TinyOS and NesC

EEC 693: Advanced Embedded Systems

An 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
What is TinyOS

- TinyOS is an open-source operation system designed for wireless embedded sensor networks
- Not an operating system for general-purpose
- Actually a set of software components that can be “wired” together into a single binary which is run on the motes

TinyOS Features

- The TinyOS system, library and applications written in NesC
- Event-driven architecture
  - Lower layer sends events to higher layer
  - Low overhead – No busy-wait cycles
- Interrupt-driven
- No kernel, process management, memory management
  - OS is really a misnomer
TinyOS Features (contd.)

- Component driven programming model
  - Size ~ 400 bytes
  - Extremely flexible component graph
- Main Ideology
  - HURRY UP AND SLEEP!!
  - Sleep as often as possible to save power
  - High concurrency between tasks and events
  - Two Level Simple FIFO scheduler, Queue of size 7

TinyOS Features (contd.)

- Single *shared* Stack
- STATIC memory allocation!
  - No heap
  - No function pointers
- Global variables
  - Available on a per-frame basis
- Local variables
  - Saved on the stack
  - Declared within a method
The NesC Programming Language

Programming Model

- Separation of construction and composition
- Programs are built out of 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
Component Model

- Application = Scheduler + Components
- Each component has
  - Commands
  - Event handlers
  - Frame (variables, internal state)
  - Tasks (concurrency, data processing)
  - Constrained storage
  - Frame per component, shared stack, no heap

Component Hierarchy

TinyOS Components

- Components use and provide interfaces, commands, and events
  - Specified by a component’s interface
  - The word “interface” has two meanings in TinyOS -- bidirectional
- Components implement the events they use and the commands they provide:
  - Must implement the commands and can signal the events which it provides
  - Can call commands and must implement handlers for events uses

Two Types of Components

- **Modules** implement application behavior
- **Configurations** wire components together
- A component does not care if another is a module or a configuration
- A component may be composed of other components
- Components are **wired** together by connecting **users** with **providers**
- Forms a hierarchy
TinyOS Interfaces

- Bi-directional
- Commands flow down
- Events flow up
- Each component provides some interfaces, and uses others

Wiring Example

```java
class TinyOS { // your class definition here }
```
**More on Wiring**

- Not only the interfaces can be wired together; commands/events also can be
- Always from user to provider
- Any wired elements must be compatible
- Can only wire interface Send to Send
- Binds a User to a Provider’s implementation

```plaintext
User.interface -> Provider.interface
BlinkM.Timer -> SingleTimer.Timer;  // Explicit
BlinkM.Leds -> LedsC;              // Implicit
```

---

**Component Specification Keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>Collection of event and command defns</td>
</tr>
<tr>
<td>module</td>
<td>Basic component implemented in nesC</td>
</tr>
<tr>
<td>configuration</td>
<td>Component composed of others wired</td>
</tr>
<tr>
<td>implementation</td>
<td>Contains code and variables for module/ configuration</td>
</tr>
<tr>
<td>components</td>
<td>List of components wired in a configuration</td>
</tr>
<tr>
<td>provides</td>
<td>Lists interfaces provided by component</td>
</tr>
<tr>
<td>uses</td>
<td>Lists interfaces used by component</td>
</tr>
<tr>
<td>as</td>
<td>Alias an interface to another name</td>
</tr>
<tr>
<td>command</td>
<td>Direct function call exposed by interface</td>
</tr>
<tr>
<td>event</td>
<td>Callback message exposed by interface</td>
</tr>
</tbody>
</table>
Commands

- Invoked using keyword **call**
- Deposit request parameters into the frame
- Non-blocking
- Need to return status/ control returns to caller
- Can call lower/same-level commands
- Generally flow downwards
- Can post a task (postpone time consuming work by posting a task)

Events

- Marked by keyword **event**, invoked using **signal**
- Time critical
- Can call commands, signal events, post tasks
- Can NOT be signaled by commands
- Interrupt trigger the lowest level events
- Shorter duration (hand off to task if need be)
- Preempt tasks, not vice versa
- Generally flow upward, control returns to signaler
- Last-in first-out semantics (no priority among events)
- **Do not confuse an hardware interrupt (event handler) from the NesC event keyword!!**
Tasks

- Marked using keyword **task**, invoked using **post**
- Provide concurrency internal to a component
- Able to perform operations beyond event context
- May call commands
- 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
  - No complex task priorities and IPC
- May signal events
- Not preempted by other tasks

Tasks contd...

- FIFO scheduling
- Perform computation-intensive work
- Can be posted from within a command, event or even another task
- Handling of multiple data flows
- Sequence of non-blocking command/event through the component graph
- Are preempted by events
Concurrency Model

- The execution model consists of
  - Run-to-completion tasks that typically represent the ongoing computation
  - Interrupt handlers that are signaled asynchronously by hardware
- **Synchronous Code (SC):** code (functions, commands, events, tasks) only reachable from tasks
- **Asynchronous Code (AC):** code that is reachable from at least one interrupt handler
- **Race-Free Invariant:** Any update to shared state is either SC-only or occurs in an atomic statement
  - Enforced at compile time

Concurrency Model contd...

- To handle concurrency, NesC provides 2 tools
  - **atomic** sections
  - task(s)
- Atomicity is implemented by simply disabling/enabling interrupts
  - Disabling interrupts for a long time can delay interrupt-handling and make system less responsive
- Atomic statements executed “as-if” no other computation occurred; Should be short!
- If potential race condition is present and programmer knows it’s not an actual race condition, can specify **norace**