TinyOS Applications
TinyOS Applications Outline

- AntiTheft Example
  - LEDs, timer, booting
- Sensing Example
  - Light Sensor
  - Wiring to AntiTheft
- Single Hop Networks
  - Active Messages interface
  - Sending packets
  - Receiving packets
module AntiTheftC {
    uses {
        interface Boot;
        interface Timer <Tmilli> as WarningTimer;
        interface Leds;
    }
}

implementation {
    enum {
        WARN_INTERVAL = 4096, WARN_DURATION = 64
    };
}
event void WarningTimer.fired () {
    if (call Leds.get () & LEDS_LED0) {
        /* Red LED is on. Turn it off, will switch on again in 4096 - 64 ms. */
        call Leds.led0Off ( );
        call WarningTimer.startOneShot (WARN_INTERVAL - WARN_DURATION);
    }
    else {
        /* Red LED is off. Turn it on for 64 ms. */
        call Leds.led0On ( );
        call WarningTimer.startOneShot (WARN_DURATION);
    }
}
event void Boot.booted() {
    /* We just booted. Perform first LED transition */
    signal WarningTimer.fired();
}

interface Leds {
    ...

    async command void led0On();
    async command void led0Off();
    async command uint8_t get();
}

AntiTheft Example [List 6.1]

Advanced Computer Networks  TinyOS Applications
configuration AntiTheftAppC { }
implementation {
    components AntiTheftC, MainC, LedsC;
    components new TimerMilliC ( ) as WTimer;

    AntiTheftC.Boot  -> MainC;
    AntiTheftC.Leds  -> LedsC;
    AntiTheftC.WarningTimer -> WTimer;
}
Sensing Example

- TinyOS provides two standard interfaces for reading sensor samples
  - `Read ::` acquire a single sample
  - `ReadStream ::` sample at a fixed rate.

```c
interface Read <val_t> {
    command error_t read ( );
    event void readDone (error_t, val_t val );
}
```
module DarkC {
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli> as TheftTimer;
    interface Read<uint16_t> as Light;
  }
}
implementation {
    enum { DARK_INTERVAL = 256, DARK_THRESHOLD = 200};

    event void Boot.booted ( ) {
        call TheftTimer.startPeriodic (DARK_INTERVAL);
    }

    event void TheftTimer.fired ( ) {
        call Light.read ( ); //Initiate split-phase light sampling
    }
}
/* Light sample completed. Check if it is a theft. */

event void Light.readDone (error_t ok, uint16_t val) {

    if (ok == SUCCESS && val < DARK_THRESHOLD) {
        call Leds.led2On(); /* Alert! Alert! */
    } else {
        call Leds.led2Off(); /* Don't leave LED on */
    }
}

Sensors are represented in TinyOS by generic components, e.g., `PhotoC` for the light sensor on the mts310 board.

generic configuration `PhotoC ( ) {`
  provides interface `Read<uint16_t>;`
}
configuration AntiTheftAppC { }
implementation {
...  /* the wiring for the blinking Red LED */
components DarkC;
components new TimerMilliC () as TTimer;
components new PhotoC ();

DarkC.Boot -> MainC;
DarkC.Leds -> LedsC;
DarkC.TheftTimer -> Ttimer;
DarkC.Light -> PhotoC;
}
Single Hop Networks

- TinyOS uses a layered network structure where each layer defines a header and footer layout.

- The lowest exposed network layer in TinyOS is called *active messages (AM)*.

- AM is typically implemented directly over a mote’s radio providing unreliable, single hop packet transmission and reception.
Single Hop Networks

- Packets are identified by an 8-bit packet type.
- 'Active Messages' indicates the type is used automatically to dispatch received packets to an appropriate handler.
- Each packet holds a user-specified payload of up to \texttt{TOSH\_DATA\_LENGTH} bytes (normally 28 bytes)**.
- A variable of type \texttt{message\_t} holds a single AM packet.

** changeable at compile time.
Platform-Independent Types

- TinyOS has traditionally used structs to define message formats and directly access messages.
- Platform-independent structs are declared with `nx_struct` and every field of a platform-independent struct must be a platform-independent type.

```c
nx_uint16_t val;         // A big-endian 16-bit value
nxle_uint32_t otherval;  // A little-endian 32-bit value
```
typedef nx_struct cc2420_header_t ** {
    nxle_uint8_t  length;
    nxle_uint16_t fcf;
    nxle_uint8_t  dsn;
    nxle_uint16_t destpan;
    nxle_uint16_t dest;
    nxle_uint16_t src;
    nxle_uint8_t  type;
} cc2420_header_t;

The CC2420 expects all fields to be little-endian.
Platform-independent struct in the antitheft.h header file:

```c
#ifndef ANTITHEFT_H
#define ANTITHEFT_H
typedef nx_struct theft {
    nx_uint16_t who;
} theft_t;

... #endif
```

struct to define payload
AMSend Interface

- Contains all the commands needed to fill in and send packets:

```c
interface AMSend {
    command error_t send (am_addr_t addr, message_t* msg, uint8_t len);
    event void sendDone (message_t* msg, error_t error);
    command error_t cancel (message_t* msg);
    command uint8_t maxPayloadLength ( );
    command void* getPayload (message_t* msg, uint8_t len);
}
```
uses interface AMSend as Theft;
...

message_t reportMsg;
bool sending;

void reportTheft ( ) {
  theft_t* payload = call Theft.getPayload (&reportMsg,
                                             sizeof (theft_t));

  if (payload && !sending)
    { //Payload fits and we are idle - Send packet
      payload->who = TOS_NODE_ID; //Report being stolen!
      //Broadcast the report packet to everyone
      if (call Theft.send(TOS_BCAST_ADDR, &reportMsg,
                          sizeof (theft_t)) == SUCCESS)
    }
}
event void Theft.sendDone (message_t *msg, error_t error) {
    sending = FALSE; //Our send completed
}
Generic AMSenderC configuration

generic configuration AMSenderC (am_id_t AMId) {
    provides {
        interface AMSend;
        interface Packet;
        interface AMPacket;
        interface PacketAcknowledgements as Acks;
    }
}
Cannot switch itself on and off on-demand, and needs the `SplitControl` interface to start and stop the radio:

```c
interface SplitControl {
    command error_t start ( );
    event void startDone (error_t error);

    command error_t stop ( );
    event void stopDone (error_t error);
}
```

[List 6.14]
MovingC using SplitControl

uses interface SplitControl as CommControl;

...

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

event void CommControl.startDone (error_t ok) {
    //Start checks once communication stack is ready
    call TheftTimer.startPeriodic (ACCEL_INTERVAL);
}

event void CommControl.stopDone (error_t ok) { }
Moving C Receiving Packet

- **MovingC** receives a packet payload (defined as a `struct` contained in a header file) that contains acceleration settings for detecting movement of the mote:

```c
typedef nx_struct settings {
    nx_uint16_t acceVariance;
    nx_uint16_t accelInterval;
} settings_t;
```

**struct to define payload**
Provided by the TinyOS Receive interface:

```c
interface Receive {
    event message_t* receive(message_t* msg, void* payload, uint8_t len);
}
```

`Receive.receive`, as a receive “handler”, receives a packet buffer which it can simply return or return as a different buffer if the handler wants to hold onto buffer.
uses interface Receive as Setting;

...  

uint16_t accelVariance = ACCEL_VARIANCE;

event message_t *Settings.receive (message_t *msg,  
    void *payload, uint8_t *len) {  
  if (len >= sizeof (settings_t)) //Check for valid packet  
    { /* Read settings by casting payload to settings_t,  
      reset check interval */  
      settings_t *settings = payload;  
      accelVariance = setting->accelVariance;  
      call TheftTimer.startPeriodic (setting->accelInterval);  
    }  
  return msg;  
}
TinyOS Applications Summary

- **AntiTheft Example**
  - LEDs, Timer, Boot
  - `get`, `enum`

- **Sensing Example**
  - Light Sensor
  - Read (split-phase)
  - Wiring to AntiTheft
  - Two Timer instances
TinyOS Applications Summary

- Single Hop Networks
  - Active Messages, typed messages
  - Platform-independent types

- Sending packets
  - AMSenderC generic configuration
  - SplitControl of Radio Stack
  - Structs for packet payloads

- Receiving packets
  - Implemented as a receive event handler.