Programming TinyOS

David Culler,
Phil Levis, Rob Szewczyk, Joe Polastre
University of California, Berkeley
Intel Research Berkeley

Available at
http://www.tinyos.net/mobisys2003.html
link entitled “Programming TinyOS”
Characteristics of Network Sensors

- Small physical size and low power consumption
- **Concurrency-intensive operation**
  - multiple flows, not wait-command-respond
- Limited Physical Parallelism and Controller Hierarchy
  - primitive direct-to-device interface
  - Asynchronous and synchronous devices
- Diversity in Design and Usage
  - application specific, not general purpose
  - huge device variation
  => efficient modularity
  => migration across HW/SW boundary
- Robust Operation
  - numerous, unattended, critical
  => narrow interfaces
A Operating System for Tiny Devices?

• Traditional approaches
  – command processing loop (wait request, act, respond)
  – monolithic event processing
  – bring full thread/socket posix regime to platform

• Alternative
  – provide framework for concurrency and modularity
  – never poll, never block
  – interleaving flows, events, energy management
=> allow appropriate abstractions to emerge
Tiny OS Concepts

- **Scheduler + Graph of Components**
  - constrained two-level scheduling model: threads + events

- **Component**:
  - Commands,
  - Event Handlers
  - Frame (storage)
  - Tasks (concurrency)

- **Constrained Storage Model**
  - frame per component, shared stack, no heap

- **Very lean multithreading**
- **Efficient Layering**
Application = Graph of Components

Example: ad hoc, multi-hop routing of photo sensor readings

3450 B code
226 B data

Graph of cooperating state machines on shared stack
TOS Execution Model

• commands request action
  – ack/nack at every boundary
  – call cmd or post task
• events notify occurrence
  – HW intrpt at lowest level
  – may signal events
  – call cmds
  – post tasks
• Tasks provide logical concurrency
  – preempted by events
• Migration of HW/SW boundary
Dynamics of Events and Threads

- bit event filtered at byte layer
- bit event =>
- end of byte =>
- end of packet =>
- end of msg send
- thread posted to start
- send next message
- radio takes clock events to detect recv
Programming TinyOS

• TinyOS 1.0 is written in an extension of C, called nesC
• Applications are too!
  – just additional components composed with the OS components
• Provides syntax for TinyOS concurrency and storage model
  – commands, events, tasks
  – local frame variable
• Rich Compositional Support
  – separation of definition and linkage
  – robustness through narrow interfaces and reuse
  – interpositioning
• Whole system analysis and optimization
Event-Driven Sensor Access Pattern

- clock event handler initiates data collection
- sensor signals data ready event
- data event handler calls output command
- device sleeps or handles other activity while waiting
- conservative send/ack at component boundary

```c
command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 200);
}

event result_t Timer.fired() {
    return call sensor.getData();
}

event result_t sensor.dataReady(uint16_t data) {
    display(data)
    return SUCCESS;
}
```
TinyOS Commands and Events

```c
{
    ...
    status = call CmdName(args)
    ...
}

command CmdName(args) {
    ...
    return status;
}

event EvtName(args) {
    ...
    return status;
}

{
    ...
    status = signal EvtName(args)
    ...
}
```
TinyOS Execution Contexts

- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks
- Both essential process state transitions
TASKS

• provide concurrency internal to a component
  – longer running operations
• are preempted by events
• able to perform operations beyond event context
• may call commands
• may signal events
• not preempted by tasks

```c
{  
  ...  
  post TskName();  
  ...
}
```

```c
task void TskName {  
  ...
}  
```
Typical application use of tasks

- event driven data acquisition
- schedule task to do computational portion

```c
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
}

task void processData() {
    int16_t i, sum=0;
    for (i=0; i < maxdata; i++)
        sum += (rdata[i] >> 7);
    display(sum >> shiftdata);
}
```

- 128 Hz sampling rate
- simple FIR filter
- dynamic software tuning for centering the magnetometer signal (1208 bytes)
- digital control of analog, not DSP
- ADC (196 bytes)
Tasks in low-level operation

• transmit packet
  – send command schedules task to calculate CRC
  – task initiated byte-level data pump
  – events keep the pump flowing

• receive packet
  – receive event schedules task to check CRC
  – task signals packet ready if OK

• byte-level tx/rx
  – task scheduled to encode/decode each complete byte
  – must take less time that byte data transfer

• i2c component
  – i2c bus has long suspensive operations
  – tasks used to create split-phase interface
  – events can procede during bus transactions
Example: Radio Byte Operation

- Pipelines transmission – transmits single byte while encoding next byte
- Trades 1 byte of buffering for easy deadline
- Separates high level latencies from low level real-time requirements
- Encoding Task must complete before byte transmission completes
- Decode must complete before next byte arrives

Hardware accelerators in MICA eliminate bit pumps
Task Scheduling

- Currently simple fifo scheduler
- Bounded number of pending tasks
- When idle, shuts down node except clock

- Uses non-blocking task queue data structure

- Simple event-driven structure + control over complete application/system graph
  - instead of complex task priorities and IPC
Tiny Active Messages

- **Sending**
  - Declare buffer storage in a frame
  - Request Transmission
  - Name a handler
  - Handle Completion signal

- **Receiving**
  - Declare a handler
  - Firing a handler
    » automatic
    » behaves like any other event

- **Buffer management**
  - strict ownership exchange
  - tx: done event => reuse
  - rx: must rtn a buffer
Sending a message

```c
bool pending;
struct TOS_Msg data;
command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)data.data;
    if (!pending) {
        pending = TRUE;
        message->val = value;
        message->src = TOS_LOCAL_ADDRESS;
        if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))
            return SUCCESS;
        pending = FALSE;
    }
    return FAIL;
}
```

- Refuses to accept command if buffer is still full or network refuses to accept send command
- User component provide structured msg storage
Send done event

```c
event result_t IntOutput.sendDone(TOS_MsgPtr msg, result_t success)
{
    if (pending && msg == &data) {
        pending = FALSE;
        signal IntOutput.outputComplete(success);
    }
    return SUCCESS;
}
```

- Send done event fans out to all potential senders
- Originator determined by match
  - free buffer on success, retry or fail on failure
- Others use the event to schedule pending communication
Receive Event

```c
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
    IntMsg *message = (IntMsg *)m->data;
    call IntOutput.output(message->val);
    return m;
}
```

- **Active message automatically dispatched to associated handler**
  - knows the format, no run-time parsing
  - performs action on message event

- **Must return free buffer to the system**
  - typically the incoming buffer if processing complete
Maintaining Scheduling Agility

• Need logical concurrency at many levels of the graph

• While meeting hard timing constraints
  – sample the radio in every bit window

⇒ Retain event-driven structure throughout application
⇒ Tasks extend processing outside event window
⇒ All operations are non-blocking
The Complete Application
Composition

- A component specifies a set of interfaces by which it is connected to other components
  - provides a set of interfaces to others
  - uses a set of interfaces provided by others

- Interfaces are bi-directional
  - include commands and events

- Interface methods are the external namespace of the component

```java
provides
  interface StdControl;
  interface Timer:
uses
  interface Clock
```
Components

- Modules
  - provide code that implements one or more interfaces and internal behavior

- Configurations
  - link together components to yield new component

- Interface
  - logically related set of commands and events

StdControl.nc

```plaintext
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
```

Clock.nc

```plaintext
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```
configuration SenseToRfm {
    // this module does not provide any interface
}
implementation {
    components Main, SenseToInt, IntToRfm, ClockC, Photo as Sensor;

    Main.StdControl -> SenseToInt;
    Main.StdControl -> IntToRfm;

    SenseToInt.Clock -> ClockC;
    SenseToInt.ADC -> Sensor;
    SenseToInt.ADCControl -> Sensor;
    SenseToInt.IntOutput -> IntToRfm;
}
Nested configuration

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;
}
includes IntMsg;

module IntToRfmM
{
  uses {
    interface StdControl as SubControl;
    interface SendMsg as Send;
  }
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  bool pending;
  struct TOS_Msg data;

  command result_t StdControl.init() {
    pending = FALSE;
    return call SubControl.init();
  }

  command result_t StdControl.start() {
    return call SubControl.start();
  }

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

  command result_t IntOutput.output(uint16_t value) {
    ... 
    if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data)
        return SUCCESS;
    ... 
  }

  event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
  {
    ... 
  }
A Multihop Routing Example
Sample Components

• Communication
  – Radio, UART, I2C of various flavors

• Timing
  – Timer, Clock

• Sensors
  – voltage, photo, light

• Busses
  – i2c, SPI

• Storage
  – eeprom, logger

• Energy management
  – snooze
Components => Services

- Multihop Routing
- Time synchronization
- Identity, Discovery
- ...

5/5/2003  MobiSys Tutorial, San Francisco
Supporting HW evolution

- Distribution broken into
  - apps: top-level applications
  - lib: shared application components
  - system: hardware independent system components
  - platform: hardware dependent system components
    » includes HPLs and hardware.h

- Component design so HW and SW look the same
  - example: temp component
    » may abstract particular channel of ADC on the microcontroller
    » may be a SW i2C protocol to a sensor board with digital sensor or ADC

- HW/SW boundary can move up and down with minimal changes
Scalable Simulation Environment

- target platform: TOSSIM
  - whole application compiled for host native instruction set
  - event-driven execution mapped into event-driven simulator machinery
  - storage model mapped to thousands of virtual nodes
- radio model and environmental model plugged in
  - bit-level fidelity
- Sockets = basestation
- Complete application
  - including GUI
Simulation Scaling

<table>
<thead>
<tr>
<th>Motes</th>
<th>Inactive</th>
<th>Quiet</th>
<th>Busy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01s</td>
<td>0.20s</td>
<td>0.22s</td>
</tr>
<tr>
<td>2</td>
<td>0.01s</td>
<td>0.43s</td>
<td>0.45s</td>
</tr>
<tr>
<td>4</td>
<td>0.02s</td>
<td>0.82s</td>
<td>0.93s</td>
</tr>
<tr>
<td>8</td>
<td>0.02s</td>
<td>1.70s</td>
<td>2.23s</td>
</tr>
<tr>
<td>16</td>
<td>0.03s</td>
<td>3.53s</td>
<td>5.79s</td>
</tr>
<tr>
<td>32</td>
<td>0.04s</td>
<td>7.45s</td>
<td>12.9s</td>
</tr>
<tr>
<td>64</td>
<td>0.06s</td>
<td>15.5s</td>
<td>27.1s</td>
</tr>
<tr>
<td>128</td>
<td>0.10s</td>
<td>32.3s</td>
<td>58.2s</td>
</tr>
<tr>
<td>256</td>
<td>0.23s</td>
<td>67.0s</td>
<td>122s</td>
</tr>
<tr>
<td>512</td>
<td>0.46s</td>
<td>140s</td>
<td>263s</td>
</tr>
<tr>
<td>1024</td>
<td>0.98s</td>
<td>290s</td>
<td>574s</td>
</tr>
<tr>
<td>2048</td>
<td>2.21s</td>
<td>665s</td>
<td>1,362s</td>
</tr>
<tr>
<td>4096</td>
<td>4.89s</td>
<td>1,699s</td>
<td>3,918s</td>
</tr>
<tr>
<td>8192</td>
<td>12.5s</td>
<td>3,794s</td>
<td>9,922s</td>
</tr>
</tbody>
</table>
Current Areas of Development

• Security
• Safe Concurrency
  – atomicity support
  – automatic race detection
• Abstract Components
Where to go for more?

- [http://www.tinyos.net/tos/](http://www.tinyos.net/tos/)


Contiki – an Operating System for Wireless Sensor Networks

Adam Dunkels
Swedish Institute of Computer Science
29 October 2004
Contiki

- Small operating system for tiny networked devices
  - Developed by Adam Dunkels

- Typical tiny device
  - 8-bit microcontroller
  - 1 kilobyte RAM, 10 kilobytes ROM (code)
  - Network: Radio, Ethernet, ...

- What? 8-bit?
Contiki

- Ported to ~20 platforms
  - MSP430, 6502, x86, AVR, Z80, PowerPC, ...
  - < 10 by Dunkels, rest by others

- Full **TCP/IP** support, provided by µIP
  - World's smallest full TCP/IP stack
Other cool features

- Graphical User Interface
  - Directly connected display
  - VNC
  - Telnet, RS232 terminal
- Mathias Bergvall, Enea
- World's smallest web browser
  - 2000 lines of code
  - Mozilla has 7000 files of code
“Tiny device” programming model

- Memory
  - Linear, no segmentation
  - No memory protection
  - Code in ROM
  - RAM much smaller than ROM (1-10%)

- Programming
  - Replace entire code memory
  - No boot loader, BIOS, OS, etc, just the raw CPU
Real-world example: ESB

- MSP430 CPU, 2 MHz
  - 60 k flash ROM (code)
  - 2 k RAM
- Download code
  - Plug into parallel port on PC
  - Download code using software on PC
- CPU supports self-reprogramming of ROM
  - Possible to do over-the-air reprogramming
Contiki – goals

- Portability
- Flexibility
  - Loadable application programs, device drivers
- Multitasking
- Networking (TCP/IP)
  - Event-driven interfaces
- Size
Specifics: challenges

- Constrained resources
  - Memory (less than 64 kilobytes)
  - CPU (less than 16 MHz)
- How do we do efficient multitasking?
  - Threads? Preemption?
  - Locking?
- Inter-process communication?
  - Over the network?
Multitasking in Contiki
Multitasking

- The traditional thread-style:
  - Multi-threading, preemption
  - Each thread needs its own stack
- Needs lots of memory
  - ... or only a few threads on a tiny device
- Locking problems
- Contiki: event-driven kernel
Contiki: event-driven kernel

- **Small**
  - Compact code (1k)
  - Very low memory requirements (a few bytes)
- **Multitasking**
  - Each process has an event handler
- **Inter-process communication**
  - Processes post events to each other
Event-driven: control flow

Start-up code: main.c

main()

Initialize

Kernel: ek.c

ek_run()

Event

App

Poll

Driver 1

Poll

Driver 2

Poll

Driver 3

Contiki – an OS for Sensor Networks – Adam Dunkels 29 October 2004
Programs as finite state machines

- Waiting
  - Command
  - No output
  - ACK received
  - Wait for ACK
  - No ACK received
  - Output sent
- Received
  - Output
- Sending
  - Output sent
Event-driven code

eventhandler1(event) {
    switch(event) {
        case INIT:
            initialize();
            return;
        case PACKET_RECEIVED:
            handle_packet();
            return;
        case EXIT:
            cleanup();
            return;
    }
}

eventhandler2(event) {
    switch(event) {
        case EVENT1:
            if(condition) {
                post(EVENT2, self);
            }
            return;
        case EVENT2:
            condition = 0;
            return;
        case EVENT3:
            condition = 1;
            return;
    }
}
Event-driven: drawbacks

- Hard to program
  - Programs **must** be implemented as state-machines
  - Modifications cumbersome
- Not suitable for all applications
  - Long running computations may be hard to discretize
- But Contiki also supports threads...
Concurrency in Contiki

- Contiki supports two additional concurrency models
  - Threads: preemptive / non-preemptive
  - Protothreads: stackless threads

- Threads make programming easier (in some cases)
  - Supports blocked waiting
Threads

- Implemented as application library
  - Applied on a per-application basis
- Scheduling on application level
  - Event handler schedules thread
- May be preempted
- Drawbacks
  - Requires stack to be allocated
  - Locking
Protothreads – stackless threads

- Extremely lightweight threads
  - Can only run within a single C function
  - Does not require any stack!
- Supports blocked waiting
- Scheduled within event handler, poll handler, TCP/IP handler, etc
- Similar to asymmetric coroutines
  - But more lightweight
Protothreads example

eventhandler(event)
{
    PT_BEGIN();
    PT_WAIT_UNTIL(event == PACKET_RECEIVED);
    send_reply();
    PT_WAIT_UNTIL(event == PACKET_TRANSMITTED);
    lock_cache();
    PT_WAIT_UNTIL(event == PACKET_RECEIVED);
    if(packet == ACK) release_cache();
    else post(PACKET_RECEIVED, self);
    PT_END();
}
Runtime partitioning
Runtime partitioning

- A running Contiki system is partitioned into two parts
  - Core
  - Programs
- Core always present in memory (ROM)
- Programs are loaded at runtime
Partitioning II

ROM

Programs

Core

RAM

Programs

Core
Core

- Core, always present in memory:
  - kernel
  - communication drivers
  - parts of language runtime (C library)
  - libraries
- In general not changed after deployment
  - But not impossible to do so
Programs

- Programs, loaded at runtime
  - Applications
  - May replace drivers, services in core
- Examples
  - Web server
  - Telnet server
  - Event logging
  - Log replicator
Runtime reprogramming
Reprogramming

- Programs can be loaded into RAM, ROM (if CPU supports rewriting flash ROM)
  - ESB does
  - Download code into EEPROM, burn into ROM

- Downloaded program starts process
  - May replace a running service
  - May even replace core
Over-the-air reprogramming

- Utilize the radio medium for programming
  - May be done in deployed networks
  - Very useful during development
- Download algorithms
  - Unicast
  - Multicast
Current algorithm

- Extremely simplistic unicast -> multicast
- Currently: lousy algorithm, poor implementation
  - Still quite useful
InceOS

Joe Sventek
Joseph.Sventek@glasgow.ac.uk
Overview

• Sensor network development challenges
• Insense
• InceOS - general
• InceOS - the interface
• Performance
• Summary
Sensor network development challenges

• Sensor networks are constructed from resource-constrained computing nodes that must operate unattended
• Thus, these sensor nodes are examples of embedded systems
• The traditional approach to development of embedded systems is to construct applications as state machines driven by events
• This manifests itself in TinyOS as split-phase programming of components
• This manifests itself in Contiki as stylized use of protothreads
Example - TinyOS

command result_t MyModule.sleep(uint16_t secs) {
    return call Timer.startOneShot(secs);
}

event result_t Timer.fired() {
    /* code to execute after sleeping for secs */
}

16 March 2011    InceOS
Example - Contiki

PT_BEGIN
startOneShotTimer(1000);
PT_WAIT_UNTIL(TIMER_FIRED)
/* code to execute */
PT_END
Embedded system development challenges

- Most programmers are unfamiliar with event-driven programming and state machines
- Even those that are familiar with these techniques are prone to making errors
- The unattended nature of embedded systems demands thorough testing, and if possible, use of formal verification techniques regarding the correctness of the programs before deployment
Insense

- Language modelled after the $\pi$ calculus
- Each component is an actor - its behaviour clause is executed as if in a for(;;) loop
- No shared state between components
- Components interact via messages over typed channels
- Message communication is blocking rendezvous - i.e. sender blocks until receiver receives or receiver blocks until sender sends
- The language also supports definition and invocation of non-recursive functions
Components have interfaces (channels they support), constructors, and a behaviour clause.

Components can send, receive, and select from bound channels.

Channels can be composed as 1:1, 1:many, many:1.

Language supports any type that can then be projected in a type-safe manner.

Dynamic allocation of types is supported via new keyword, and heap data is garbage collected.

Blocking nature of send/receive/select requires provision of a stack for each component.
InceOS - general

- Insense compiler generates C
- First version targeted code and a runtime built over Contiki
- Discovered that Contiki and the runtime consumed most of the RAM and ROM in a TelosB mote (10kB RAM, 48 kB ROM)
- Decided to build a domain-specific operating system to support Insense programs - InceOS - to overcome this memory bloat
InceOS - general (2)

• Abstractions supported by InceOS
  - Components
  - Channels
  - Timer
  - Radio
  - Sensors

• Takes advantage of the compiler’s static analysis of Insense programs to allocate the minimally-sized stack to each component

• Provides access to Timer/Radio/Sensors through well-known functions
InceOS – general (3)

- It is assumed that the Insense compiler has done all of the type checking to ensure that the channels are used correctly. The kernel has no need to understand the types of messages sent over bound channels; its primary job is to implement the appropriate rendezvous and scheduling semantics as defined previously. The compiler may generate complex structures to represent such messages; all that is required is that the sender and receiver components know how to refer to that structure, with the kernel guaranteeing that each such structure is delivered intact. Therefore, the messaging system calls are specified in terms of an array of bytes.
InceOS - the interface

/* component_create()
 * if successful, returns allocated data structure representing
 * the component
 * if unsuccessful, returns NULL
 *
 * when the component is eventually scheduled, it is called using
 * its own stack as follows: behaviour(void *this);
 * both constructor() and behaviour() must cast `this' to the
 * appropriate type upon entry, as follows:
 *   FooStruct *me = (FooStruct *)this;
 */

void *component_create(void (*constructor)(),
            void (*behaviour)(),
            int struct_size, int stack_size,
            int argc, void *argv[]);
InceOS - the interface

/*
 * currently running component terminates itself
 * there is no return; after destroying the calling
 * component, the scheduler schedules ready to run component
 */

void component_exit();

/*
 * instruct the component referenced by this_ptr terminate
 * itself the next time it tests its loop condition with
 * component_exit()
 * there is no return;
 */

void component_stop(void *this_ptr);
InceOS - the interface

/*@ 
* current component will relinquish control of the processor
* it remains eligible to execute and is added to the scheduler
* the scheduler will then schedule the next ready component
* there is no return;
*/

void component_yield();

typedef int channel;

//
* create a channel of the indicated type, returning its
* identifier as the value of the function, if successful
* the channel is initially unbound
* return -1 if unsuccessful
*/

enum direction {IN, OUT};

channel channel_create(enum direction d);
InceOS - the interface

/*
 * gracefully destroy the indicated channel
 * no return value
 */
void channel_destroy(channel id);

/*
 * bind channel 1 to channel 2
 * returns >= 0 if successful or -1 if unsuccessful
 * (one or both are unknown channels, same polarity, ...)
 */
int channel_bind(channel id1, channel id2);

/*
 * unbind channel from its peer
 * no return value
 */
void channel_unbind(channel id);
InceOS - the interface

/*
 * send data over channel
 * returns >= 0 (number of sent bytes) if successful
 * or -1 if unsuccessful
 *
 * id: the channel to send the data on
 * buffer: a pointer to the data to be sent
 * len: length of the buffer
 */

int channel_send(channel id, void *buffer, int len);
InceOS - the interface

/*
 * receive data over channel from its peer
 * returns number of bytes placed in buffer if successful
 * returns -1 if unsuccessful OR if buffer too small
 *
 * id:     the channel to receive the data on
 * buffer: a pointer to the space for incoming data
 * len:    length of the buffer
 */

int channel_receive(channel id, void *buffer, int len);
struct select_struct {
    int nchans;    /* number of channels */
    channel *chans; /* list of channels */
    /* bitmask for the when clauses and default value, read from
     * right to left with 1 indicating true. Default is the
     * leftmost value
    */
    int whensANDdef;
    void *buffer;    /* ptr to the receiving buffer */
    int size;    /* size of this buffer */
    int *len;    /* on return will contain the number of bytes read */
};
InceOS - the interface

/*
 * implements the select statement
 * if one or more arms are eligible, copies the message * from the chosen
 arm into buffer, writes the actual * number of bytes copied into *len,
 and returns the index
 * (0 .. nchans-1) of the chosen arm
 * if error encountered, returns -1
 * if no arms are eligible, two possible actions:
 * if (if_default) returns nchans
 * else blocks until one of the arms is eligible
 */

int channel_select(struct select *s);
/* channel_duplicate() */

* duplicate the indicated channel
* 
* direction of the channel is determined from that of `id'
* if `id' is bound, the new channel returned is also bound to the
* same set of channels; if unbound, the new channel is unbound
* 
* return value is duplicate channel identifier if successful, -1
* otherwise
*/

channel channel_duplicate(channel id);
/* channel_adopt()
 *
 * adopt channel - set its owner to the currently running component
 *
 * this is used when a channel is sent between components over a
 * bound channel the receiving component adopts the channel after
 * successfully receiving it over the bound channel
 *
 * returns >= 0 if successful
 * returns -1 if unsuccessful (channel unknown)
 */

int channel_adopt(channel id);
Individual applications

- **countToLeds** involves two motes, one to maintain a counter which is transmitted over the radio to the other mote, which displays the lower three bits of the transmitted value on its LED's.
- **senseToLeds** is a similar application, except it collects and sends sensor data as opposed to a software counter.
- **sense** is similar to senseToLeds, except it uses a single mote and does not use the radio.
- **fourier** performs a fourier transform of an array of 40 integers repeatedly.
- **grid** broadcasts a request to all nodes; sends an array of 10 integers to the first respondent to perform a fourier transform, then receives the max value.
- **Testroundrobinarbiter** is an example of an access control mechanism where three resource users request access from a central controller, which grants access to each in turn.
## Code composition

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines of Code</th>
<th>Components</th>
<th>Wiring Statements</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insense</td>
<td>nesC</td>
<td>Insense</td>
<td>nesC</td>
</tr>
<tr>
<td>BlinkA</td>
<td>28</td>
<td>40</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>BlinkB</td>
<td>30</td>
<td>54</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>TestSineSensor</td>
<td>13</td>
<td>45</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>RadioStress</td>
<td>50</td>
<td>94</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>55</td>
<td>103</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>57</td>
<td>101</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Sense</td>
<td>29</td>
<td>43</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>TestRoundRobinArbiter</td>
<td>49</td>
<td>180</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Fourier</td>
<td>19</td>
<td>30</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Grid</td>
<td>97</td>
<td>177</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
Application sizes

Figure 5: Space consumed by InceOS applications.

Figure 6: Space consumed by TinyOS applications.
Performance

RadioCountToLeds Application

Time (s)

Experiment Number

RadioSenseToLeds Application

Time (s)

Experiment Number

Sense Application

Time (s)
Performance (2)

- Fourier Application
- Test Round Robin Arbiter Application
- Grid Application
Conclusions

• It is possible to write functionally-equivalent programs in Insense with many fewer lines of code
• InceOS applications consistently outperform TinyOS for all applications in our test suite
• Insense/InceOS systems consume more RAM and Flash memory to provide this performance; a substantial fraction of each resource still remains for use by applications
• More familiar programming style achieved with memory to spare