# Wireless Embedded Systems and Networking

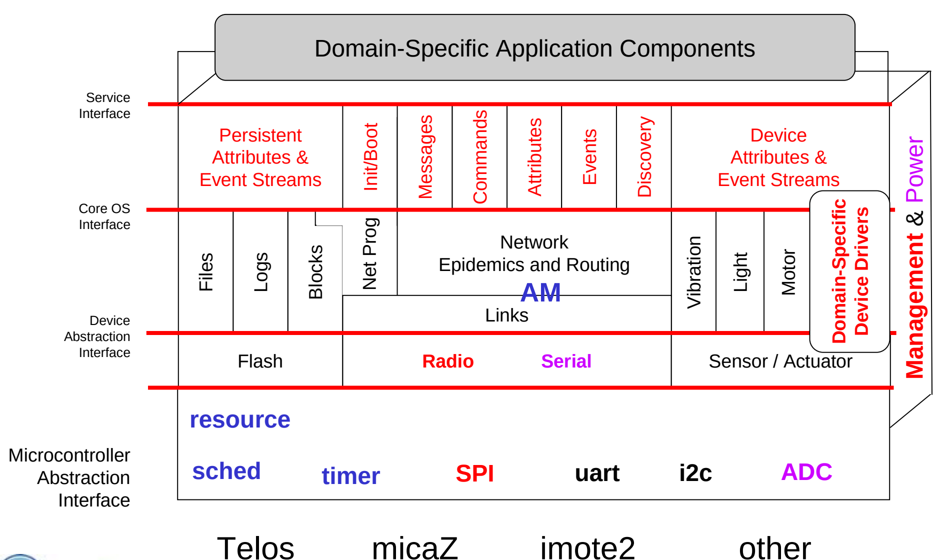
### Foundations of IP-based Ubiquitous Sensor Networks

## TinyOS 2.0 Design and Application Services

**David E. Culler**
***University of California, Berkeley***
***Arch Rock Corp.***
***July 10, 2007***



# Complete Network Embedded System



## Outline

- Key TinyOS Concepts
- TinyOS Abstraction Architecture
- A Simple Event-Driven Example
- Execution Model
- Critical system elements
  - Timers, Sensors, Communication
- Service Architecture



## TinyOS 2.0

- Primary Reference: http://www.tinyos.net/tinyos-2.x/doc/
- http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/

# Key TinyOS Concepts

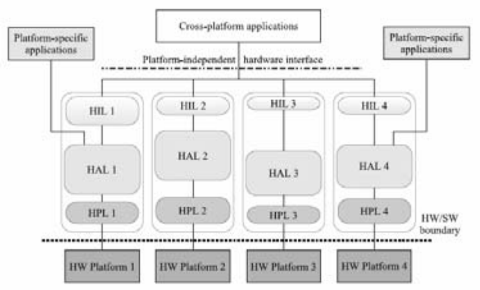
- Application / System = Graph of Components + Scheduler
- *Module*: component that implements functionality directly
- *Configuration*: component that composes components into a larger component by connecting their interfaces
- *Interface*: Logically related collection of commands and events with a strongly typed (polymorphic) signature
  - May be parameterized by type argument
  - *Provided* to components or *Used* by components
- *Command*: Operation performed (called) across components to initiate action.
- *Event*: Operation performed (signaled) across components for notification.
- Task: Independent thread of control instantiated within a component. Non-preemptive relative to other task.
- Synchronous and Asynchronous contexts of execution.

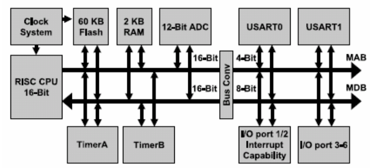


# TinyOS Abstraction Architecture

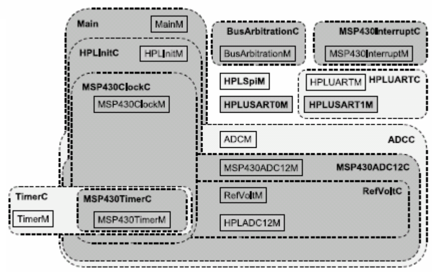
- HPL – Hardware Presentation Layer
  - Components that encapsulate physical hardware units
  - Provide convenient software interface to the hardware.
  - The hardware is the state and computational processes.
  - Commands and events map to toggling pins and wires
- HAL –Hardware Abstraction Layer
  - Components that provide useful services upon the basic HW
  - Permitted to expose any capabilities of the hardware
    - » Some platforms have more ADC channels, Timers, DMA channels, capture registers, …
  - Logically consistent, but unconstrained
- HIL – Hardware Independent Layer
  - Components that provide well-defined services in a manner that is the same across hardware platforms.
  - Implement common interfaces over available HAL

# Illustration




(a) Functional block diagram of the TI MSP430F149 $\mu$C





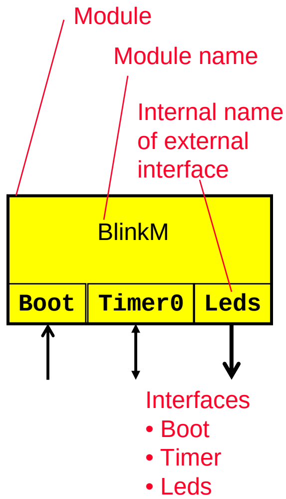

# TinyOS – a tool for defining abstractions

- All of these layers are constructed with the same TinyOS primitives.
- We'll illustrate them from a simple application down.

- Note, components are not objects, but they have strong similarities.
  - Some components encapsulate physical hardware.
  - All components are allocated statically (compile time)
    - » Whole system analysis and optimization
  - Logically, all components have internal state, internal concurrency, and external interfaces (Commands and Events)
  - Command & Event handlers are essentially public methods
  - Locally scoped
    - » Method invocation and method hander need not have same name (like libraries and objects)
    - » Resolved statically by wiring
      - Permits interpositioning

## A simple event-driven module – BlinkM.nc

```
#include "Timer.h"
module BlinkM
{
  uses interface Boot;
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
```

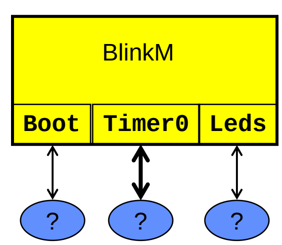



- **Coding conventions: TEP3**



## A simple event-drvien module (cont)

```
#include "Timer.h"
module BlinkM
{
  uses interface Boot;
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
```




Two Event Handlers

Each services external event by calling command on some subsystem

## Simple example: Boot interface

```
interface Boot {
  /**
   * Signaled when the system has booted successfully. Components can
   * assume the system has been initialized properly. Services may
   * need to be started to work, however.
   *
   * @see StdControl
   * @see SplitConrol
   * @see TEP 107: Boot Sequence
   */
  event void booted();
}
```

- $tinyOS-2.x/tos/interfaces/
- Defined in TEP 107 – Boot Sequence
- Consists of a single event.
- Hardware and operating system actions prior to this simple event may vary widely from platform to platform.
- Allows module to initialize itself, which may require actions in various other parts of the system.



## Simple example: LEDs interface

```
#include "Leds.h"

interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();  ...
 /*
  * @param  val   a bitmask describing the on/off settings of the LEDs
  */
  async command uint8_t get();
  async command void set(uint8_t val);
}
```

- $tinyOS-2.x/tos/interfaces/
- set of Commands
  - Cause action
  - get/set a physical attribute (3 bits)
- async => OK to use even within interrupt handlers
- Physical wiring of LEDs to microcontroller IO pins may vary

## Timer

```
interface Timer<precision_tag>
{
  command void startPeriodic(uint32_t dt);
  event void fired();

  command void startOneShot(uint32_t dt);
  command void stop();
  command bool isRunning();
  command bool isOneShot();
  command void startPeriodicAt(uint32_t t0, uint32_t dt);
  command void startOneShotAt(uint32_t t0, uint32_t dt);
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

- $tinyOS-2.x/tos/lib/timer/Timer.nc
- Rich application timer service built upon lower level capabilities that may be very different on different platform
  - Microcontrollers have very idiosyncratic timers
- Parameterized by precision



## TinyOS Directory Structure

- *tos/system/* - Core TinyOS components. This directory's
  - components are the ones necessary for TinyOS to actually run.
- *tos/interfaces/* - Core TinyOS interfaces, including
  - hardware-independent abstractions. Expected to be heavily used not just by *tos/system* but throughout all other code. *tos/interfaces* should only contain interfaces named in TEPs.
- *tos/platforms/* - code specific to mote platforms, but chip-independent.
- *tos/chips/***/* - code specific to particular chips and to chips on particular platforms.
- *tos/lib/***/* - interfaces and components which extend the usefulness of TinyOS but which are not viewed as essential to its operation.
- *apps/*, *apps/demos*, *apps/tests*, *apps/tutorials*.

## Timers

```
#include "Timer.h"

…
typedef struct { } TMilli; // 1024 ticks per second
typedef struct { } T32khz; // 32768 ticks per second
typedef struct { } TMicro; // 1048576 ticks per second
```

- Timers are a fundamental element of Embedded Systems
  - Microcontrollers offer a wide range of different hardware features
  - Idiosyncratic
- Logically Timers have
  - Precision - unit of time the present
  - Width - # bits in the value
  - Accuracy - how close to the precision they obtain
- TEP102 defines complete TinyOS timer architecture
- Direct access to low-level hardware
- Clean virtualized access to application level timers



## Example – multiple virtual timers

```
#include "Timer.h"

module Blink3M
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

```
event void Timer0.fired()
  {
     call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
     call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
     call Leds.led2Toggle();
  }
}
```

## Composition

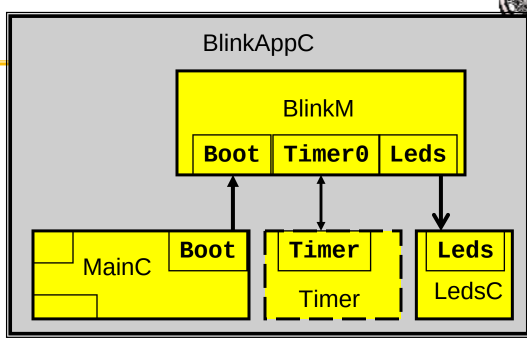
- Our event-driven component, Blink, may be built directly on the hardware
  - For a particular microcontroller on a particular platform
- or on a simple layer for a variety of platforms
- or on a full-function kernel

- Or it may run in a simulator on a PC,
- Or…

- As long as it is wired to components that provide the interfaces that this component uses.
- And it can be used in a large system or application



## Configuration

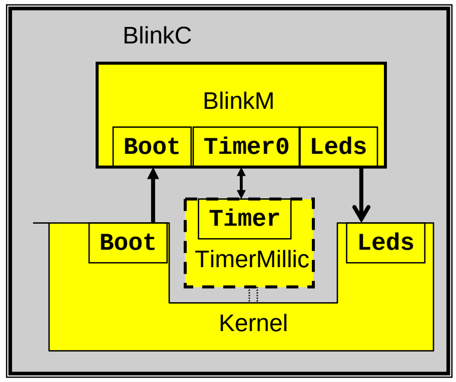



```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkM, LedsC;
  components new TimerMilliC() as Timer;

  BlinkM       -> MainC.Boot;
  BlinkM.Leds  -> LedsC;
  BlinkM.Timer0 -> Timer.Timer;
}
```

- Generic components create service instances of an underlying service. Here, a virtual timer.
- If the interface name is same in the two components, only one need be specified.

## A Different Configuration

```
configuration blinkC{
}

implementation{
  components blinkM;
  components MainC;
  components Kernel;

  blinkM.Boot -> Kernel.Boot;
  blinkM.Leds -> Kernel.Leds;
  components new TimerMilliC();
  blinkM.Timer0 -> TimerMilliC.Timer;
}
```




- **Same module configured to utilize a very different system substrate.**



## Execution Behavior

- **Timer interrupt is mapped to a TinyOS event.**
- **Performs simple operations.**
- **When activity stops, entire system sleeps**
  - **In the lowest possible sleep state**
- **Never wait, never spin. Automated, whole-system power management.**

## Module state

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  users interface Boot;
}
implementation
{
  uint8_t counter = 0;

  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
  }
}
```

- Private scope
- Sharing through explicit interface only!
  - Concurrency, concurrency, concurrency!
  - Robustness, robustness, robustness
- Static extent
- HW independent type
  - unlike int, long, char



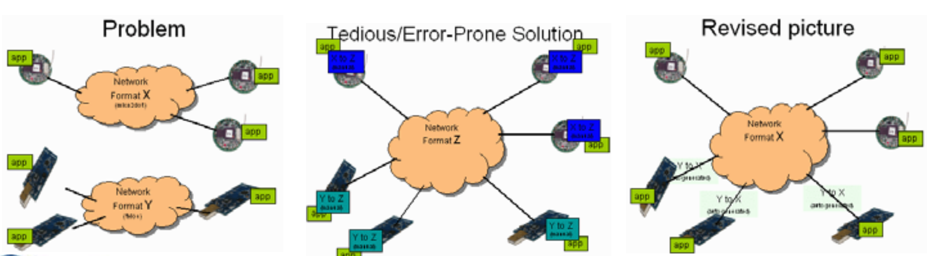
## TinyOS / NesC Platform Independent Types

- Common numeric types

| | 8 bits | 16 bits | 32 bits | 64 bits |
|---|---|---|---|---|
| signed | int8_t | int16_t | int32_t | int64_t |
| unsigned | uint8_t | uint16_t | uint32_t | uint64_t |

- Bool, …
- Network Types
  - Compiler does the grunt work to map to canonical form

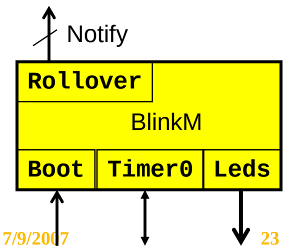
http://nescc.sourceforge.net



# Events

```
module BlinkM {
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  uses interface Boot;
  provides interface Notify<bool> as Rollover;
}
implementation
{
  uint8_t counter = 0;

  event void Boot.booted()
  { call Timer0.startPeriodic( 250 ); }

  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
    if (!counter) signal Rollover.notify(TRUE);
  }
}
```

- **Call commands**
- **Signal events**
- **Provider of interface handles calls and signals events**
- **User of interface calls commands and handles signals**



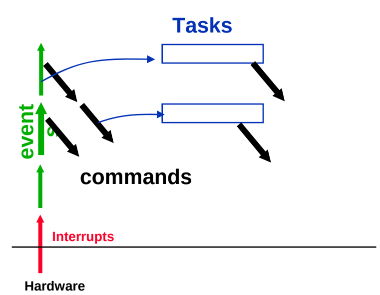



# Tasks

```
/* BAD TIMER EVENT HANDLER */
event void Timer0.fired() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {
    call Leds.led0Toggle();
  }
}
```

```
/* Better way to do a silly thing */
task void computeTask() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {}
}

event void Timer0.fired() {
  call Leds.led0Toggle();
  post computeTask();
}
```

- **Need to juggle many potentially bursty events.**
- **If you cannot get the job done quickly, record the parameters locally and post a task to do it later.**
- **Tasks are preempted by lower level (async) events.**
  - Allow other parts of the system to get the processor.
  - Without complex critical semaphores, critical sections, priority inversion, schedulers, etc.



# Split-Phase Operations

- For potentially long latency operations
  - Don't want to spin-wait, polling for completion
  - Don't want blocking call - hangs till completion
  - Don't want to sprinkle the code with explicit sleeps and yields
- Instead,
  - Want to service other concurrent activities will waiting
  - Want to go sleep if there are none, and wake up upon completion
- Split-phase operation
  - Call command to initiate action
  - Subsystem will signal event when complete
- The classic concurrent I/O problem, but also want energy efficiency.
  - Parallelism, or sleep.
  - Event-driven execution is fast and low power!



# Examples

```
/* Power-hog Blocking Call */
if (send() == SUCCESS) {
  sendCount++;
}
```

```
/* Split-phase call */
// start phase
…
 call send();
…
}
//completion phase
void sendDone(error_t err) {
  if (err == SUCCESS) {
    sendCount++;
  }
}
```

```
/* Programmed delay */
state = WAITING;
op1();
sleep(500);
op2();
state = RUNNING
```

```
state = WAITING;
op1();
call Timer.startOneShot(500);

command void Timer.fired() {
  op2();
  state = RUNNING;
```

## Sensor Readings

- Sensors are embedded I/O devices
  - Analog, digital, … many forms with many interfaces
- To obtain a reading
  - configure the sensor
    - » and/or the hardware module it is attached to,
      - ADC and associated analog electronics
      - SPI bus, I2C, UART
  - Read the sensor data
- Want applications to do this in a platform-independent manner



## Read Interface

```
interface Read<val_t> {
  /*  Initiates a read of the value.
   *  @return SUCCESS if a readDone() event will eventually come back.
   */
  command error_t read();

  /**
   * Signals the completion of the read().
   *
   * @param result SUCCESS if the read() was successful
   * @param val the value that has been read
   */
  event void readDone( error_t result, val_t val );
}
```

- Split-phase data acquisition of typed values
- Flow-control handshake between concurrent processed
  - Hardware or software
- $tinyOS-2.x/tos/interface/read.nc

# Example

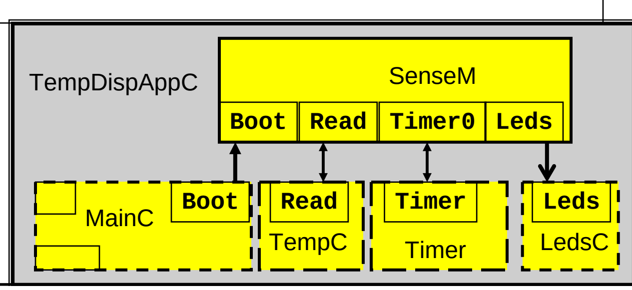
```
#include "Timer.h"
module SenseM
{
  uses {
    interface Boot;    interface Leds;    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
implementation
{
  #define SAMPLING_FREQUENCY 100
  event void Boot.booted() {
    call Timer.startPeriodic(SAMPLING_FREQUENCY);  }

  event void Timer.fired()
  {    call Read.read();   }

  event void Read.readDone(error_t result, uint16_t data)
  {
    if (result == SUCCESS){ call Leds.set(data & 0x07);}
  }
}
```

- **What does it sense?**



# Temp example configuration

```
configuration TempDispAppC
{
}
implementation {

  components SenseM, MainC, LedsC, new TimerMilliC() as Timer,
TempC ;

  SenseM.Boot  -> MainC;
  SenseM.Leds  -> LedsC;
  SenseM.Timer -> TimerMilliC;
  SenseM.Read  -> TempC;
}
```



## Uses of tasks (???)

- High speed sampling
- Filtering
- Queueing
- Smoothing
- Detection
- Classification
- …



## Sensor NETWORK

- We have a flexible, low-power, event-driven sensor / actuator platform.
- Let's add the network

- Send / Receive of information
- Dispatching incoming data to computation processes that will handle it.
  - Automate in a systematic fashion
- Parsing the packet
  - Define the structure, let the compiler do the work.
  - Handler knows what it should be receiving

## message_t structure

- Packet - Provides the basic accessors for the message_t abstract data type. This interface provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.
- Send - Provides the basic *address-free* message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.
- Receive - Provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area.
- PacketAcknowledgements - Provides a mechanism for requesting acknowledgements on a per-packet basis.
- RadioTimeStamping - Provides time stamping information for radio transmission and reception.



## Active Messages - Dispatching messages to their handlers

- AM type – dispatch selector
  - Frame_type at link layer
  - IP Protocol Field at network layer
  - Port at Transport layer
- AM_address

- AMPacket - Similar to Packet, provides the basic AM accessors for the message_t abstract data type. This interface provides commands for getting a node's AM address, an AM packet's destination, and an AM packet's type. Commands are also provides for setting an AM packet's destination and type, and checking whether the destination is the local node.
- AMSend - Similar to Send, provides the basic Active Message sending interface. The key difference between AMSend and Send is that AMSend takes a destination AM address in its send command.

## Communication Components

- AMReceiverC - Provides the following interfaces: Receive, Packet, and AMPacket.
- AMSenderC - Provides AMSend, Packet, AMPacket, and PacketAcknowledgements as Acks.
- AMSnooperC - Provides Receive, Packet, and AMPacket.
- AMSnoopingReceiverC - Provides Receive, Packet, and AMPacket.
- ActiveMessageAddressC - Provides commands to get and set the node's active message address. This interface is not for general use and changing the a node's active message address can break the network stack, so avoid using it unless you know what you are doing.



## HAL to HIL

- Since TinyOS supports multiple platforms, each of which might have their own implementation of the radio drivers, an additional, platform-specific, naming wrapper called ActiveMessageC is used to bridge these interfaces to their underlying, platform-specific implementations. ActiveMessageC provides most of the communication interfaces presented above.
- Platform-specific versions of ActiveMessageC, as well the underlying implementations which may be shared by multiple platforms (e.g. Telos and MicaZ) include:
  - ActiveMessageC for the intelmote2, micaz, telosa, and telosb are all implemented by CC2420ActiveMessageC.
  - ActiveMessageC for the mica2 platform is implemented by CC1000ActiveMessageC.
  - ActiveMessageC for the eyesIFX platform is implemented by Tda5250ActiveMessageC.

## tos/types/message.h.

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_header_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

- Link level concept used throughout the TinyOS research community and industry.
- How does this move forward to IP/WSN?



```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
}
implementation {
  uint16_t counter = 0;

  event void Boot.booted() {
    call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
  }

  event void Timer0.fired() {
    counter++;
    call Leds.set(counter);
  }
}
```

## Sending a packet to the neighborhood

```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface Receive;
  uses interface SplitControl as AMControl;
}
implementation {

  uint16_t counter;
  message_t pkt;
  bool busy = FALSE;

  event void Boot.booted() {
    call AMControl.start();
  }
```

```
 event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
    }
  }

 event void Timer0.fired() {
    counter++;
    if (!busy) {
      BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call
Packet.getPayload(&pkt, NULL));
      btrpkt->nodeid = TOS_NODE_ID;
      btrpkt->counter = counter;
      if (call AMSend.send(AM_BROADCAST_ADDR,
          &pkt, sizeof(BlinkToRadioMsg)) == SUCCESS) {
        busy = TRUE;
      }
    }
  }

  event void AMSend.sendDone(message_t* msg, error_t err) {
    if (&pkt == msg) {
      busy = FALSE;
    }
  }

  event message_t* Receive.receive(message_t* msg, void*
payload, uint8_t len){
    if (len == sizeof(BlinkToRadioMsg)) {
      BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
      call Leds.set(btrpkt->counter & 0x7);
    }
    return msg;
  }
}
```



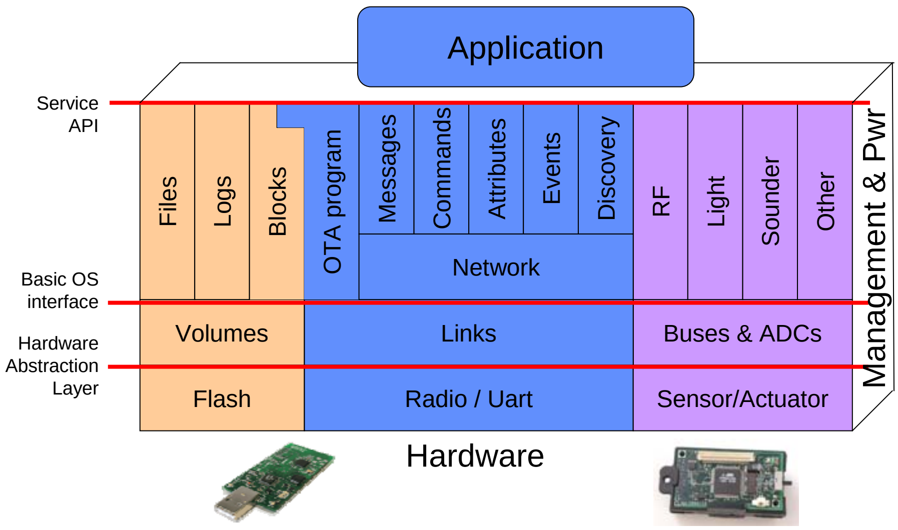
## Receive – a network event

```
event message_t* Receive.receive(message_t* msg, void* payload,
uint8_t len) {
  if (len == sizeof(BlinkToRadioMsg)) {
    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
    call Leds.set(btrpkt->counter);
  }
  return msg;
}
```

```
enum {  AM_BLINKTORADIO = 6,  };
typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;
```

- Service the incoming message
  - Automatically dispatched by type to the handler
- Return the buffer
  - Or if you want to keep it, you need to return another one.
- Overlay a network type structure on the packet so the compiler does the parsing.

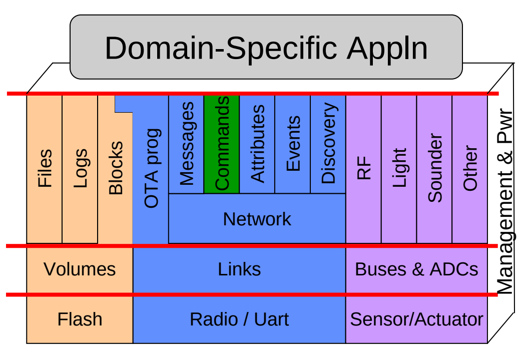
# Example TinyOS Service Architecture






# Generalized Application

- Application component contains core functionality associated with an application domain, rather than a specific application instance within that domain
  - Environmental monitoring
  - Condition based Maintenance
  - Tracking
  - …



## Permanent Data Storage

- TinyOS 2.x provides three basic storage abstractions:
  - small objects,
  - circular logs, and
  - large objects.
- also provides *interfaces* the underlying storage services and *components* that *provide* these interfaces.
- Flash devices
  - ST Microelectronics M25Pxx family of flash memories used in the Telos family of motes (tos/chips/stm25p)
  - Atmel AT45DB family of flash memories used in the Mica2/MicaZ motes (tos/chips/at45b)
  - Special pxa271p30 versions for the Intel Mote2 contributed by Arch Rock. (tos/platforms/intelmote2)
- TEP103



## Storage Interfaces and Components

- Interfaces
  - BlockRead
  - BlockWrite
  - Mount
  - ConfigStorage
  - LogRead
  - LogWrite
  - Storage.h

Components
  - ConfigStorageC - Configuration Data
    - » calibration, identity, location, sensing configuration, ..
  - LogStorageC
    - » data
  - BlockStorageC
    - » Code, …

## Volumes

- **TinyOS 2.x divides a flash chip into one or more fixed-sized *volumes* that are specified at compile-time using an XML file.**

```
<volume_table>
  <volume name="CONFIGLOG" size="65536"/>
  <volume name="PACKETLOG" size="65536"/>
  <volume name="SENSORLOG" size="131072"/>
  <volume name="CAMERALOG" size="524288"/>
</volume_table>
```



## Example – blink period config

```
typedef struct config_t {
  uint16_t version;
  uint16_t period;
} config_t;
```

Define config storage object

```
<volume_table>
  <volume name="LOGTEST" size="262144"/>
  <volume name="CONFIGTEST" size="131072"/>
</volume_table>
```

chipname.xml file

```
#include "StorageVolumes.h"
```

Added to the TinyOS configuration

```
module BlinkConfigC {
  uses {
    ...
    interface ConfigStorage as Config;
    interface Mount;
    ...
  }
}
```

New interfaces for the module

```
configuration BlinkConfigAppC {
}
implementation {
  components BlinkConfigC as App;
  components new ConfigStorageC(VOLUME_CONFIGTEST);
  ...

  App.Config     -> ConfigStorageC.ConfigStorage;
  App.Mount      -> ConfigStorageC.Mount;
  ...
}
```

Wire to the new interfaces

# On boot – Mount and Read

```
event void Boot.booted() {
  conf.period = DEFAULT_PERIOD;

  if (call Mount.mount() != SUCCESS) {
    // Handle failure
  }
}
```

```
event void Mount.mountDone(error_t error) {
  if (error == SUCCESS) {
    if (call Config.valid() == TRUE) {
      if (call Config.read(CONFIG_ADDR, &conf, sizeof(conf)) != SUCCESS) {
        // Handle failure
      }
    }
    else {
      // Invalid volume.  Commit to make valid.
      call Leds.led1On();
      if (call Config.commit() == SUCCESS) {
        call Leds.led0On();
      }
      else {
        // Handle failure
      }
    }
  }
  else{
    // Handle failure
  }
}
```

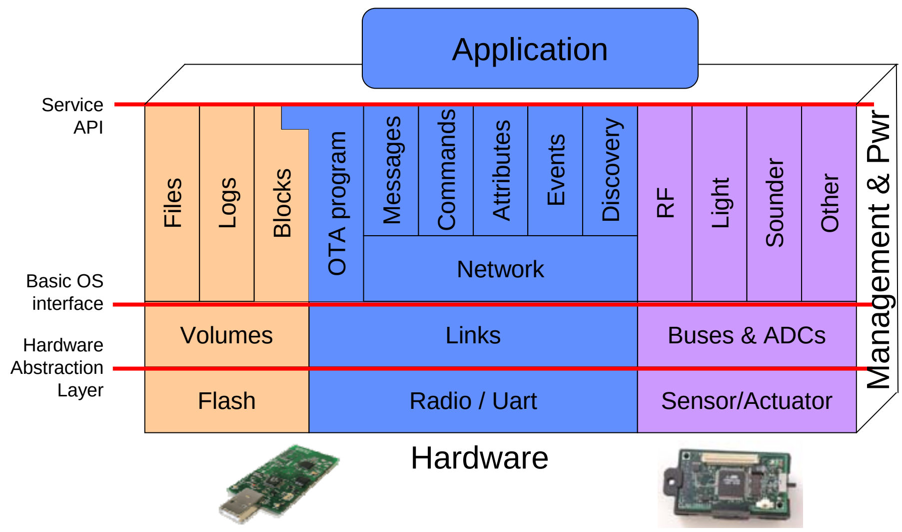


# Config data – done, write, commit

```
event void Config.readDone(storage_addr_t addr, void* buf,
  storage_len_t len, error_t err) __attribute__((noinline)) {

  if (err == SUCCESS) {
    memcpy(&conf, buf, len);
    if (conf.version == CONFIG_VERSION) {
      conf.period = conf.period/2;
      conf.period = conf.period > MAX_PERIOD ? MAX_PERIOD : conf.period;
      conf.period = conf.period < MIN_PERIOD ? MAX_PERIOD : conf.period;
    }
    else {
      // Version mismatch. Restore default.
      call Leds.led1On();
      conf.version = CONFIG_VERSION;
      conf.period = DEFAULT_PERIOD;
    }
    call Leds.led0On();
    call Config.write(CONFIG_ADDR, &conf, sizeof(conf));
  }
  else {
    // Handle failure.
  }
}
```

```
event void Config.writeDone(storage_addr_t addr, void *buf,
  storage_len_t len, error_t err) {
  // Verify addr and len

  if (err == SUCCESS) {
    if (call Config.commit() != SUCCESS) {
      // Handle failure
    }
  }
  else {
    // Handle failure
  }
}
```

# Network Embedded *Systems*

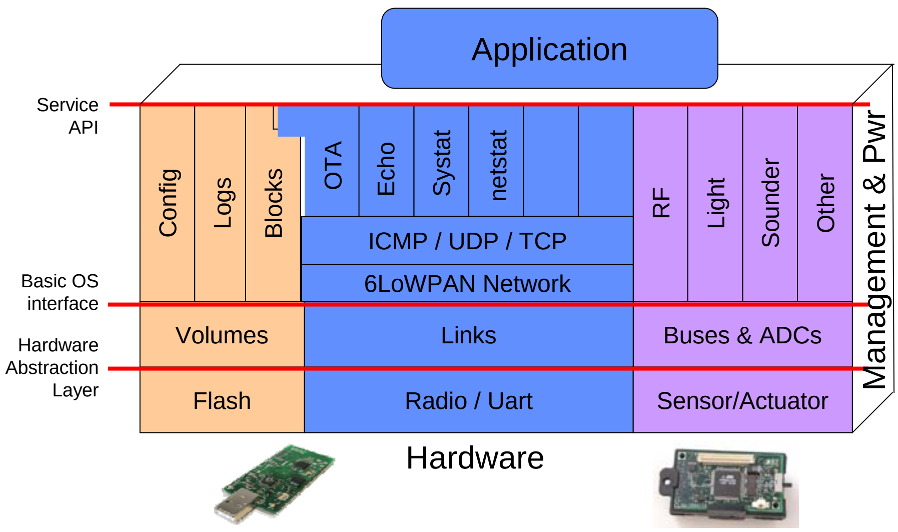





# IP/6LoWPAN "Kernel Component"

```
#include <IPv6.h>
#include <SampleEventType.h>

component BinKernel{
  uses interface Boot;
  uses interface Init as TimerInit;
  uses interface Timer<TMilli> as TimerMilli[uint8_t id];
  uses interface TaskBasic[uint8_t id];
  uses interface SplitControl as RadioSplitControl; /* Radio Control */
  uses interface LocalTime<TMilli>;                 /* Local Time */
  uses interface Leds;                              /* Leds */

  uses interface Notify<bool> as AppUserButton;     /* User Button */
  uses interface LocalIeeeEui64;                    /* EUI MAC Address */

  uses interface IPv6Addresses;                     /* IP */
  uses interface UdpSend;
  uses interface UdpAppReceive as Udp_9000;

  uses interface Read<uint16_t> as HumidityRead;    /* Sensors */
  uses interface Read<uint16_t> as TemperatureRead;
  uses interface Read<uint16_t> as LightPARRead;
  uses interface Read<uint16_t> as LightTSRRead;
}
```

# Network Embedded *Systems*






# Discussion