Relazione Smart Coffee Machine

Introduzione

Lo scopo del progetto era quello di realizzare, attraverso la scheda programmabile Arduino e la comunicazione seriale con una applicazione software realizzata in Java, un simulatore di una macchina del caffè intelligente, che attraverso la rilevazione di distanza e movimento permetta un interazione automatizzata con l'utente.

Progettazione

Nella fase di design del progetto, abbiamo deciso di organizzarlo in task, ognuno con un compito specifico. Nel dettaglio abbiamo strutturato un main task, cioè il task principale che gestirà gli stati del sistema, svolgendo un compito di controllore del sistema e che avrebbe impostato le variabili globali condivise in modo da far eseguire gli altri task nel momento opportuno. Questi altri task possono essere divisi in due tipologie:

- Task di rilevazione : Sono task che assegnano il valore letto dai sensori alle corrispettive variabili condivise, in modo da permettere al main task di avere i dati dei sensori sempre aggiornati, ma senza doversene occupare lui direttamente.
- Task di esecuzione : In modo differente, questi task vengono "avviati" dal main task ed eseguono delle routine come il lampeggiamento dei led e la comunicazione con la piattaforma in Java. In modo più accurato, questi task vengono eseguiti dallo scheduler in continuazione come gli altri task e con stessa priorità, ma al loro interno hanno un semplicissimo controllo sulla variabile condivisa rispettiva e nel caso in cui si a impostata a false, il task non esegue niente. Altrimenti viene eseguita la routine specificata.

Durante questa fase, ci siamo imbattuti nella decisione di dover eseguire la modalità di risparmio energetico nello stato STAND BY. Inizialmente avevamo pensato una soluzione in cui tale operazione veniva gestita da un task a se stante, ma dopo varie considerazione abbiamo deciso per l'inserimento della sleep mode all'interno del main task, in quanto essendo il task principale era giusto che una funzionalità importante come questa fosse gestita al suo interno. Per la sua implementazione successiva c'erano due possibilità:

- L'assegnamento di un interrupt al sensor PIR, che risvegliasse il micro controllore quando veniva rilevato un movimento.
- Una sleep mode ciclica che si risvegliasse periodicamente con il timer di esecuzione dello scheduler.

Fra le due soluzioni abbiamo optato per la seconda in quanto abbiamo voluto mantenere il sistema basato su macchina totalmente sincrona, evitando qualsiasi elemento asincrono.

Architettura

Siamo partiti con una progettazione orientata agli oggetti e che al suo interno includeva la creazione di più task, ognuno con un determinato compito e la possibilità di comunicare con gli altri attraverso delle variabili globali condivise. Per la realizzazione del sistema abbiamo implementato i seguenti task, che rispecchiano il funzionamento delle FSM mostrato nelle immagini rispettive.

- **MovementTask**: Il MovementTask incapsula la logica per rilevare il movimento, che ha lo scopo di risvegliare la macchina. Contiene una classe PirSensor che implementa la logica di utilizzo del sensore PIR. Quando si crea una nuova istanza di esso bisogna dargli in ingresso il pin a cui il sensore(PIR) è agganciato. Quando il MovementTask viene inizializzato si richiama la sua funzione init() che prende in ingresso un periodo il quale specifica il quanto di tempo che intercorre tra due rilevamenti. Tale task viene eseguito quando il sistema si trova nello stato di **STAND BY**. Il MovementTask ad ogni tick del periodo assegna alla variabile globale booleana movement che viene inizializzata nel setup l'input preso dal PIR. Se il sensore rileva movimento cioè, arriva in input un valore HIGH la variabile movement viene impostata a true dal task altrimenti rimane a false come inizializzata in principio. Il task ha come stato iniziale **M0** che simula il fatto che non ci sia presenza, nel momento in cui viene rilevato un primo movimento passa a stato **M1**, per motivi di sicurezza si è deciso di gestire l'effettivo movimento solo dopo che il PIR ha preso per due volte un valore HIGH, per questo il task passa da stato **M1** a stato **M2** dove viene impostsata la variabile movement. Sia da **M1** che da **M2** si torna allo stato **M0** se il PIR non rileva più movimento.

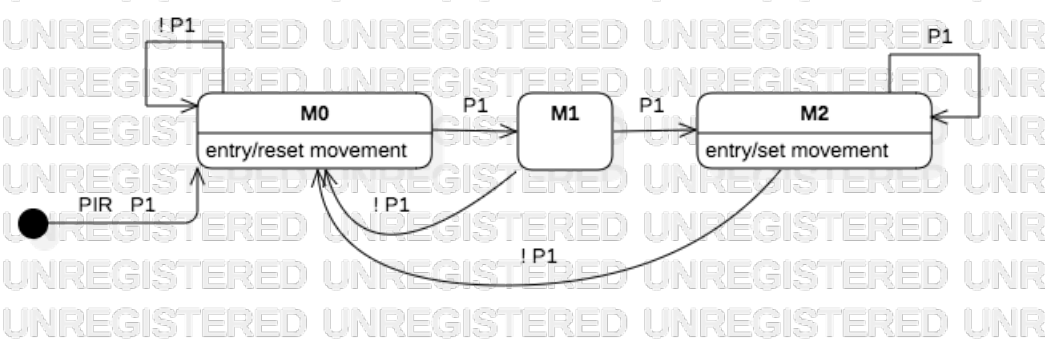

Figura 1: Diagramma a stati Movement Task

- **DistanceTask**: Il DistanceTask incapsula la logica per rilevare la distanza che intercorre tra la macchina e la persona. Si avvale di una classe Sonar che astrae il funzionamento del sensore per la rilevazione della distanza(SONAR). Come per il MovementTask anche esso ha bisogno di un periodo per gestire le rilevazioni sulla distanza, ad esso come al MovementTask è stato assegnato un periodo di 50ms. Il task viene eseguito quando il task principale cioè il **Main Task** si trova nello stato di **ON1** e **ON2** subito dopo che alla variabile movement è stata assegnata il valore true. Ad ogni tick del periodo il task prende la distanza dal SONAR e la assegna alla variabile globale distance. Sia la variabile distance che movement vengono utilizzate per svegliare o addormentare il comportamento dei task. Per gestire il rilevamento corretto da parte del SONAR si è deciso di utilizzare una media pesata dei valori presi in INPUT, cioè i valori precedentemente rilevati avranno una priorità pari ad α invece il valore rilevato correntemente avrà priorità 1-α così da gestire gli sbalzi che il sensore ha in certi rilevamenti.

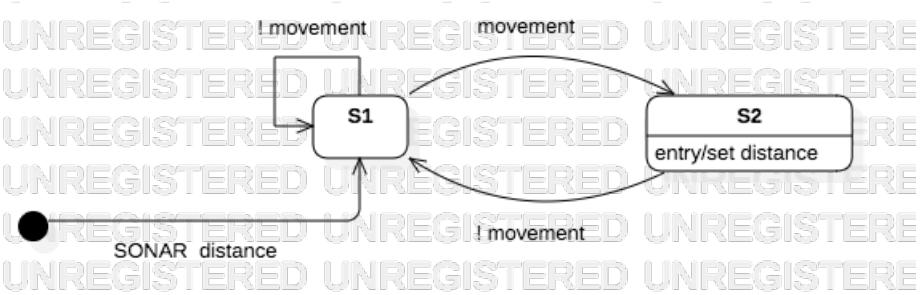

Figura 2: Diagramma a stati Distance Task

- **Maintenance Task**:

  Questo task gestisce lo scambio dei messaggi con la piattaforma realizzata su java attraverso la comunicazione seriale.

  Tale task viene eseguito quando il sistema realizzato entra nello stato di MAINTENANCE. Tale variazione di stato viene espressa attraverso il set della variabile condivisa maintenance. A questo punto il task controlla si mette in ascolto di messaggi in arrivo sulla seriale, in quanto attende il messaggio di ricarica che verrà inviato lato java, per effettuare la ricarica del caffè.

  Tale messaggio contiene un stringa indicante il numero di caffè con cui ricaricare la macchina. Questo dato viene convertito in un intero e sommato alla variabile condivisa che tiene traccia dei caffè da poter fare.

  Il messaggio ricevuto viene poi cancellato, in quanto Arduino non contiene il garbage collector, portando alla saturazione di memoria dopo pochi cicli di esecuzione. E la variabile maintenance viene rosetta e si esce dallo stato di MAINTENANCE.

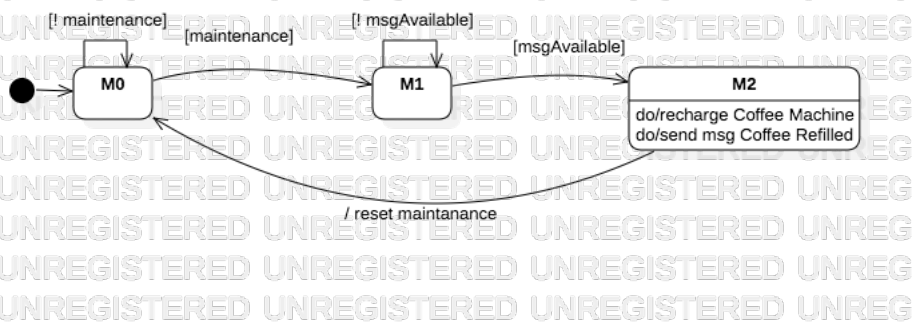

Figura 3: Diagramma a stati Maintenance Task

- **MakeCoffee Task**: Questo task gestisce la produzione di un caffè. Nel nostro sistema è emulata attraverso il lampeggiamento di tre led in sequenza, per la durata totale di 3 secondi. Il tempo di accensione di ogni led viene tenuto memorizzato attraverso una variabile , che viene incrementata del periodo del task, ad ogni chiamata dello scheduler. In questo modo si riesce ad avere un periodo di esecuzione del task più basso, per essere responsive, e una gestione dei led indipendente da esso. Tale task entra in esecuzione quando viene impostata a true la variabile booleana condivisa makeCoffee, e viene impostata a false dal task stesso. In questo modo il task viene mandato in esecuzione da main task e dopo essere terminato, permette di capire al main task il cambiamento di stato. Nella FSM di questo task si vede come ad ogni cambio di stato, venga acceso un led differente e spenti tutti gli altri due.

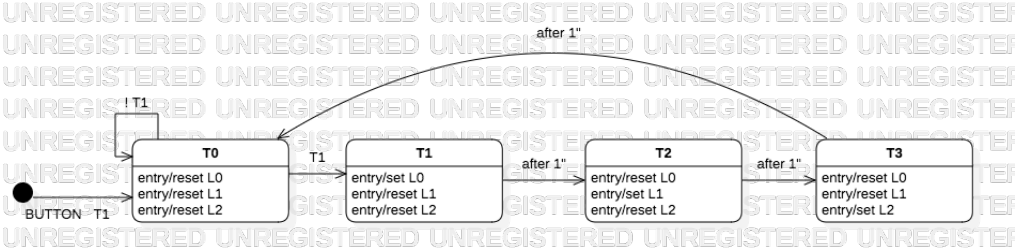

Figura 4: Diagramma a stati Make Coffe Task

- **Main Task**: Il Task principale di questo elaborato in grado di gestire la giusta sequenza di attivazione dei task è il Main Task, al quale è stato assegnato un periodo di 150. Lo stato da cui il main task dovrà partire è quello di **STANDBY**, nel quale si attende la verifica del rilevamento di movimento data dal sensore PIR nel suo task di riferimento, dal quale una volta accertata, si potrà passare allo stato di **ON**, nel frattempo il sistema dovrà entrare in uno stato di sleep dal quale risvegliarsi periodicamente ed effettuare il controllo di variabile. Abbiamo deciso in fase di progettazione di dividere il funzionamento dello stato di **ON** in due stati: **ON1** e **ON2**, in quanto grazie a questa divisione diventa più facile gestire il lavoro a lui assegnato, in particolare ciò che deve fare il primo (**ON1**) è verificare che la variabile di movimento del Pir sia false (quindi ci sia assenza di movimento), nel qual caso bisognerà far partire un contatore e una volta raggiunto il tempo previsto faccia tornare la macchina allo stato di **StandBy** e inviare un messaggio di notifica sulla seriale a java, altro compito di questo stato è attendere che la distanza rilevata dal sonar diventi minore da DIST1 (quella richiesta), in tal caso il sistema dovrà eseguire una transizione di stato da **ON1** a **ON2** inizializzando un nuovo contatore a 0. Entrati in **ON2** bisogna attendere che la variabile appena inizializzata raggiunga il tempo previsto DT1, se invece si ha un ritorno al valore della distanza rilevata dal sonar maggiore di DIST1, bisognerà tornare immediatamente a **ON1**, se invece viene raggiunto il valore di DT1 si effettua un nuovo cambio si stato verso **READY** e si invia un messaggio

di notifica a java. **READY**, come **ON**, si divide in 2 stati **READY1** e **READY2**. In **READY1** avremo la persistenza nell'attuale stato nel caso che la distanza rilevata dal sonar sia ancora una volta inferiore a DIST1 e si avrà la possibilità di impostare tramite il potenziometro il valore dello "zucchero" e inviarlo a java, nel caso che invece venga premuto il bottone T1 si dovrà inviare la notifica a java ed effettuare un cambio di stato a **MAKECOFFEE** che in questo task avrà soltanto il compito di impostare la variabile makecoffe uguale a true, cosi facendo si da la possibilità al Task **MAKECOFFEE** di essere eseguito una volta richiamato dallo Scheduler, se invece la distanza rilevata diventa maggiore di DIST1 si inizializza un nuovo contatore e si passa allo stato di **READY2**, il quale avrà il compito di aspettare che il contatore arrivi al tempo previsto DT2 e in tal caso si avrà un ritorno allo stato di **ON1**, mentre se invece la distanza torna ad essere inferiore a DIST1 un ritorno a **READY1**. **MAKECOFFEE** ha il compito di inviare tramite seriale un messaggio a java per far sapere che ci si trova nello stato attuale, incrementare la variabile numCoffee, segnalatrice del numero di caffè già generati, e nel caso siano finite "le ricariche" (quindi si raggiunga un certo valore) si abbia un cambio di stato a **MAINTAINCE**, l'invio di un messaggio a java del cambio effettuato e l'assegnamento della variabile maintenance a true in modo da poter eseguire il suo task associato, comunque una volta che il task di **MAKECOFFEE** viene eseguito (il quale dovrà rispettare un determinato periodo) si torna allo stato **READY1**. Nello stato **MAINTAINCE** si dovrà tornare allo stato di **STANDBY** una volta eseguito il suo rispettivo task.

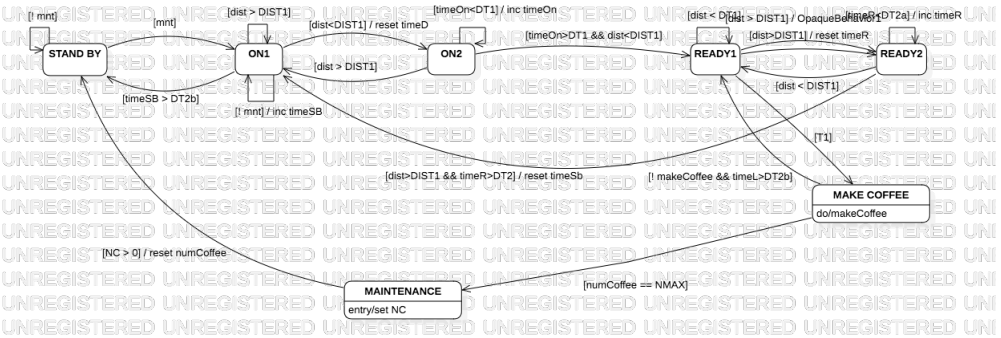

Figura 5: Diagramma a stati Main Task