



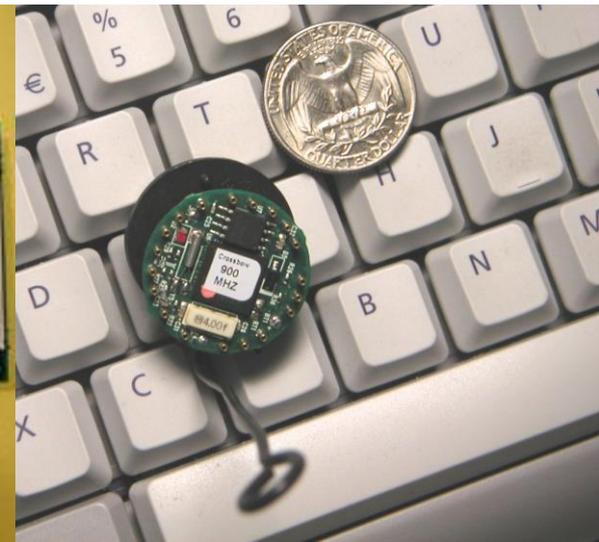
**TINYOS & NESC
INTRODUZIONE ALLA
PROGRAMMAZIONE DI WSN**

Gentilmente concesse da
Claudio Vairo

SENSORI

Piccoli dispositivi multi-funzionali con capacità di calcolo, risorse ed energia limitati.

Diverse misure, ma in genere piccoli



MICRO-CONTROLLORE

Processore: 8-bit, 8MHz

Memoria RAM: 8-10KB

Memoria Flash (per il codice): 48-128KB

Memoria Flash (per storage): 1MB

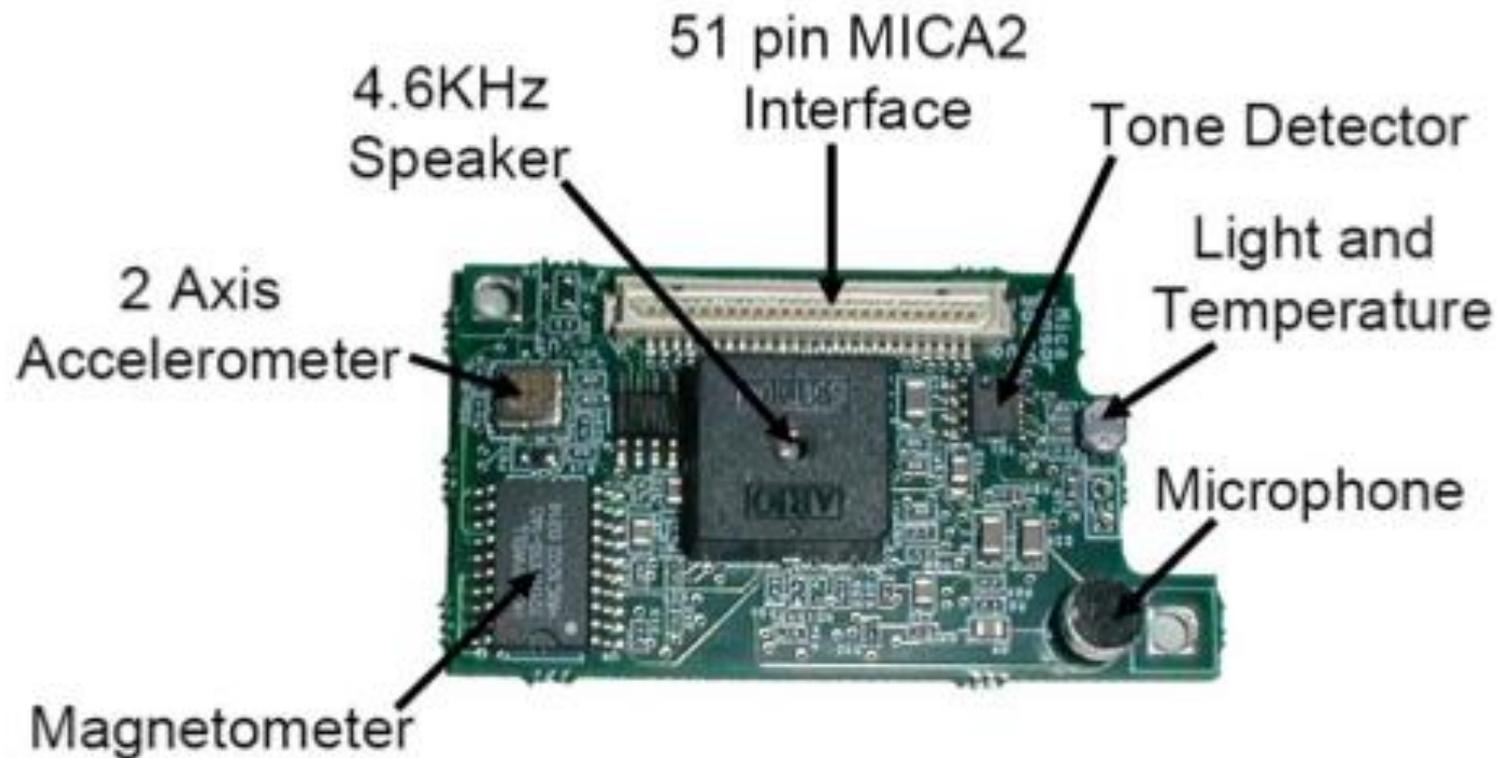
Interfaccia radio Wireless: 2.4GHz, 802.15.4 compliant, 250Kbps data rate

Alimentati con 2 batterie AA

3 Leds



SENSOR BOARD



TINYOS

Un sistema operativo opensource minimale progettato per reti di sensori wireless

Architettura a componenti basata su nesC per minimizzare l'uso di memoria

Modello di esecuzione ad eventi, che fornisce buon controllo dell'energia e dello scheduling

Astrazione dell'hardware

NESC

Estensione del linguaggio C

Paradigma orientato a eventi

Linguaggio orientato ai componenti

- Composizione tramite interfacce bi-direzionali
- Wiring dei componenti

Allocazione statica della memoria

- Linking statico dei componenti
- Vietata `malloc` e ricorsione

Controllo statico della concorrenza

- Task e eventi

TINYOS - INSTALLAZIONE

Disponibile su tutte le piattaforme

- [Windows installation using cygwin](#)
- [Debian installation using apt-get](#)
- [Snow Leopard installation using Mac Ports](#)

Configurare le variabili d'ambiente

- aggiungere `source /opt/tinyos-2.1.1/tinyos.sh` in `.bashrc` o `.profile`

Assicurarsi che il file `/opt/tinyos-2.x/support/sdk/java/tinyos.jar` sia nel `CLASSPATH`

TINYOS - APPLICAZIONE

- TinyOS permette l'esecuzione di una sola applicazione
- Insieme di **componenti** legati tra loro tramite **interfacce**
- File di **configurazione** che gestisce il wiring
- Un unico codice statico viene generato grazie al binding fornito dal wiring

INTERFACCE

Punto di accesso per interazione tra componenti

Bi-direzionali

Definiscono *comandi* ed *eventi*

- **Comandi** – Specificano una funzionalità offerta
 - Implementati dai componenti
 - Usati da altri componenti per invocare azioni sul componente (down-call)
- **Eventi** – Notificano il verificarsi di un evento
 - Generati dai componenti
 - Devono essere gestiti da chi usa il componente (up-call)

INTERFACCE (2)

Possono essere *usate* o *fornite* dai componenti

- Il componente che **usa** l'interfaccia:
 - **può** *invocare* comandi
 - **deve** implementare il gestore degli eventi per ognuno degli eventi definiti nell'interfaccia
- Il componente che **fornisce** l'interfaccia
 - **deve** implementare tutti i comandi definiti nell'interfaccia
 - **può** *segnalare* l'occorrenza degli eventi

COMPONENTI

Unità software dotata di un proprio stato e che svolge un determinato compito

Moduli

- Definisce le interfacce usate e/o fornite dal componente
- Contiene l'implementazione del componente
 - Gestore di tutti gli eventi definiti nelle interfacce *usate*
 - Implementazione di tutti i comandi definiti nelle interfacce *fornite*
 - Codice interno del componente: variabili, funzioni, ecc...

COMPONENTI (2)

Configurazioni

- Una configurazione per ogni modulo
- Dichiarare i componenti usati dal modulo
- Definisce il wiring: collega le interfacce usate dal modulo con quelle fornite da altri componenti

ESEMPIO: BLINK

Configurazione: `BlinkAppC.nc`

Modulo: `BlinkC.nc`

Makefile: `Makefile`

- Nel makefile specificare il nome del componente di **configurazione** dell'applicazione (in questo caso `COMPONENT=BlinkAppC`)

Compilare con "make <platform>"

- Per esempio `make iris`

ESEMPIO: BLINK (2)

In output abbiamo occupazione in RAM e occupazione della flash per il programma

- **Attenzione a non sfiorare!!!**

Il risultato della compilazione è un file binario (`build/iris/main.exe`) che include TinyOS e tutti i componenti dell'applicazione

Flashare il binario con:

- `make iris reinstall.ID_NODO mib510,SERIAL_PORT`
- `make telosb reinstall.ID_NODO bsl,SERIAL_PORT`
- `ID_NODO` è l'indirizzo assegnato al sensore, `SERIAL_PORT` è la porta serial a cui è collegato il sensore

BLINK - CONFIGURATION

```
configuration BlinkAppC {  
}  
implementation {  
    components MainC, BlinkC, LedsC;  
    components new TimerMilliC() as Timer0;  
    components new TimerMilliC() as Timer1;  
    components new TimerMilliC() as Timer2;  
  
    BlinkC.Boot -> MainC.Boot;  
    BlinkC.Timer0 -> Timer0;  
    BlinkC.Timer1 -> Timer1;  
    BlinkC.Timer2 -> Timer2;  
    BlinkC.Leds -> LedsC;  
}
```

WIRING DI COMPONENTI

Tramite l'operatore '->'

- La parte a sinistra specifica il componente che *usa* l'interfaccia
- La parte destra specifica il componente che *fornisce* l'interfaccia

Esempio nel Blink:

- `BlinkC.Boot -> MainC.Boot`
- L'interfaccia `Boot` *usata* nel modulo `BlinkC` viene *fornita* dal componente `MainC`

BLINK - MODULE

```
module BlinkC {  
  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  
{  
  // next slide  
}
```

BLINK - MODULE

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();  
}  
}
```

SENSING

Nel componente modulo *usare* l'interfaccia generica Read

- `uses interface Read<uint16_t> as Light;`

Interfaccia `Read<val_t>`

- `command error_t read();`
- `event void readDone(error_t result, val_t val);`

Nel componente configurazione istanziare il componente che *fornisce* il driver relativo al trasduttore che si vuole leggere e fare il wiring corrispondente

- `components new SensorMts300C() as Light;`
- `LocalSenseC.Light -> Light.Light;`

Nel componente modulo *invocare* la `read()` e aspettare le notifica della `readDone()`

SPLIT-PHASE

Tecnica che realizza una forma comunicazione asincrona tra componenti

Usata per eseguire operazioni bloccanti o di lunga durata

Dividere l'operazione in due fasi

- Invocazione (comando che termina subito)
 - `call Light.read();`
- Restituzione del risultato (notifica di un evento)
 - `event void Light.readDone(error_t result, uint16_t val) {}`

Non blocca il sistema

ESEMPIO: LOCALSENSE

Legge il valore di luce ogni 500 millisecondi

Se il valore letto è superiore a una data soglia “switcha” lo stato dei led

COMPONENTI GENERIC

I componenti di default in TinyOS sono singleton

- Una sola istanza per tutta l'applicazione

Necessità di più istanze di uno stesso componente

- Timer, Read

Generic Component

- Possono essere istanziati più volte (con `new`)
- Importante definire l'alias!

CONCORRENZA

Gli eventi sono asincroni e prerilasciabili

- Esempio: ricezione di messaggi temporalmente vicini tra loro

Come si gestisce la concorrenza?



☞ Si usano i **Task** e gli **atomic** statement

TASK

Ogni task è garantito essere atomico rispetto agli altri task

- Politica di tipo **run-to-completion**

Usati per lunghe computazioni

Hanno scope di componente

Sono funzioni void senza parametri

- Va usato il contesto per “passare i parametri”

Implementati tramite una coda con politica FIFO

Un thread apposito gestisce questa coda

Un task viene “attivato” (inserito in coda) tramite la keyword **post**

TASK (2)

In TinyOS 1 era permessa una sola istanza di un task in coda **o in esecuzione**

- Non si poteva postare un task di tipo t se un'altra istanza di task dello stesso tipo t era già presente nella coda dei task **o era in esecuzione**
 - Un task non poteva postare un'altra istanza di se stesso

In TinyOS 2 è permessa una sola istanza di un task in coda

- Il task in esecuzione può postare un'altra istanza di se stesso
 - Non più di una perchè il controllo sulla coda rimane

MODELLO DI ESECUZIONE DI NESC

EventX: un thread gestisce gli handler degli eventi

TaskX: un thread gestisce i task

EventX ha priorità più alta di TaskX:

- Il TaskX può essere descheduled dal EventX, ma non dal TaskX
- Il EventX può essere descheduled da altri eventi

COSA SI FA IN PRATICA

Gli eventi sono più corti possibile

- Preparano l'“ambiente” e postano il task

Il task esegue la computazione “lunga”

- Eventualmente posta un'altra istanza di se stesso se la computazione va spezzettata

OSSERVAZIONE: l'ambiente è globale

PROBLEMA:

- 🌀 Cosa succede se nella gestione di un evento arriva la notifica di un altro evento che modifica le stesse variabili?

ATOMIC STATEMENT

```
atomic {  
    read_variable;  
    update_variable;  
}
```

La parte di codice dentro il blocco `atomic` è garantita essere atomica **anche nei confronti degli eventi**

- Ne ritardano l'esecuzione
- Non devono essere lunghe computazioni

STACK DI COMUNICAZIONE

Active Message (AM) è la tecnologia di comunicazione di TinyOS

- Comunicazione radio (mote-to-mote)
- Comunicazione seriale (mote-pc, bidirezionale)
- Implementa una comunicazione basata sul canale
 - In un canale un solo tipo di messaggio
- Ogni pacchetto è caratterizzato da un **AM_type** (intero a 8 bit)
- Identifica il canale su cui viaggia il messaggio
 - Indirettamente identifica anche i componenti per l'invio e la ricezione

MESSAGE_T

```
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_footer_t)];
    nx_uint8_t
metadata[sizeof(message_metadata_t)];
} message_t;
```

La grandezza di default del campo data è 28 bytes

- Si può aumentare, ma aumenta anche il rischio di collisioni poichè la radio è occupata per più tempo

Per accedere ai campi di message_t si devono usare apposite interfacce

INTERFACCE

Si usano le interfacce:

- `AMSend`, `Receiver`, `AMPacket`, `Packet` e `SplitControl`
- `AMSend`: per l'invio
- `Receiver`: per la ricezione
- `AMPacket`: set e get di informazioni sul messaggio
 - Sorgente e destinazione, `AM_type`, etc..
- `Packet`: accesso al payload del messaggio
- `SplitControl`: accende/spegne l'interfaccia di comunicazione

Tutte queste funzionano indifferentemente sia per la radio che per la seriale

- Sta al wiring decidere a quale interfaccia di comunicazione legarle

COMPONENTI

I componenti che forniscono le interfacce necessarie sono:

- `AMSenderC()`, `AMReceiverC()`, `ActiveMessageC` e i corrispettivi per la seriale -> stesso nome ma con `serial` davanti

I componenti `AMSenderC()` e `AMReceiverC()` vanno istanziati uno per ogni tipo di messaggio da inviare/ricevere e parametrizzati con l'`AM_type` corrispondente

- In questo modo si ottiene anche il dispatch automatico dei messaggi

```
components new AMReceiverC(AM_LIGHT_MSG) as  
LightReceive;
```

```
event message_t* LightReceive.receive(message_t*  
msg, void* payload, uint8_t len) {}
```

MESSAGGI

Per ogni tipo di messaggio va definita una struct dati di tipo `nx_struct` con tutti i campi `nx_*`

- `nx_*` identifica un external type che usa una rappresentazione platform-independent
- Usati per le comunicazioni tra dispositivi

Il nome della struct deve essere: `<nome-messaggio>msg`

Bisogna definire l'`AM_type` corrispondente

- Da usare poi come parametro dell'interfacce per inviare/ricevere quel tipo di messaggio
- Dev'essere: `AM_<nome-messaggio>_MSG`