TinyOS
Introduction
TinyOS Outline

- Introduction to the Architecture of TinyOS and nesC
- Component Model
  - Components, interfaces, wiring
  - Commands and events; user and provider;
  - Modules and configurations
  - Wiring and callbacks
- Concurrency Model
  - Tasks and event handlers.
TinyOS Outline (cont)

- Components
  - Signatures and implementation blocks

- Interfaces
  - Commands and events
  - Generic and bidirectional

- Configurations
  - Wiring

- Generic components

- Split-Phase Operation
TinyOS: a general, embedded, lightweight operating system designed for low-power wireless sensors developed at UC Berkeley.

It provides a set of services and abstractions to make building sensor network applications easier.

It defines a concurrency execution model that emphasizes building applications from reusable services and components while avoiding unforeseen interactions.
Figure 1.3 Example application architecture. Application code uses a timer to act periodically, sensors to collect data, and a routing layer to deliver data to a sink.
The TinyOS system, libraries and applications are written in nesC.

nesC, a dialect of C, has features to reduce RAM use and help prevent race conditions.

nesC applications are built from components with well-defined bidirectional interfaces linked together to form an executable.
TinyOS

- Provides three features to make writing systems and applications easier:
  - a component model which defines how to write small, reusable pieces of code and compose them into larger abstractions.
  - a concurrent execution model which defines how components interleave their computations and how interrupt and non-interrupt code interact.
  - application program interfaces (APIs), services, component libraries and an overall component structure that simplifies writing new applications and services.
A Operating System for Tiny Devices?

- **Main Concept**
  - **HURRY UP AND SLEEP!!**
    - Sleep as often as possible to save power.
  - provide framework for concurrency and modularity.
    - Commands, events, tasks
  - interleaving flows, events - never poll, never block.

- **Separation of construction and composition.**

- **Programs are built out of components.**
  - Libraries and components are written in nesC.
  - Applications are too -- just additional components composed with the OS components

- **Each component is specified by an interface.**
  - Provides “hooks” for wiring components together.

- **Components are statically wired together based on their interfaces.**
  - Increases runtime efficiency.
Components and Interfaces

- A component provides and uses interfaces.
  - These interfaces are the only point of access to the component and are bi-directional.

- An interface declares a set of functions called commands and events.

- The interface’s user makes requests (calls commands) on the interface’s provider.

- The provider makes callbacks (signals events) to the interface’s user.
An interface defines a logically related set of commands and events.

Components implement the events they use and the commands they provide:

<table>
<thead>
<tr>
<th>Component</th>
<th>Commands</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>Can call</td>
<td>Must implement</td>
</tr>
<tr>
<td>Provide</td>
<td>Must implement</td>
<td>Can signal</td>
</tr>
</tbody>
</table>

Commands and events themselves are like regular functions (they contain arbitrary C code).

Calling a command or signaling an event is just a function call.
module PowerupC {
  uses interface Boot;
  uses interface Leds;
}

implementation {
  event void Boot.booted () {
    call Leds.led0On ();
  }
}
Components

- There are two types of components in nesC: **modules** and **configurations**.
- **modules** provide application code, implementing one or more interfaces.
- **module** code declares variables and functions, calls functions and compiles to assembly code. **module** implementation sections consist of nesC code that looks like C.
Configurations

- Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by other components. This is called **wiring**.
  - Every nesC application is described by a top-level configuration that wires together the components inside.

- nesC uses the filename extension "\.nc" for all source files - interfaces, modules and configurations.
configuration PowerupAppC { }

implementation {
    components MainC, LedsC, PowerupC;

    MainC.Boot <- PowerupC.Boot;
    PowerupC.Leds -> LedsC.Leds;
}
Wiring and Callbacks

- Wiring leaves the component connection decision to the programmer.
- It also provides an efficient mechanism for supporting callbacks.
- TinyOS provides a variable number of periodic or deadline timers.
- Associated with each timer is a callback to a function that is executed each time the timer fires.
module BlinkC {
  uses interface Boot;
  uses interface Timer;
  uses interface Leds;
}

implementation {
  event void Boot.booted ( ) {
    call Timer.startPeriodic (250); /* start the 250ms timer when it boots */
  }

  event void Timer.fired ( ) {
    call Leds.led0Toggle ( );
  }
}
interface Timer {
    command void startPeriodic (uint32_t interval);
    event void fired ( );
    ...
}

The connection between the startPeriodic command that starts the timer and the fired event which blinks the LED is implicitly specified by having the command and the event in the same interface.
configuration BlinkAppC { }
implementation {
    components MainC, LedsC,
    new TimerC () as MyTimer, BlinkC;
    BlinkC.Boot -> MainC.Boot;
    BlinkC.Leds -> LedsC.Leds;
    BlinkC.Timer -> MyTimer.Timer;
}

The Timer must be connected to a component that provides the actual timer. BlinkAppC wires BlinkC.Timer to a newly allocated MyTimer.
Concurrency Model

- TinyOS executes only one program consisting of selected system components and custom components needed for a single application.
- TinyOS has two threads of execution: tasks and hardware event handlers.
- Tasks are functions whose execution is deferred.
- Once scheduled, tasks run to completion and do not preempt each other.
Event Handlers

- Hardware event handlers execute in response to hardware interrupts.
- They run to completion, but event handlers may preempt execution of a task or other event handlers.
- Commands and events executed as part of a hardware event handler must be declared with the **async** keyword.
Components and Interfaces

- All components have two code blocks.
- The first block describes its *signature* and the second block describes its *implementation*.
- A component signature declares whether it provides or uses an interface.
(module PowerupC { // signature 
   // implementation
})

classification LedsC { // signature 
   // implementation
}

All components have two code blocks.
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}

- MainC is a configuration that implements the boot sequence of a node and provides the Boot interface so that components, such as PowerupC, can be notified when a node has fully booted.
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}

- The `as` keyword lets a signature provide an alternate name for an interface for clarity or to distinguish multiple instances of the same interface (see LedsP Module).
Interfaces define a functional relationship between two or more components.

Like components, interfaces have a one-to-one mapping between names and files and exist in global namespace.

- e.g., the file `Boot.nc` contains the interface `Boot`. 
An interface declaration has one or more functions in it.

The two kinds of functions are: \textit{commands and events}.

interface Init {
    command error_t init ( );
}

interface Boot {
    event voidBooted ( );
}

Remember – While users call commands and providers can signal events, users implement events while providers implement commands.
Interfaces

User

Commands

Interface

Provider

Events
module PowerupC {
    uses interface Boot;
    uses interface Leds;
}

implementation {
    event void Boot.booted ( ) {
        call Leds.led0On ( );
    }
}
Event Implementation

• As a user of the **Boot** interface which has a single event **booted**, **PowerupC** must provide an implementation.

{an event implementation is essentially an event handler.}

• As the provider of the single event **Boot.booted**, **MainC** signals the event when a node has booted successfully.
As the user of the interface Leds, PowerupC calls the command Leds.led0On which is implemented by the provider Leds.

Boot, Init and Leds are type-free interfaces.
- Take one or more types as parameters.

```cpp
Interface Queue]<t> {  
    command bool empty ( );
    command uint8_t size ( );
    command uint8_t maxsize ( );
    command t head ( );
    command t dequeue ( );
    command error_t enqueue (t newVal);
    command t element (uint8_t idx):
}
```
When a component declares a generic interface, it must specify its parameters.

Example:

```
Module QueueUserC {
    uses interface Queue<uint32_t>;
}
```

Note – when connecting users to providers, interface types must match.
· Declare both commands from a user to a provider as well as events from a provider to a user.

```c
interface Notify <val_t> {
    command error_t enable ( );
    command error_t disable ( );
    event void notify (val_t val);
}
```
The Notify interface has two commands. If notifications are enabled, the provider of the interface signals notify events.

Bidirectional interfaces enable components to register callbacks without needing function pointers.
Module PowerupToggleC {
    uses interface Boot;
    uses interface Leds;
}

implementation {
    event void Boot.booted () {
        while (1) {
            call Leds.led0Toggle ( );
            call Leds.led1Toggle ( );
            call Leds.led2Toggle ( );
        }
    }
} /* modules allocate state and implement executable logic */
configuration PowerupToggleAppC { }
implementation {
    components MainC, Leds, PowerupToggleC;

    PowerupToggleC.Boot -> MainC.Boot;
    PowerupToggleC.Leds -> LedsC.Leds;
}

The configuration wires components by mapping names in one component’s signatures to a set of names in another component’s signature.
interface Get <val_t> {
    command val_t get ( );
}

All module variables are private.

interface, part of UserButtonC, demonstrates that interfaces are the only way to access to a variable.
module CountingGetC {
    provides interface Get<uint8_t>;
}

implementation {
    uint8_t count;

    command uint8_t Get.get() {
        return count++;
    }
}
module CountingGetC {
    provides interface Get <uint8_t>;
}

implementation {
    uint8_t count;
    command uint8_t Get.get (){
        return count++
    }
}

Generic Components

- Have multiple instances unlike hardware singletons.
- Generic components have the keyword `generic` before their signature:

```c
generic configuration TimerMilliC ( ) {
    provides interface Timer <TMilli>;
}
```
- Configurations must instantiate using `new`:

```c
... components new TimerMilliC ( ) as Timer0;
```
Split-phase Interfaces

- Hardware is almost always split-phase rather than blocking.
- In split-phase operations, the request that initiates an operation completes immediately.
- Actual completion of the operation is signaled by a separate callback.
- To save RAM space, TinyOS does not use multiple threads, but rather uses bidirectional split-phase software interfaces:
  - A command starts the operation.
  - An event signals the operation is complete.
The **Read** interface is the basic TinyOS interface for **split-phase** data acquisition. Most sensor drivers provide Read which is generic:

```cpp
interface Read <val_t> {
    command error_t read ( );
    event void readDone (error_t, val_t val );
}
```
Send Interface

- The basic TinyOS packet transmission interface, **Send**, is also a split-phase operation.
- It is more complex because it requires passing a pointer for a packet to transmit.
interface Send {

  command error_t send (message_t* msg, uint8_t len);
  event void sendDone (message_t* msg, error_t error);

  command error_t cancel (message_t* msg);
  command void* getPayload (message_t* msg);
  command uint8_t maxPayloadLength (message_t* msg);

}
Split-Phase Send Interface

- A provider of **Send** defines the **send** and **cancel** functions and can **signal** the **sendDone** event.

- A user of **Send** needs to define the **sendDone** event and can **call** the **send** and **cancel** commands.

- When a **send** call returns **SUCCESS**, the **msg** parameter has been passed to the provider which tries to send the packet.

- When the **send** completes, the provider signals **sendDone**, passing the pointer back to the user.
Introduction to TinyOS Summary

- **Introduction to TinyOS and nesC**
- **Component Model**
  - Components, interfaces, wiring
  - Commands and events; user and provider;
  - Modules and configurations
  - Wiring and callbacks
- **Concurrency Model**
  - Tasks and event handlers.
Introduction to TinyOS Summary

- Components
  - Signatures and implementation blocks

- Interfaces
  - Commands and events
  - Generic and bidirectional (as)

- Configurations
  - Wiring

- Generic components (new)

- Split-Phase Operation
  - Read and Send