CSEP567– tonight:

TinyOS

TinyOS

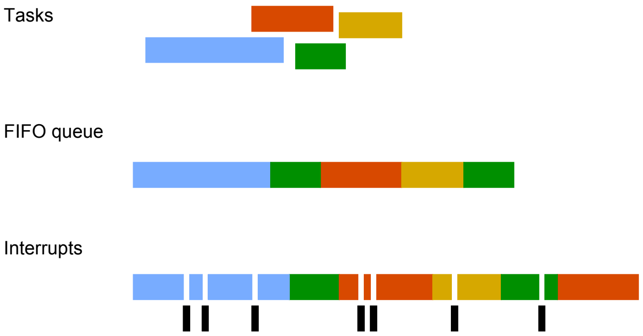
- Open-source development environment
- Simple (and tiny) operating system – TinyOS
- Programming language and model – nesC
- Set of services

- Principal elements
  - Scheduler/event model of concurrency
  - Software components for efficient modularity
  - Software encapsulation for resources of sensor networks

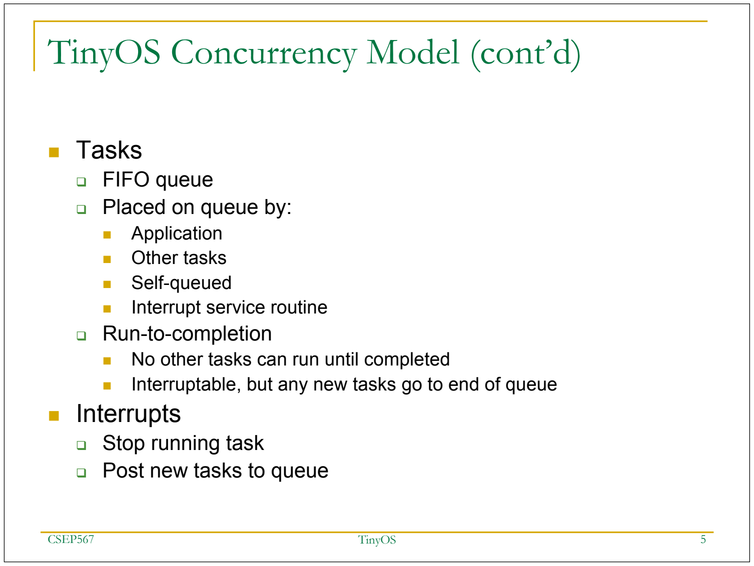
TinyOS Kernel Design

- Two-level scheduling structure
  - Events
    - Small amount of processing to be done in a timely manner
    - E.g. timer, ADC interrupts
    - Can interrupt longer running tasks
  - Tasks
    - Not time critical
    - Larger amount of processing
    - E.g. computing the average of a set of readings in an array
    - Run to completion with respect to other tasks
      - Only need a single stack

TinyOS Concurrency Model

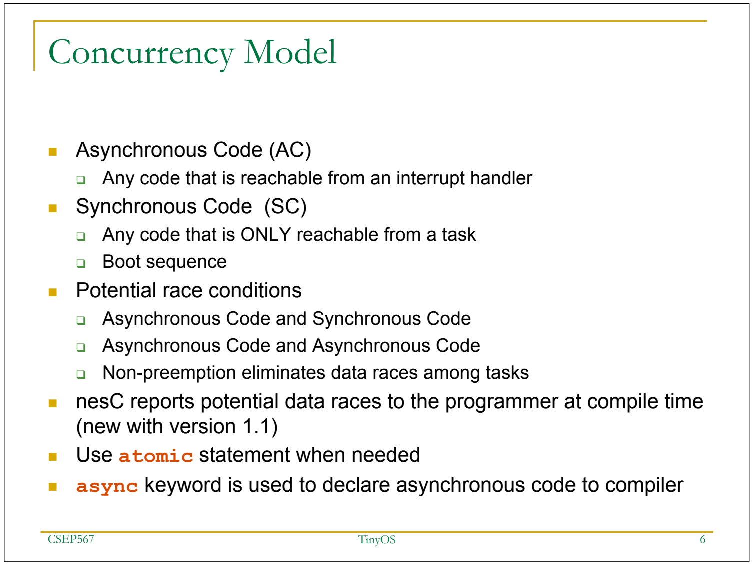


Two-level of concurrency: tasks and interrupts



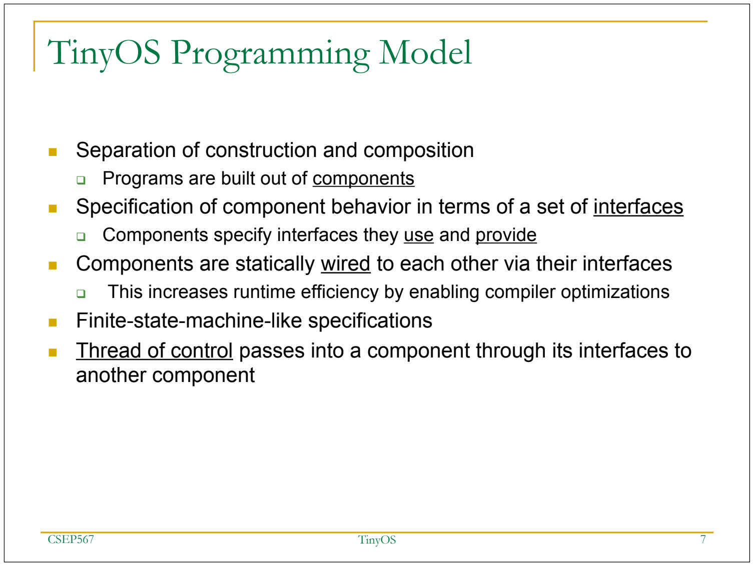
TinyOS Concurrency Model (cont'd)

- Tasks
  - FIFO queue
  - Placed on queue by:
    - Application
    - Other tasks
    - Self-queued
    - Interrupt service routine
  - Run-to-completion
    - No other tasks can run until completed
    - Interruptable, but any new tasks go to end of queue
- Interrupts
  - Stop running task
  - Post new tasks to queue



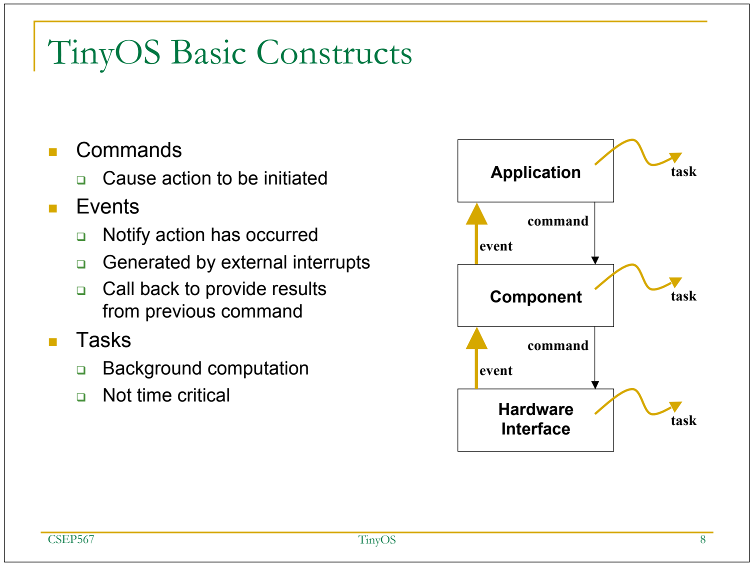
Concurrency Model

- Asynchronous Code (AC)
  - Any code that is reachable from an interrupt handler
- Synchronous Code (SC)
  - Any code that is ONLY reachable from a task
  - Boot sequence
- Potential race conditions
  - Asynchronous Code and Synchronous Code
  - Asynchronous Code and Asynchronous Code
  - Non-preemption eliminates data races among tasks
- nesC reports potential data races to the programmer at compile time (new with version 1.1)
- Use `atomic` statement when needed
- `async` keyword is used to declare asynchronous code to compiler



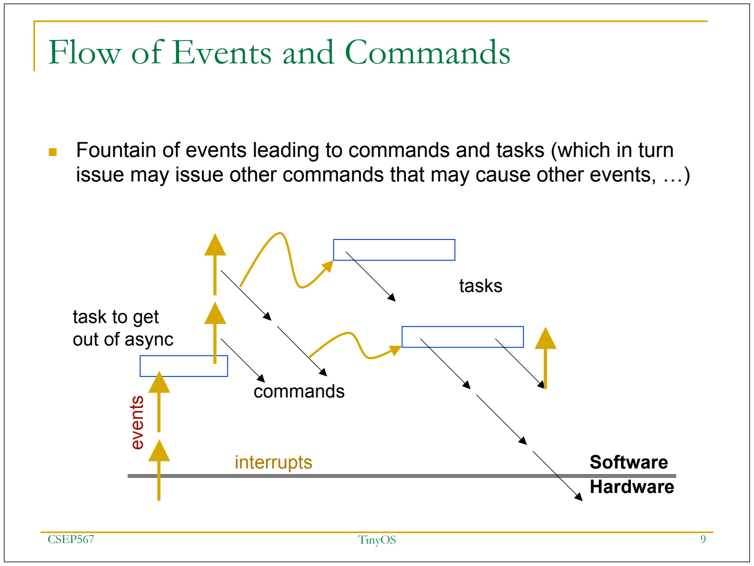
TinyOS Programming Model

- Separation of construction and composition
  - Programs are built out of components
- Specification of component behavior in terms of a set of interfaces
  - Components specify interfaces they use and provide
- Components are statically wired to each other via their interfaces
  - This increases runtime efficiency by enabling compiler optimizations
- Finite-state-machine-like specifications
- Thread of control passes into a component through its interfaces to another component



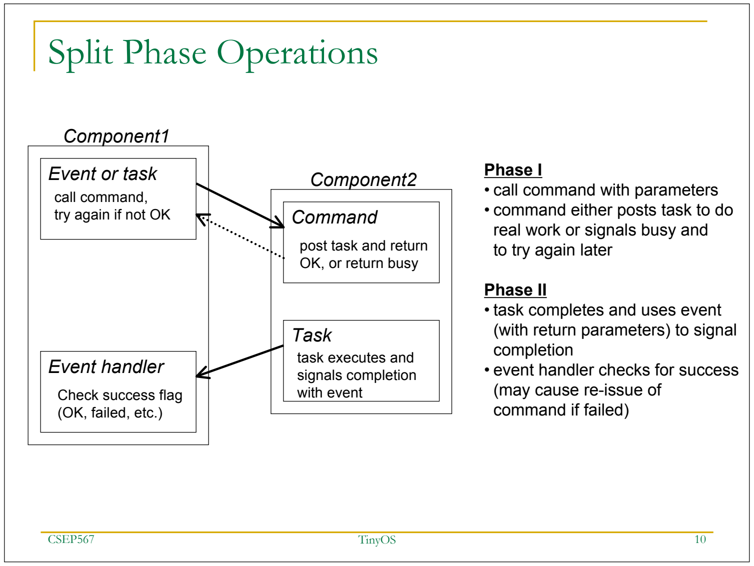
TinyOS Basic Constructs

- Commands
  - Cause action to be initiated
- Events
  - Notify action has occurred
  - Generated by external interrupts
  - Call back to provide results from previous command
- Tasks
  - Background computation
  - Not time critical



Flow of Events and Commands

- Fountain of events leading to commands and tasks (which in turn issue may issue other commands that may cause other events, …)



Split Phase Operations

Component1

Event or task
call command,
try again if not OK

Event handler
Check success flag
(OK, failed, etc.)

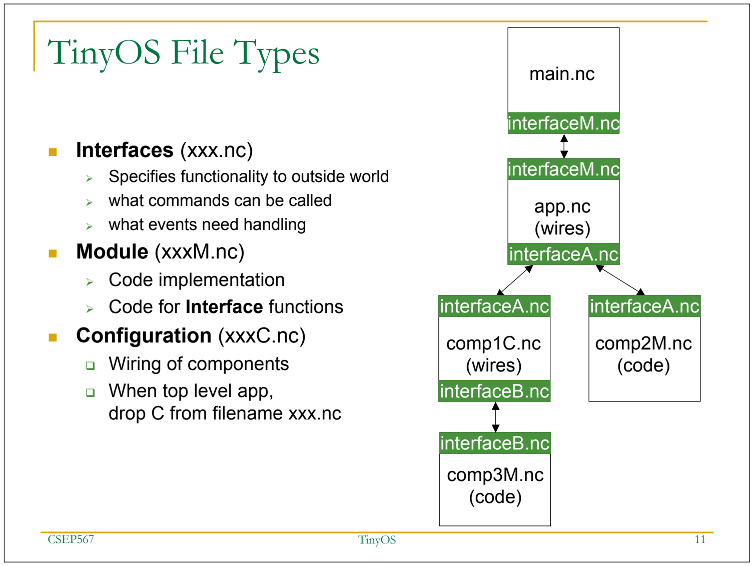
Component2

Command
post task and return
OK, or return busy

Task
task executes and
signals completion
with event

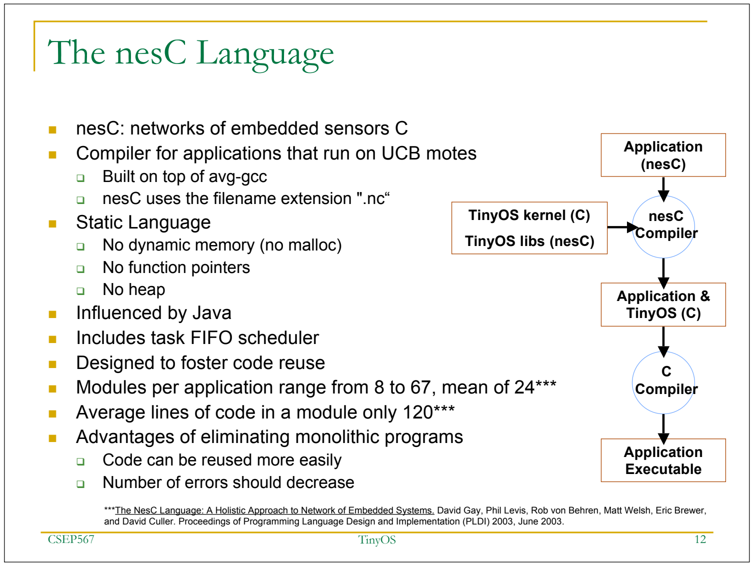
Phase I
- call command with parameters
- command either posts task to do real work or signals busy and to try again later

Phase II
- task completes and uses event (with return parameters) to signal completion
- event handler checks for success (may cause re-issue of command if failed)



TinyOS File Types

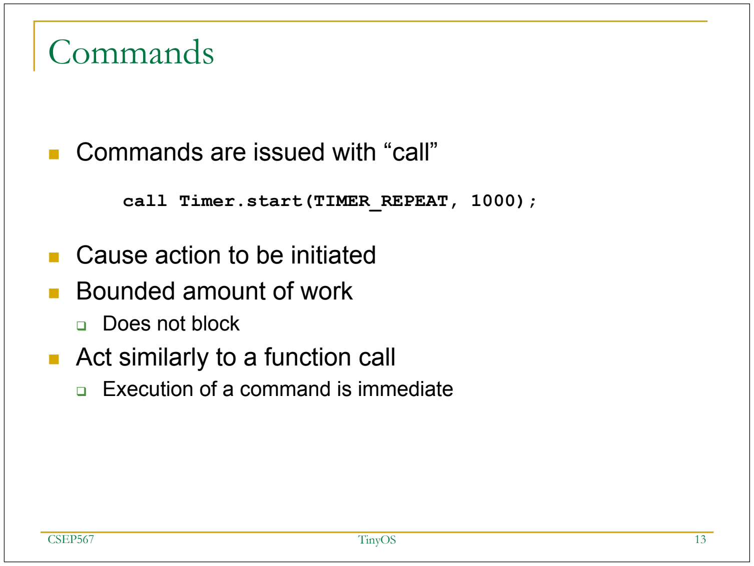
- **Interfaces** (xxx.nc)
  - Specifies functionality to outside world
  - what commands can be called
  - what events need handling
- **Module** (xxxM.nc)
  - Code implementation
  - Code for **Interface** functions
- **Configuration** (xxxC.nc)
  - Wiring of components
  - When top level app,
    drop C from filename xxx.nc



The nesC Language

- nesC: networks of embedded sensors C
- Compiler for applications that run on UCB motes
  - Built on top of avg-gcc
  - nesC uses the filename extension ".nc“
- Static Language
  - No dynamic memory (no malloc)
  - No function pointers
  - No heap
- Influenced by Java
- Includes task FIFO scheduler
- Designed to foster code reuse
- Modules per application range from 8 to 67, mean of 24***
- Average lines of code in a module only 120***
- Advantages of eliminating monolithic programs
  - Code can be reused more easily
  - Number of errors should decrease

***The NesC Language: A Holistic Approach to Network of Embedded Systems. David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

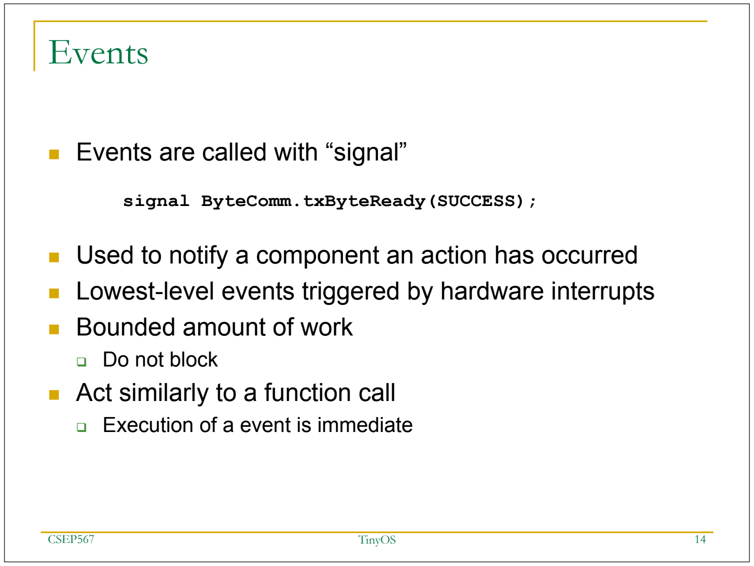


Commands

- Commands are issued with “call”

```
call Timer.start(TIMER_REPEAT, 1000);
```

- Cause action to be initiated
- Bounded amount of work
  - Does not block
- Act similarly to a function call
  - Execution of a command is immediate

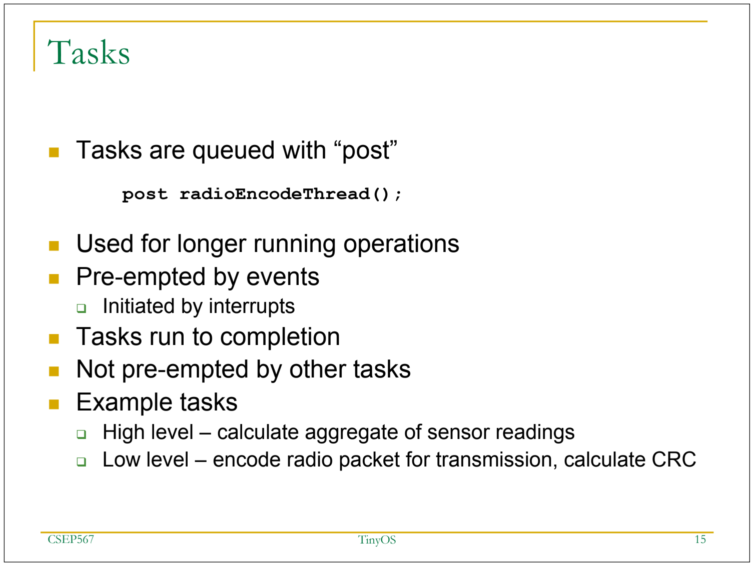


Events

- Events are called with “signal”

```
signal ByteComm.txByteReady(SUCCESS);
```

- Used to notify a component an action has occurred
- Lowest-level events triggered by hardware interrupts
- Bounded amount of work
  - Do not block
- Act similarly to a function call
  - Execution of a event is immediate

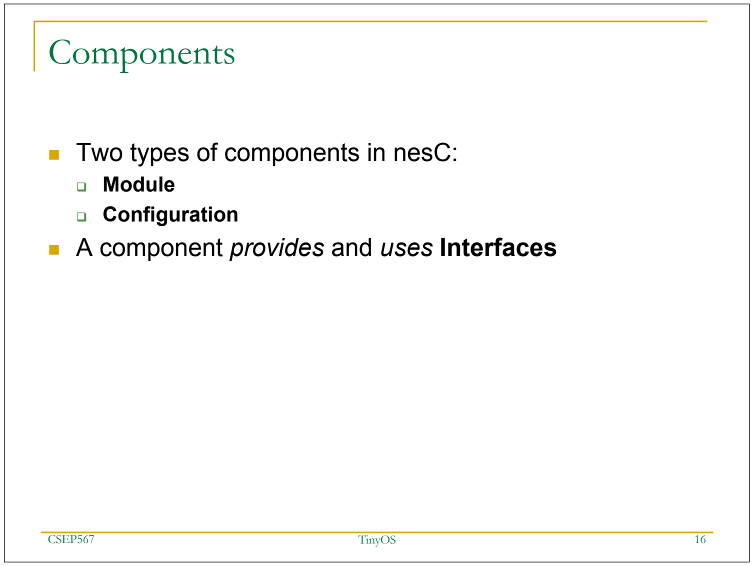


Tasks

- Tasks are queued with “post”

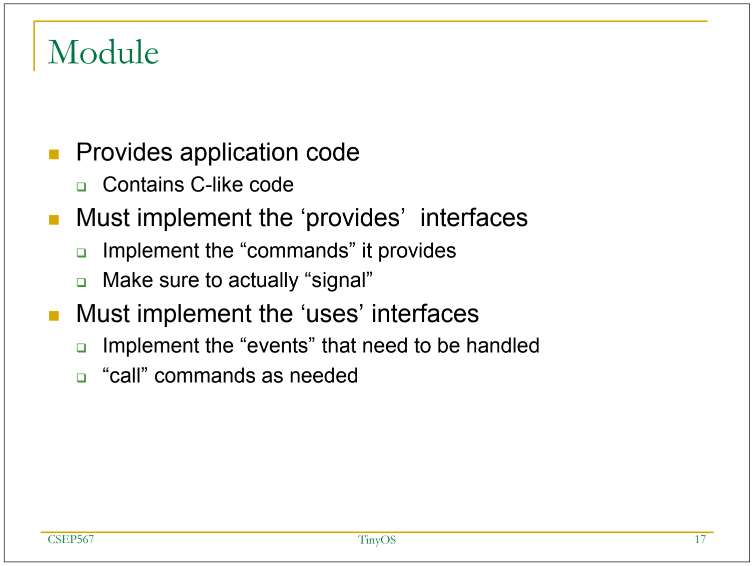
```
post radioEncodeThread();
```

- Used for longer running operations
- Pre-empted by events
  - Initiated by interrupts
- Tasks run to completion
- Not pre-empted by other tasks
- Example tasks
  - High level – calculate aggregate of sensor readings
  - Low level – encode radio packet for transmission, calculate CRC



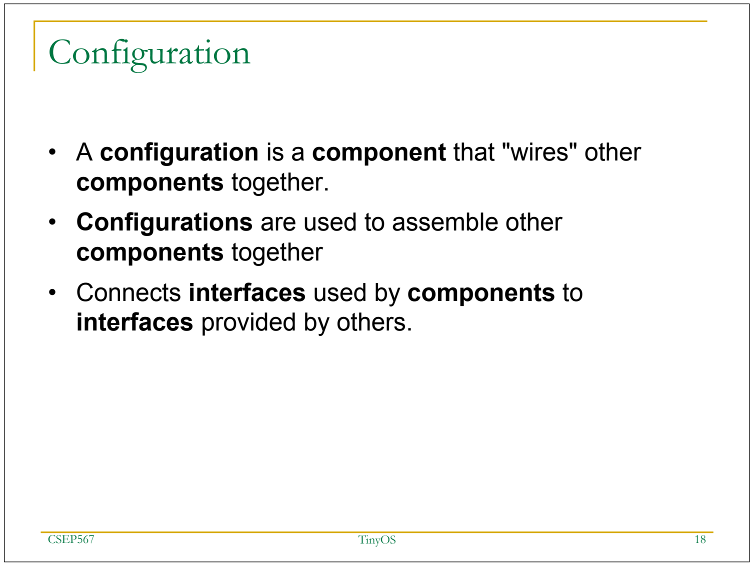
Components

- Two types of components in nesC:
  - **Module**
  - **Configuration**
- A component *provides* and *uses* **Interfaces**



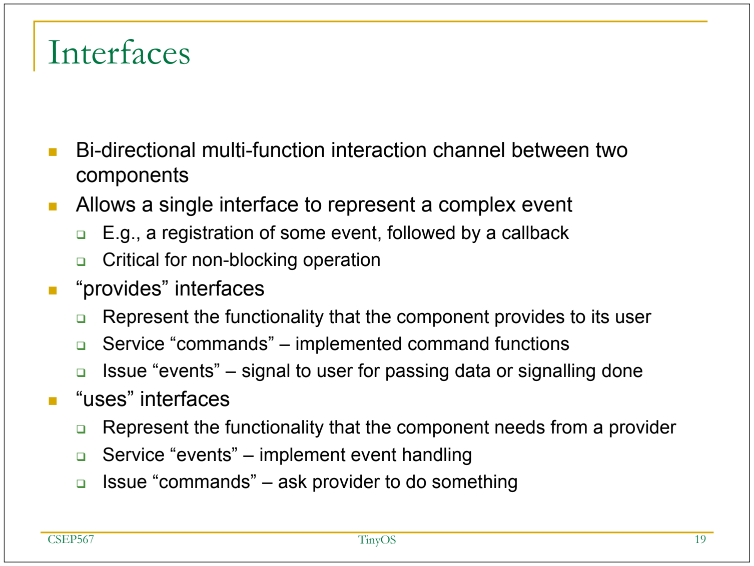
Module

- Provides application code
  - Contains C-like code
- Must implement the ‘provides’ interfaces
  - Implement the “commands” it provides
  - Make sure to actually “signal”
- Must implement the ‘uses’ interfaces
  - Implement the “events” that need to be handled
  - “call” commands as needed



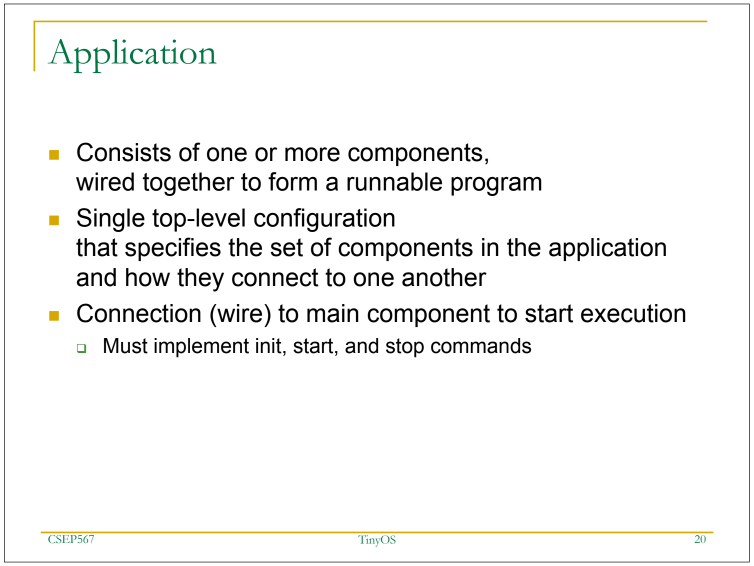
Configuration

- A **configuration** is a **component** that "wires" other **components** together.
- **Configurations** are used to assemble other **components** together
- Connects **interfaces** used by **components** to **interfaces** provided by others.



Interfaces

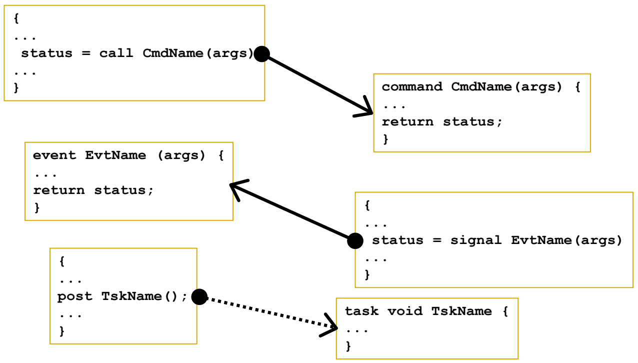
- Bi-directional multi-function interaction channel between two components
- Allows a single interface to represent a complex event
  - E.g., a registration of some event, followed by a callback
  - Critical for non-blocking operation
- “provides” interfaces
  - Represent the functionality that the component provides to its user
  - Service “commands” – implemented command functions
  - Issue “events” – signal to user for passing data or signalling done
- “uses” interfaces
  - Represent the functionality that the component needs from a provider
  - Service “events” – implement event handling
  - Issue “commands” – ask provider to do something



Application

- Consists of one or more components,
wired together to form a runnable program
- Single top-level configuration
that specifies the set of components in the application
and how they connect to one another
- Connection (wire) to main component to start execution
  - Must implement init, start, and stop commands

Commands, Events, and Tasks



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

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

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

```
{
...
 status = signal EvtName(args)
...
}
```

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

```
task void TskName {
...
}
```

Interfaces can fan-out and fan-in

- nesC allows interfaces to *fan-out* to and *fan-in* from multiple components
- One “provides” can be connected to many “uses” and vice versa
- Wiring fans-out, fan-in is done by a *combine* function that merges results

```
implementation {
  components Main, Counter, IntToLeds, TimerC;

  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> TimerC.StdControl;
```

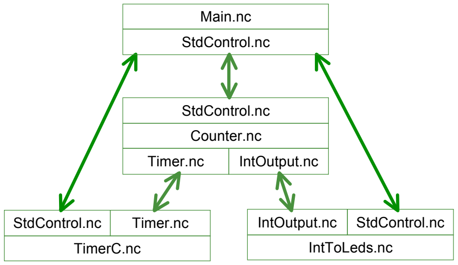
Fan-out by wiring

```
result_t ok1, ok2, ok3;
….
ok1 = call UARTControl.init();
ok2 = call RadioControl.init();
ok3 = call Leds.init();
….
return rcombine3(ok1, ok2, ok3);
```

Fan-in using rcombine
- rcombine is just a simple logical AND for most cases

Example

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

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



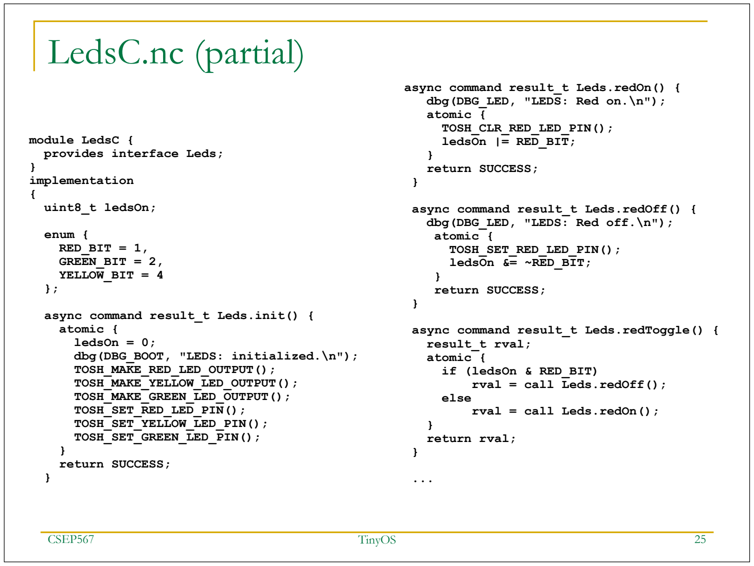
Controlling the Hardware in TinyOS

in tos/platform/mica2dot/hardware.h: you have
TOSH_ASSIGN_PIN(INT0, D, 0);

```
#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);}
```

Gives these control mechanisms:

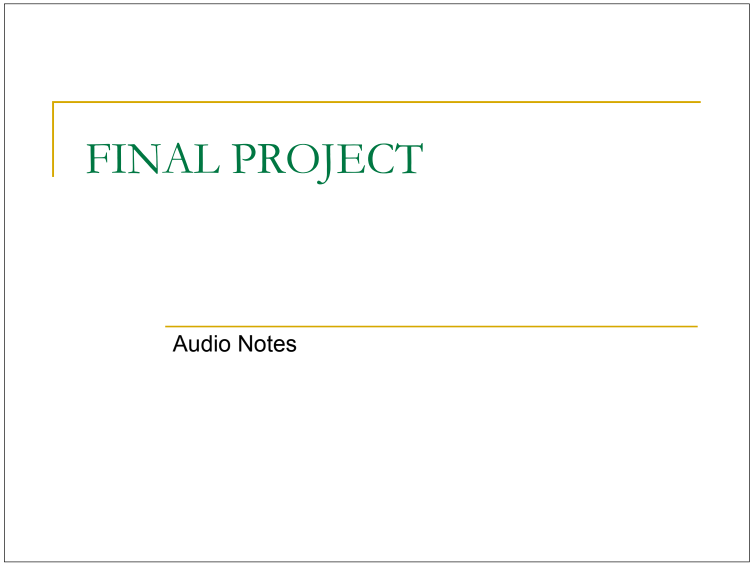
```
    TOSH_SET_INT0_PIN();
    TOSH_SET_INT0_PIN();
    TOSH_CLR_INT0_PIN();
    TOSH_READ_INT0_PIN();
    TOSH_MAKE_INT0_OUTPUT();
    TOSH_MAKE_INT0_INPUT();
```



LedsC.nc (partial)

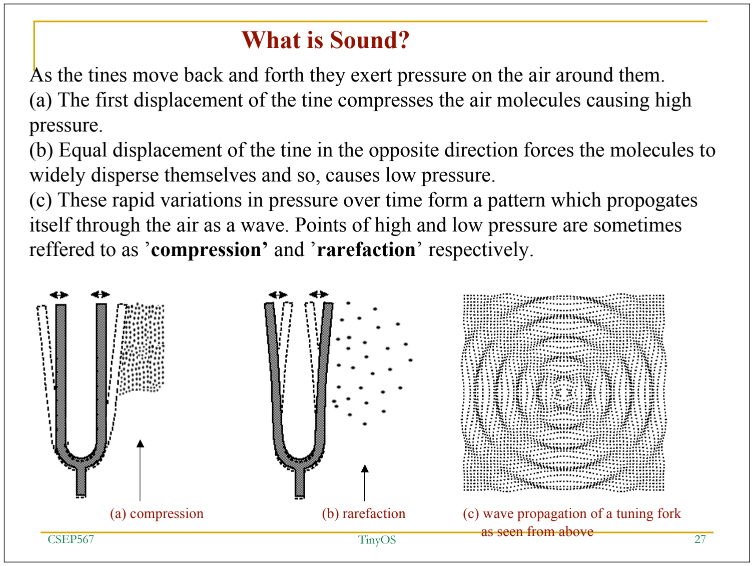
```
module LedsC {
  provides interface Leds;
}
implementation
{
  uint8_t ledsOn;

  enum {
    RED_BIT = 1,
    GREEN_BIT = 2,
    YELLOW_BIT = 4
  };

  async command result_t Leds.init() {
    atomic {
      ledsOn = 0;
      dbg(DBG_BOOT, "LEDS: initialized.\n");
      TOSH_MAKE_RED_LED_OUTPUT();
      TOSH_MAKE_YELLOW_LED_OUTPUT();
      TOSH_MAKE_GREEN_LED_OUTPUT();
      TOSH_SET_RED_LED_PIN();
      TOSH_SET_YELLOW_LED_PIN();
      TOSH_SET_GREEN_LED_PIN();
    }
    return SUCCESS;
  }
```

```
async command result_t Leds.redOn() {
   dbg(DBG_LED, "LEDS: Red on.\n");
   atomic {
     TOSH_CLR_RED_LED_PIN();
     ledsOn |= RED_BIT;
   }
   return SUCCESS;
 }

 async command result_t Leds.redOff() {
   dbg(DBG_LED, "LEDS: Red off.\n");
    atomic {
      TOSH_SET_RED_LED_PIN();
      ledsOn &= ~RED_BIT;
    }
    return SUCCESS;
 }

 async command result_t Leds.redToggle() {
   result_t rval;
   atomic {
     if (ledsOn & RED_BIT)
         rval = call Leds.redOff();
     else
         rval = call Leds.redOn();
   }
   return rval;
 }

 ...
```



FINAL PROJECT

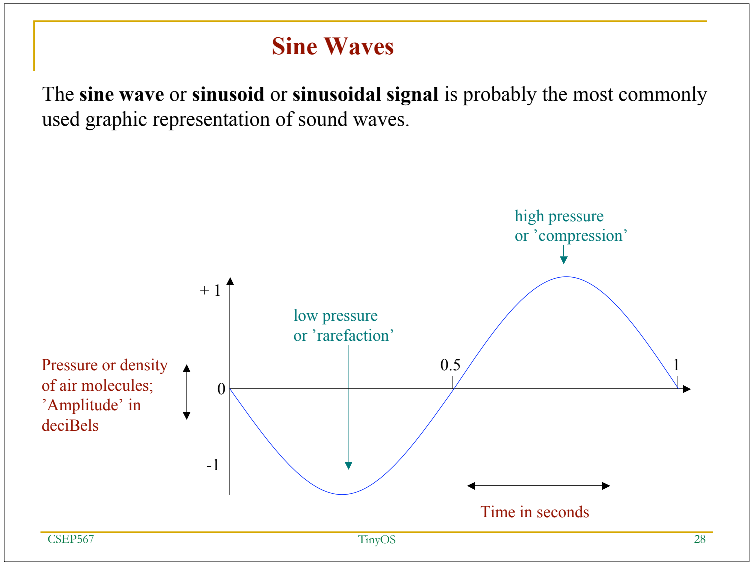
Audio Notes



What is Sound?

As the tines move back and forth they exert pressure on the air around them.
(a) The first displacement of the tine compresses the air molecules causing high pressure.
(b) Equal displacement of the tine in the opposite direction forces the molecules to widely disperse themselves and so, causes low pressure.
(c) These rapid variations in pressure over time form a pattern which propogates itself through the air as a wave. Points of high and low pressure are sometimes reffered to as ’**compression’** and ’**rarefaction**’ respectively.

(a) compression (b) rarefaction (c) wave propagation of a tuning fork as seen from above

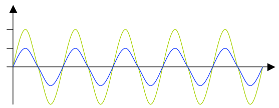


Sine Waves

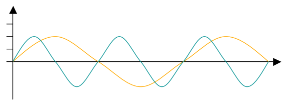
The **sine wave** or **sinusoid** or **sinusoidal signal** is probably the most commonly used graphic representation of sound waves.

Amplitude

•Amplitude describes the size of the pressure variations.
•Amplitude is measured along the vertical y-axis.
•Amplitude is closely related to *but not the same as!!!*, loudness.



(a) Two signals of equal frequency and varying amplitude



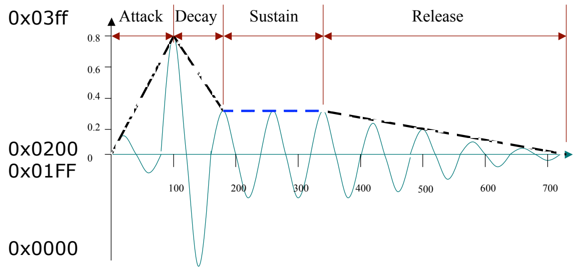
(b) Two signals of varying frequency and equal amplitude

Amplitude Envelope

The amplitude of a wave changes or 'decays' over time as it loses energy.

These changes are normally broken down into four stages;
Attack , **Decay**, **Sustain** and **Release**.

Collectively, the four stages are described as the **amplitude envelope**.


0x03ff
0x0200
0x01FF
0x0000
Attack
Decay
Sustain
Release