SMART EXPERIMENT

Sistemi embedded & internet of things

Giorgia Castelli 873787
Lorenzo Pisanò 900590

Progetto #2 - Smart Experiment

Il sistema è basato su un approccio a task, con modello di comportamento basato su macchine a stati finiti sincroni. Ad ogni task infatti viene associato un quanto di tempo che caratterizza la sua esecuzione.

E' presente dunque una classe scheduler che si occupa di gestire la politica di scheduling pensata per questo progetto. Ad ogni tick, che avviene ogni 50 ms (stabiliti nell'inizializzazione della classe stessa nel file .ino), lo scheduler manda in esecuzione il task che si trova nello stato attivo nel momento del controllo. Ogni task, ha il compito di segnalare allo scheduler quale sarà il prossimo processo che dovrà andare in esecuzione. Questa decisione è stata presa a seguito dell'analisi delle specifiche che richiedevano, in alcuni casi, di transitare in diversi task in base allo stato precedente. Si è ritenuto necessario quindi aggiungere tutte le dipendenze dei vari stati allo scheduler. Così facendo, lo scheduler evita di conoscere l'oggetto in esecuzione e si occupa solamente di ricavare, attraverso un getter della classe Task, il prossimo stato che dovrà andare in esecuzione.

Durante il super-loop dello scheduler possono accadere due principali casistiche all'esecuzione del task;

- Può essere interrotto
- Può essere completato

L'interruzione viene gestita attraverso l'utilizzo di un interrupt handler ed una classe Interrupts. All'interno di quest'ultima vengono stabiliti tutti gli agenti esterni che, durante l'esecuzione, possono generare un'interruzione del task in esecuzione.

```
void Interrupts::init(){
  enableInterrupt(8, handleInterruptsStart, RISING);
  enableInterrupt(12, handleInterruptsStop, RISING);
  enableInterrupt(7, handleInterruptsPir, RISING);
}
```

Facendo una breve panoramica del funzionamento delle interruzioni, una volta che viene cliccato un bottone (pin digitali 8/12), oppure il PIR, si rileva un movimento (pin digitale 7) e viene richiamato un metodo della classe Interrupts. Questo, a sua volta, modifica una variabile definita all'interno nella stessa classe visibile alla classe scheduler attraverso la keyword external.
Ad ogni tick dello scheduler vengono controllate, attraverso il costrutto if, lo stato di queste variabili. Quando vengono notificate, allora vengono concretizzate, ma solo nel caso in cui si trovino nello stato definito nelle specifiche.

```
}else if(InterruptPir == true && i == 2){
   InterruptPir = false;
   taskList[i] -> setFirstRun(false);
   Scheduler::redirectTask(taskList[i] -> getNextTask());
}
```

In questo esempio, l'interruzione generata del pir viene presa in considerazione solo nel caso in cui lo stato si trovi ad indice 2, che corrisponde allo stato di sleep.

SetFirstRun invece è un metodo della classe Task che ci permette di comprendere quando il task si trova nel primo tick di esecuzione. Nel caso venisse eseguito per la prima volta (dopo transizione di stato), è necessario cambiare lo stato di accensione dei led.

Infine, RedirectTask è un metodo della classe Scheduler necessario per gestire le casistiche prima descritte. Prende in input il futuro task da eseguire e lo imposta ad uno stato pari ad attivo, in modo tale che al prossimo tick, verrà preso in considerazione dallo scheduler.

Un altro aspetto che è stato affrontato, ha riguardato, lo stato di sleep del microprocessore. In particolare, si è scelto di utilizzare la power mode state, tenendo sempre in ascolto l'interrupt Handler. La conclusione di questa task è condizionata dall' occorrenza di un evento (come visto prima). Questo viene pensato per garantire il massimo risparmio energetico al microprocessore.

	Active Clock Domains					Oscillators		Wake-up Sources							Software BOD Disable
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT and PCINT	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other I/O	
Idle			Yes	Yes	Yes	Yes	Yes[2]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
ADC Noise Reduction				Yes	Yes	Yes	Yes[2]	Yes[3]	Yes	Yes[2]	Yes	Yes	Yes		
Power-down								Yes[3]	Yes				Yes		Yes
Power-save					Yes		Yes[2]	Yes[3]	Yes	Yes			Yes		Yes
Standby[1]						Yes		Yes[3]	Yes				Yes		Yes
Extended Standby					Yes[2]	Yes	Yes[2]	Yes[3]	Yes	Yes			Yes		Yes

Secondo quanto studiato e confermato da questo articolo, la power down è il più alto livello di risparmio. Tutte le funzionalità vengono disattivate ad eccezione dell'interrupt handler. Essendo questo lo stato che permette il più basso utilizzo di energia da parte dell'arduino, è ovviamente questo lo stato che necessita di più tempo per riattivare tutti i suoi componenti -in precedenza disattivati- e ritornare allo stato di sua massima efficienza.

Altro aspetto preso in considerazione è quello legato alla gestione del potenziometro. Questo è fortemente condizionato dalle costanti, definite in fase di analisi di specifica, MINFREQ e MAXFREQ.

Viene richiesta la frequenza con la quale deve essere valutata la distanza dell'oggetto in esame dal sensore sonar. Questa viene regolata in base al potenziometro ed ai dati prima presentati.

Il componente potenziometro 10k ha un range di valore che varia da 1 a 1024. Attraverso l'equazione 1:1024=MINFREQ:MAXFREQ si modula il valore in base alle costanti definite in design time (offline).

Le frequenze che variano da minfreq a maxfreq vengono suddivise in quattro macro gruppi che definiscono la frequenza media dei valori estremi degli insiemi.

Infine, viene definita, sempre in fase di design, la VEL_MAX (VEL_MIN come definita dalle specifiche. Questa, sarà pari a 30. Vengono anche in questo caso modulate le velocità attraverso l'equazione 1:180=1:30. Si deduce, quindi, che l'ampiezza massima del servomotore (180°) sarà raggiunta nel caso in cui la velocità istantanea dell'oggetto in questione sarà pari a 30. L'unità presa in esame per questo esperimento, considerando l'ambiente dove viene svolto ed i componenti in possesso, è di cm/s.

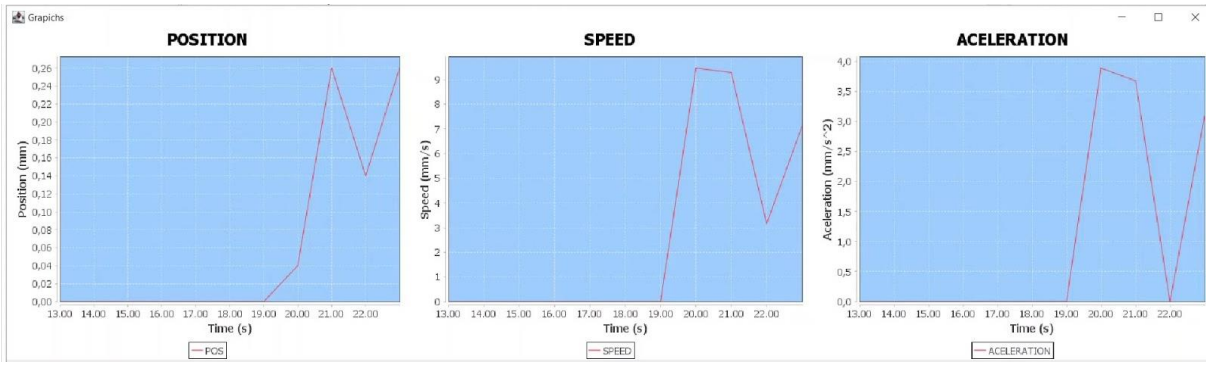
Prendendo in considerazione la poca memoria flash disponibile su Arduino, si preferisce mantenere lo storico della posizione, velocità ed accelerazione dell'oggetto in movimento nel programma java. In fase di running infatti, ad ogni tick, il programma arduino invia tramite seriale, la posizione attuale, in metri, dell'oggetto ed il tempo, in millisecondi. Così facendo viene salvato su arduino il minimo indispensabile per effettuare i calcoli necessari.

In sintesi, vengono previste due strutture dati di 4 posizioni (0...3) necessarie per tenere in memoria la posizione ed il tempo in cui è stato registrato il dato. Ogni quattro registrazioni vengono dunque sostituiti i valori preesistenti. Questo è necessario farlo per calcolare la velocità istantanea, che deve essere riportata dal servomotore attraverso il movimento del rotore.

Per quanto riguarda l'integrazione con il programma java, è stata utilizzata la libreria jscc. Tramite questa libreria è possibile mettere in comunicazione Arduino con il programma su PC in Java tramite seriale.

Viene utilizzata anche la classe "SerialCommChannel.java" che sfrutta la libreria jssc per aprire la porta seriale e leggere i messaggi ricevuti in formato String.

I dati calcolati e memorizzati temporaneamente su Arduino vengono inviati al viewer che, oltre a visualizzarli in modo grafico, si occupa di salvarli all'interno dei relativi arraylist. Prima di un nuovo esperimento questi vengono svuotati per poter memorizzare i nuovi dati, tuttavia rimane la possibilità di visualizzare i grafici degli esperimenti precedenti.



Oltre a comunicare i dati calcolati, ogni volta in cui l'esperimento cambia lo stato in cui si trova, Arduino invia un messaggio al viewer, il quale visualizza tramite terminale il messaggio relativo allo stato ricevuto.

```
while(true) {
    String msg = channel.receiveMsg();
    switch(msg) {
    case("IDLE"):
        System.out.println("IDLE Press start button");
        break;
    case("ERROR"):
        System.out.println("ERROR: Object not detected");
        break;
    case("SLEEP"):
        System.out.println("SLEEP: Going in sleep-mode");
        break;
    case("RUNNING"):
        /*
         * una volta entrato in stato di running tramite il metodo running(...)
         * viene visualizzato il pannello contenente i tre grafici
         *
         * ogni volta che l'esperimento ricomincia viene visualizzato un nuovo grafico
         */
        System.out.println("THE EXPERIMENT IS RUNNING");
        msg = channel.receiveMsg();
        pot = Float.parseFloat(msg);
        running(channel, pot);
        break;
    case("COMPLETED"):
        System.out.println("THE EXPERIMENT IS COMPLETED");
        break;
    }
```

Per la realizzazione dei grafici viene utilizzata la libreria **jfree.chart** che offre la possibilità di creare grafici in tempo reale, aggiornandoli periodicamente in base alla gestione di un timer. Questo timer ha come periodo la frequenza definita da Arduino, che viene inviata al programma in java appena l'esperimento si trova in stato di running. Dopo questo primo messaggio, quelli inviati successivamente e per tutto lo stato di running, contengono i dati calcolati nel seguente pattern:

"posizione | velocità | accelerazione"

Successivamente java si occupa di fare lo split della stringa ricevuta e memorizzare i dati nei rispettivi arraylist per poi aggiungerli ai grafici tramite il metodo appendData().

```
timer = new Timer((int)pot, new ActionListener() {

    float[] newDataSpeed = new float[1];
    float[] newDataAcel = new float[1];
    float[] newDataPos = new float[1];

    @Override
    public void actionPerformed(ActionEvent e) {
            try {
                msg = channel.receiveMsg();
            } catch (InterruptedException e1) {e1.printStackTrace();}
            if (msg.equals("COMPLETED")) {
                ((Timer)e.getSource()).stop();
                System.out.println("THE EXPERIMENT IS COMPLETED")
                data.delData();
            }else {
                String[] splitMsg = msg.split("\\|",3);
                data.addPos(Float.parseFloat(splitMsg[0]));
                data.addSpeed(Float.parseFloat(splitMsg[1]));
                data.addAcel(Float.parseFloat(splitMsg[2]));
```

```
datasetPos.advanceTime();
datasetPos.appendData(newDataPos);

datasetSpeed.advanceTime();
datasetSpeed.appendData(newDataSpeed);

datasetAcel.advanceTime();
datasetAcel.appendData(newDataAcel);
```

Per poter aggiungere i dati ai grafici jfree.chart utilizza dei data set, quindi prima di creare la struttura del grafico è necessario creare tre differenti data set, che saranno poi aggiornati periodicamente nel action listener del timer e poi aggiunti ai grafici.

```
final DynamicTimeSeriesCollection datasetSpeed =
    new DynamicTimeSeriesCollection(1, 500, new Second());
datasetSpeed.setTimeBase(new Second(0, 0, 0, 1, 1, 2020));
datasetSpeed.addSeries(new float[1], 0, "SPEED");
```