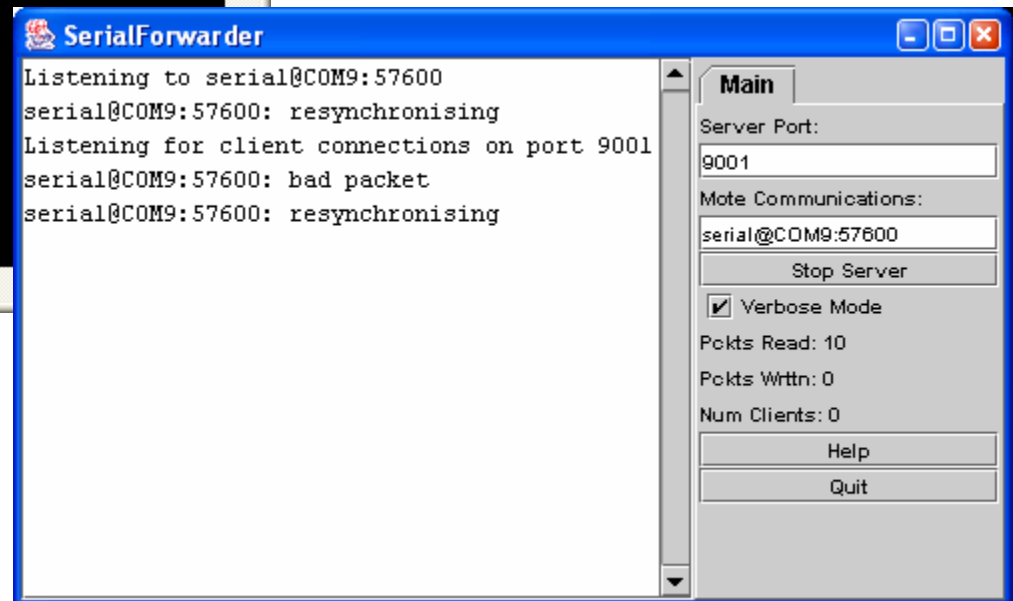
# Working with the Received Data

- TinyOS comes with a SerialPortForwarder that forwards UART packets to a local TCP socket
  - Allows multiple applications to perform I/O with the sensor network
  - Unset MOTECOM environment variable!

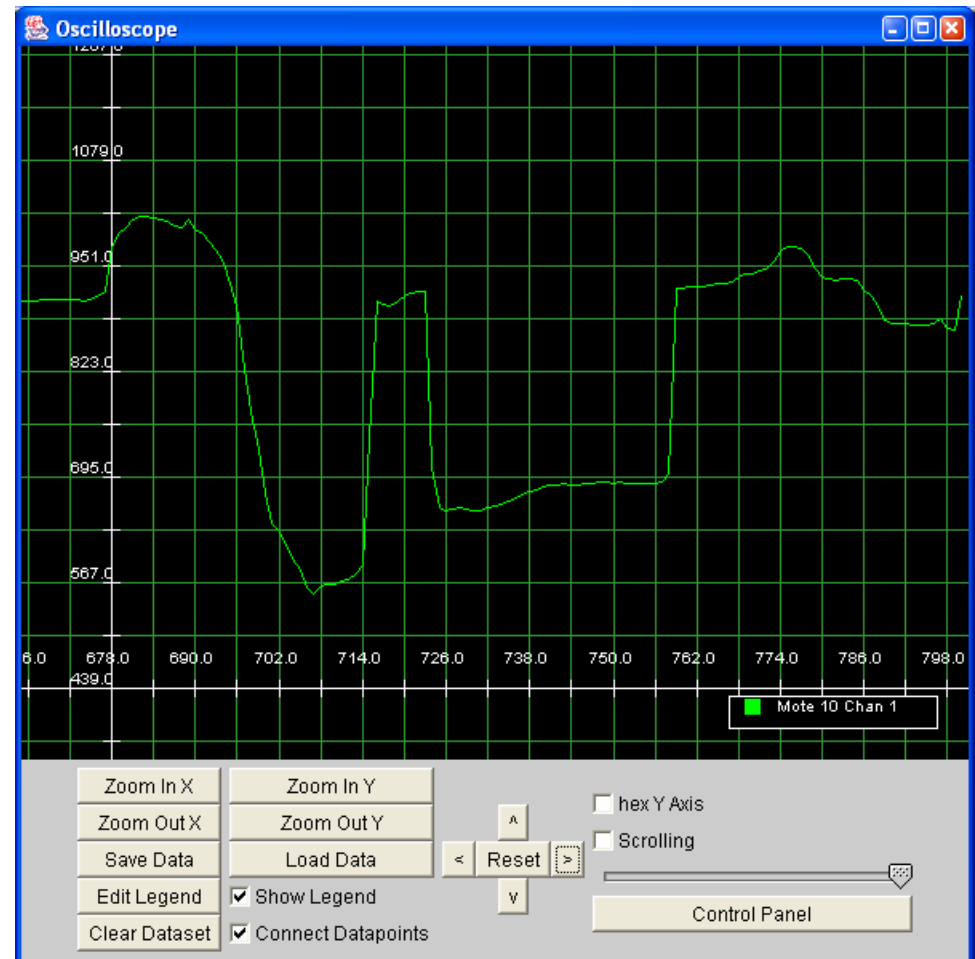
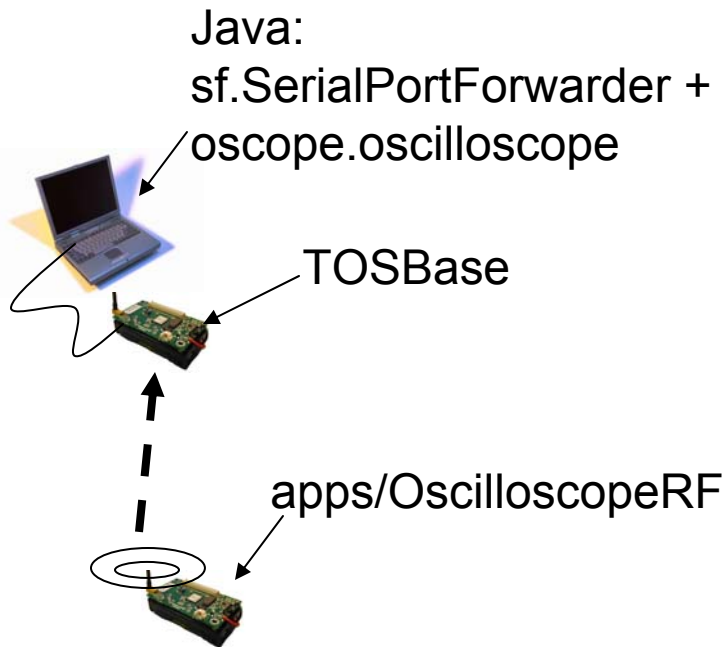
```
liang@pluto /opt/tinyos-1.x/tools/java
$ unset MOTECOM

liang@pluto /opt/tinyos-1.x/tools/java
$ java net/tinyos.sf.SerialForwarder -comm serial@COM9:57600 &
[1] 2052

liang@pluto /opt/tinyos-1.x/tools/java
$
```



# Java Applications w/ SPF



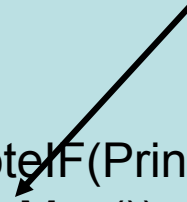
# Java Applications

- Class **net.tinyos.message.MoteIF** interfaces with the SerialPortForwarder TCP port
  - It uses **net.tinyos.message.Message** objects containing the message data

```
import net.tinyos.message.*;
import net.tinyos.util.*;
```

```
public class MyJavaApp {
    int group_id = 1;
    public MyJavaApp() {
        try {
            MoteIF mote = new MoteIF(PrintStreamMessenger.err, group_id);
            mote.send(new OscopeMsg());
        } catch (Exception e) {}
    }
}
```

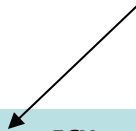
This must extend  
`net.tinyos.message.Message`,  
generated using  
`/usr/local/bin/mig`



# MIG

- Message Interface Generator
  - Generates a Java class representing a TOS message
  - Located in /usr/local/bin
  - Usage:

This is the generator as defined in /usr/local/lib/ncc/gen\*.pm



```
mig -java-classname=[classname] java [filename.h] [struct name] > outputFile
```

- Normally, you allow the Makefile to generate the Message classes

OscopeMsg.java:


```
$(MIG) -java-classname=$(PACKAGE).OscopeMsg \  
$(APP)/OscopeMsg.h OscopeMsg -o $@ \  
$(JAVAC) $@
```

# Obtaining Sensor Data

- Each sensor has a component that provides one or more ADC interfaces
  - MTS300CA:
    - components in `<tos>\tos\sensorboards\micasb`
    - Include in Makefile: **SENSORBOARD=micasb**
  - MTS400/420:
    - components in `<tos>\tos\sensorboards\micawb`
    - Include in Makefile: **SENSORBOARD=micawb**

```
includes ADC;
includes sensorboard; // this defines the user names for the ports

interface ADC {
  async command result_t getData();
  async command result_t getContinuousData();
  async event result_t dataReady(uint16_t data);
}
```



# Sensor Components

- Sensor components usually provide StdControl in addition to ADC
  - Be sure to initialize them before trying to take measurements!!
- Same goes with GenericComm
  - Initializing it turns on the power

```
module SenseLightToLogM {
  provides interface StdControl;
  uses {
    interface StdControl as PhotoControl;
  }
}

Implementation { ...
  command result_t StdControl.init() {
    return rcombine(call PhotoControl.init(),
                    call Leds.init());
  }
  command result_t StdControl.start() {
    return call PhotoControl.start();
  }
  command result_t StdControl.stop() {
    return call PhotoControl.stop();
  }
  ...
}
```

# Debugging Tips

- Join and/or search TOS mailing lists
  - <http://webs.cs.berkeley.edu/tos/support.html>
- Update TOS
  - Download latest CVS snapshot from <http://webs.cs.berkeley.edu/tos/download.html> and patch TOS (be sure to backup /opt)
- Develop apps in a private directory
  - (e.g., <tos>/broken)
- Debug with LEDs

# Debug with UART

- Include SODebug.h

- Copy from

- `C:\tinyos\cygwin\opt\tinyos-1.x\contrib\xbow\tos\interfaces`

- to

- `<tos>/tos/interfaces`

- Insert print statements into program

- `SODbg(DBG_USR2, "AccelM: setDone: state %i \n", state_accel);`

- Use any terminal program to read output from the serial port

# Debugging with JTAG

- Allows you to debug the system in real-time by stepping through the program and reading the register values from the processor
- Requires a \$300 Atmel JTAG pod

Questions?