TinyOS

An Embedded Operating System

A Different Kind of Operating System

- programming framework + reusable code *components* = application-specific OS
- written in nesC
  - no function pointers
    - compiler can optimize an entire call path
  - no dynamic memory allocation
    - instead, memory is statically allocated at compile-time (*static virtualization*)
    - prevents memory fragmentation, runtime allocation failures
- open source
- designed for networks of sensor nodes:
  - limited resources (memory, cpu, etc.)
  - reactive concurrency (nodes must be able to respond in real time)
  - low-power operation
- uses today include sensor networks, smart meters, and building automation

Component-Based Programming Model

program is a graph of *components* compiled by the nesC compiler

- hardware resources are abstracted as components

components:

- *interfaces* specify <u>split-phase</u> requests for services using *commands* and *events*, which can post *tasks*
    - *command*: ask component to do something (a service; e.g. sending a message)
    - *event*: signal service is complete (e.g. hardware interrupt, message arrived)
    - *task*: can be posted to defer computation to later (managed by scheduler)

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

interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

interface Clock {
  command result_t setRate(char interval, char scale);
  event result_t fire();
}

interface SendMsg {
  command result_t send(uint16_t address,
                        uint8_t length,
                        TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg,
                          result_t success);
}
```

Fig. 3. Sample TinyOS interface types

Component-Based Programming Model

2 types of components:

1. *modules*: code
2. *configurations*: connect other components' interfaces

application = top-level configuration

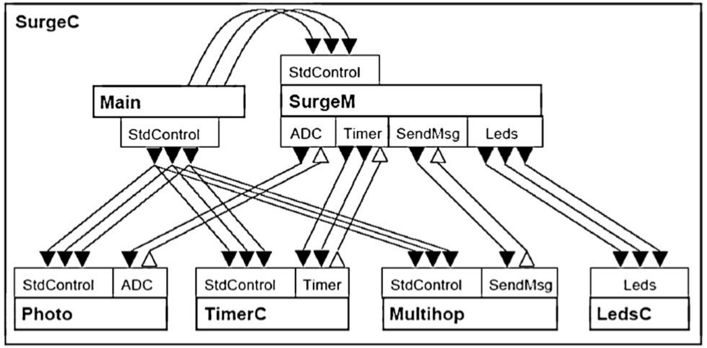
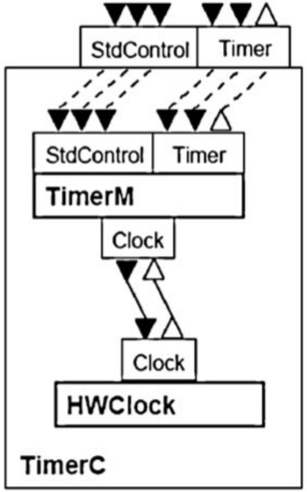


Fig. 5. The top-level configuration for the Surge application




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

Fig. 4. TinyOS's Timer Service: the `TimerC` configuration

Benefits of Component System

- separate interface and implementation
- data privacy
- code reuse
- linguistic structure for nesC optimizations
- easy to work with limited resources
    - each program uses a finite number of components
    - NesC compiler further minimizes TinyOS program size
        - inlines some components to remove procedure call boundary-crossings
        - reduces dead code
        - removes redundancy

drawback: hard to understand when first looking at a system

Scheduler

- *non-preemptive* (run-to-completion): tasks run either until they are complete or until they explicitly yield control to the scheduler
    - tasks are atomic wrt other tasks
- TinyOS programming framework does not require the scheduler to execute tasks in a particular order
    - standard scheduler is FIFO
    - other policies have been implemented, including earliest-deadline first

Concurrency

- Concurrency is event-driven
    - split-phase interfaces, asynchronous events, and tasks that defer computation
- *asynchronous code* = code reachable from at least 1 interrupt handler.
  to handle asynchronous code:
    1. convert all conflicting code to tasks (synchronous), or
    2. use atomic sections
        - NesC `atomic` keyword guarantees atomicity by turning off interrupts and preventing looping
- safety enforced at compile time -- no locks needed!
- time-critical sections of code can be handled in an *event handler*, even when they update shared state

Low Power

- TinyOS is application-specific, so it doesn't contain any unnecessary functions
- split-phase operations and event-driven model avoid spinlocks and heavyweight concurrency, which reduces CPU usage
- `StdControl` interface gives components a low-power *idle* state
  - save state in RAM/nonvolatile memory
  - inform CPU about decreased resource use (so system can decide whether to use deep power save modes)
- hardware/software transparency: can replace software with more efficient hardware implementations without changing application structure

Resources

TinyOS: An Operating System for Sensor Networks, especially pp. 115-133 (read it here)

Experiences from a Decade of TinyOS Development

You can view the TinyOS GitHub repository here.