Progetto Real Time Systems
Line follower

Stefano Maugeri

Relatório de Aprendizagens

Sommario—Il goal di questo progetto è stato quello di programmare un rover affinchè fosse capace di inseguire un determinato percorso e, se richiesto, di essere guidato manualmente. È stata inoltre richiesta una interfaccia grafica per controllare e monitorare il rover da un computer.
Il rover è stato fornito dalla Scuola Superiore Sant'Anna, il software è invece stato scritto per intero dallo studente.

Palavras Chave—Comunicazione seriale, thread e loro sincronizzazione, allegro v.4 su sistemi Unix.

1 Introduzione

Lo scopo principale del progetto è stato il sincronismo in una applicazione multithread. Nello sviluppo di questa applicazione si è scelto di usare l'approccio *Bottom Up*. Utilizzando questo approccio è stato infatti possibile da subito testare il codice sul rover. Il primo lavoro svolto è stato quindi l'interfacciamento motori e, a seguire, dei sensori.
Dopo i test positivi sul firmware relativo al controllo motori si è passato all'implementazione del protocollo di comunicazione seriale tra il rover ed il computer. Questo è stato necessario per soddisfare la richiesta di monitoraggio e controllo tramite Computer. Chiaramente da questo protocollo dipende la buona riuscita dell'intera applicazione, i dati scambiati devono essere affidabili.
Fatto questo si è passato all'implementazione dell'applicazione con GUI sul pc. Questa è stata la fase più delicata, in cui si è prestata particolare attenzione alle linee guida di programmazione date durante il corso e alla progettazione generale del software.
L'intero codice è stato scritto nell'ottica di essere modulare, è stato quindi diviso in più file. Questo ha reso possibile saltuarie modifiche e/o migliorie fatte anche quando si era nell'ultimo stadio dello sviluppo. Dividiamo in due la trattazione del codice scritto per Arduino. Nella prima parte verrà trattato il codice scritto per permettere la locomozione al rover e la lettura dei suoi sensori, nella seconda parte invece il firmware completo scritto per Aduino.

• *Stefano Maugeri, mat. 591511 ,*
E-mail: stefanomaugeri@hotmail.it, Ingegneria Robotica e dell'Automazione, Università di Pisa.

Data: 29 Gennaio 2019.

2 Arduino

2.1 Driver Motori

Una libreria era fornita dal produttore al seguente *sito*, però tale software si è dimostrato essere mal progettato e con grossolani errori che pregiudicavano il corretto funzionamento. È stato così interamente riscritto.
Nel file **DriverMotors.h** situato nella cartella **firmwareArduino** troviamo tutte le funzioni relative alla locomozione e alla lettura dei sensori. Qui si è dovuto trasgredire ad una regola, infatti questo file contiene delle implementazioni di funzioni nonostante l'estensione del file sia *.h*, questa è stata però una scelta obbligata dal compilatore di Arduino.
Nella stesura del programma si è voluto partire dal controllo di ogni motore, per poi creare delle funzioni ad alto livello che permetteranno la locomozione del rover in modo autonomo.

2.1.1 Variabili

```
int speedmotorLeft;
int speedmotorRight;
int directionMotorLeft;
int directionMotorRight;
int initValueMotorLeft;
int initValueMotorRight;
```

Iniziamo parlando delle variabili. L'idea su cui è stato strutturato il firmaware è: il rover deve muoversi senza bisogno di un thread dedicato che lo controlli. Tecnicamente si sono volute creare funzioni non bloccanti, che dicessero cosa dover fare al rover e ritornassero subito il controllo al chiamante. Si è però dovuto cercare un modo per tenere traccia dei comandi dati, e per questo sono state usate le sei variabili sopra riportate. Si sono così create funzioni che modificano la velocità e funzioni che modificano la direzione. Nelle variabili *speedmotorLeft, speedmotorRight, directionMotorLeft, directionMotorRight* salviamo proprio queste informazioni, motore per motore.
Ricordandoci poi che stiamo usando un sistema fisico reale sorge un problema: con che valore minimo il rover inizia a muoversi? Empiricamente è stato trovato un valore, e questo valore viene memorizzato nelle restanti variabili *initValueMotorLeft* e *initValueMotorRight*. Il motivo per cui salviamo questi valori è dovuto al fatto che useremo questi valori come estremi inferiori quando modificheremo la velocità. In questo modo limitiamo il range in cui è possibile cambiare la velocità e, se venisse richiesto di decrementare la velocità sotto questo valore, non sarà possibile farlo. Tecnicamente sono valori fissi e potevamo usare delle costanti, ma vedremo che questi valori sono settati da una specifica funzione di inizializzazione, in modo da essere modificabili se lo si dovesse ritenere opportuno in futuro, e il rover partirà proprio da questa velocità e sotto questa non scenderà mai.

2.1.2 Costanti

Le costanti che troviamo sono utilizzate per maggiore chiarezza e leggibilità del codice.

2.1.3 Funzioni sensori

```
int readSn();
```

Partiamo dalla funzione che legge i sensori. Questa semplicemente ritorna il valore dal pin a cui è collegato il sensore. Nel prototipo sopra riportato n indica un numero $\in [1, 5]$, abbiamo infatti 5 sensori e l'implementazione di ogni funzione è la stessa. La logica della funzione è: se il sensore vede scuro ritorna **LOW**. Altrimenti **HIGH**.
È così possibile far seguire un percorso di colore nero al rover.

2.1.4 Funzioni controllo motori

```
void rightMotor(int);
void leftMotor(int);
```

Queste due funzioni sono quelle che permetteranno alle ruote di girare, ogni funzione controllerà uno specifico motore. Le implementazioni delle due funzioni sono analoghe. I pin su cui sono chiamate le funzioni *analogWrite()* e *digitalWrite()* ci sono stati detti dal costruttore dello shieldbot. Il ponte H che governa i motori è infatti li integrato. Quello che ci aspetteremo da queste funzioni quando le chiameremo è che, modificando i valori delle tensioni ai pin dell'arduino, il motore giri in un ben preciso senso e ad una ben precisa velocità. Nell'implementazione della funzione dobbiamo inoltre stare attenti a passare un corretto valore alla funzione *analogWrite*. Sappiamo infatti che il secondo parametro deve essere un numero $\in [0, 255]$. Nel secondo argomento è stata inserita quindi una espressione che ci assicura ciò, senza utilizzo di variabili di appoggio o verbosi *if () - else*.
Adesso possiamo far spostare il rover avanti e indietro e a destra e a sinistra muovendo alternativamente i motori. Per semplicità è stata ora scritta una funzione di interfacciamento a queste due funzioni, ovvero la funzione:

```
void drive(int left, int right);
```

Lo scopo di questa funzione è quello di chiamare entrambe le funzioni motori e di aggiornare le variabili che riguardano la direzione. Scrivendo, ad esempio:

```
void drive(FORWARD, STOP);
```

questa funzione salverà nelle opportune variabili che il motore sinistro sarà acceso e che la sua direzione di rotazione sarà in avanti mentre il motore destro sarà spento, successivamente chiamerà le due funzioni sopra commentate per effettuare lo spostamento a sinistra. Il rover, in questo caso, andrà a destra quindi. Per maggiore semplicità sono state create delle funzioni che usano questa funzione in modo autonomo. Abbiamo infatti le funzioni:

```
void goForward();
void goBackwards();
void goLeft();
void goRight();
```

si preoccupano loro di passare le corrette costanti alla funzione *drive()* esulandone dall'uso l'utilizzatore di questo firmware. Queste funzioni si preoccupano inoltre di assicurarsi che il rover si muova controllando la velocità inserita dall'utilizzatore e correggendo eventuali errori (esempio un valore troppo alto).
Prima però di usare queste funzioni è essenziale utilizzare la funzione che setta i pin di arduino e che inizializza tutte le variabili. La funzione:

```
void initialize(
    int min_speed_left,
    int min_speed_right);
```

fa proprio questo. Inoltre setta la velocità al suo estremo inferiore, velocità sotto la quale abbiamo detto che il rover non riesce a muoversi a causa dell'inerzia.

2.1.5 Funzioni modifica velocità

Sono state scritte sei funzioni per modificare la velocità:

```
void increaseSpeed(int powerOffset);
void decreaseSpeed(int);
void increaseSpeedLeft(int);
void increaseSpeedRight(int);
void decreaseSpeedLeft(int);
void decreaseSpeedRight(int);
```

È possibile quindi modificare la velocità di ogni motore singolarmente o in modo generale. Anche in questo caso sono state scritte prime le funzioni relative ad ogni motore e poi le due funzioni per un uso più semplice e diretto.

Le funzioni si assicurano che il parametro passato non sia superiore al numero 255 o inferiore allo 0, questo perchè tali valori sono gli estremi ammessi dalla funzione *analogWrite*, comunque questo controllo risulta essere ridondante, infatti le funzioni *left/right motor()* limitano già tali valori. Inoltre le funzioni tengono conto di una eventuale saturazione. Se vogliamo aumentare la velocità, ma siamo già arrivati al massimo, o analogamente vogliamo diminuire la velocità, ma siamo già al valore minimo consentito, imposteranno i valori delle variabili ai loro valori massimi o minimi. Soltanto le due funzioni generali *increaseSpeed()* e *decreaseSpeed()* si occuperanno di chiamare la funzione *drive()* per far accelerare o decelerare il rover nelle variazioni di velocità.

2.2 Firmware multitask

2.2.1 Generale

Il firmware è stato scritto per intero in un file in quanto riguarda poche funzioni. In questo file troviamo i tre thread che gestisce il SO ARTe. Lo scopo di questo scketch è di gestire la comunicazione seriale, leggere i sensori e far muovere il rover di conseguenza. Quello che si vuole è inoltre il poter comandare, sempre tramite seriale, il rover.
Sono state quindi implementate tre modalità di funzionamento:

- **Modalità automatica.** In questa modalità il rover seguirà un percorso nero sotto di lui. Nessuna interazione con l'utente è quindi richiesta. Scriverà inoltre sulla seriale il comando che eseguirà al fine di inseguire il percorso. Chiameremo questa modalità AUTOMATIC.
- **Modalità "stoppato".** Questa è una modalità particolare alla quale è possibile accedere solo se precedentemente il rover si trovava in **modalità automatica**. In questa modalità il rover interrompe l'inseguimento del percorso e aspetterà il comando di RESTART via seriale. Comunque il rover avviserà di essere in questa modalità scrivendo un particolare comando sulla seriale. Chiameremo questa modalità STOPPED.

- **Modalità manuale.** In questa modalità il rover ignorerà i sensori che non verranno più letti e aspetterà di ricevere i comandi dalla seriale. Anche in questo caso, una volta arrivato il comando, il rover scriverà nella seriale il comando che sta eseguendo. Chiameremo questa modalità MANUAL.

2.2.2 Variabili globali

Le variabili globali che usa sono:

```
static int S1, S2, S3, S4, S5;
static char mode;
static char actualCommand;
```

Nelle prime 5 variabili S1, S2, ..., S5 salviamo lo stato dei sensori. Nella variabile *mode* salviamo l'attuale modalità. Nella variabile *actualCommand* l'ultimo comando arrivato. Quest'ultima variabile è usata solo quando siamo in modalità manuale. Salviamo li l'attuale comando ricevuto dalla seriale e un task si occuperà di rimandarlo nella seriale ciclicamente. Questo serve perchè, nell'applicazione finale, vogliamo sapere sempre cosa stanno facendo i motori.
Tutte questa variabili sono protette dagli accessi concorrenziali tra i thread.

2.2.3 Tasks

Su arduino abbiamo in esecuzione 3 thread. La sintassi richiesta da ARTe richiederebbe che ogni task fosse definito in questo modo:

```
void loop_i(int period) {
    //action
}
```

Per maggiore chiarezza del codice sono state definite le macro:

```
#define SENSORS         loop1
#define MOTORS          loop2
#define COMMUNICATION   loop3
```

Così le funzioni verranno dichiarate in questo modo:

```
void SENSORS(50){
    //action
}
```

In questo modo per ogni thread leggiamo il nome per il quale è stato destinato ed il codice risulterà essere più chiaro. Nelle implementazioni di questi 3 task troviamo inoltre la gestione della concorrenza delle variabili globali. Quello che faranno i task è sotto riportato.

- **SENSORS.** Questo task si occuperà di leggere i sensori e di immagazzinarne i valori nelle variabili globali sopra definite se, e solo se, il rover si troverà in modalità AUTOMATIC. Ha un periodo di 50ms, e sarà quindi lui il task a priorità più alta.
- **MOTORS.** Questo task si occuperà del controllo motori, differenziando il proprio compito in base alla modalità in cui si trova il rover. È stato scelto per lui un periodo di $100ms$.
  - Modalità STOPPED: bloccherà le ruote e invierà tramite seriale, ogni $100ms$, il comando CMD_STOP.
  - Modalità AUTOMATIC: chiamerà la funzione *follow_line()* che comanderà i motori sulla base dei dati letti dal task SENSORS per permettere al rover di inseguire un percorso. Inoltre questa funzione scriverà sulla seriale il comando sui motori che sta effettuando.
  - Modalità MANUAL: invierà l'attuale comando, che è stato precedentemente ricevuto tramite seriale, sempre sulla seriale. Oltre a svolgere la funzione di ack, vedremo che questa informazione sarà usata da un thread dell'applicazione.
- **COMMUNICATION.** Questo task si occuperà della ricezione dei dati (in modo asincrono) proveniente dalla seriale e della loro gestione. Le scelte che farà dipenderanno dall'attuale modalità in cui si trova. È grazie a questo task che sarà possibile comandare il rover via seriale, o modificarne la modalità. Il periodo di questo task ' di $200ms$.

Il motivo per cui non sarà solo il task COMMUNICATION ad utilizzare la seriale è dovuto al fatto che vorremmo sempre mandare informazioni all'applicazione che è stata scritta sul computer, ma il task COMMUNICATION resterà sempre in attesa di leggere nuovi dati, e la sua esecuzione dipende quindi da quando

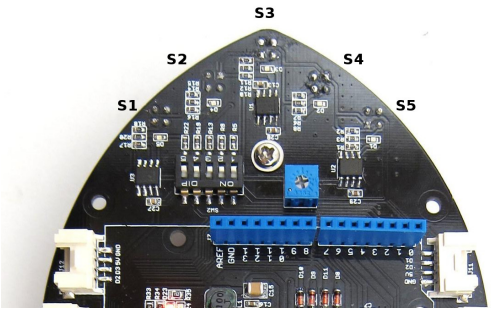


Figura 1. Posizionamento sensori rover

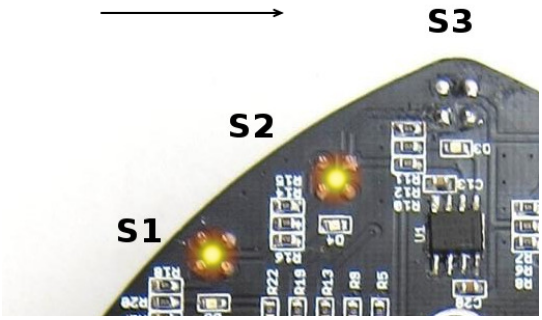


Figura 2. Esempio curva verso destra

riceverà un comando.
Per questo motivo sarà il task MOTORS, quando sarà in modalità MANUAL, ad inviare via seriale l'ultimo comando ricevuto e che starà eseguendo tramite la funzione *sendAction()*. Quando invece si troverà in modalità STOPPED oltre a tenere fermi i motori invierà sulla seriale il comando CMD_STOP.

2.2.4 Funzioni

Le funzioni usate dai task sono quattro.

```
char getCommand();
```

Questa funzione bloccherà l'esecuzione fino a quando non riceverà un carattere salla seriale, che ritornerà al chiamante.

```
void follow_line();
```

Questa funzione muoverà il rover cercando di spostarlo in modo tale che sia solo il sensore centrale ad identificare la linea nera. Per vedere dove sono posizionati i sensori osserviamo la Figura 1. Quello che farà la funzione sarà di spostare il rover verso destra se si accenderanno i sensori **S1** e/o **S2** oppure di spostarlo verso sinistra se si accenderanno i sensori **S4** e/o **S5**. Vediamo un esempio nella Figura 2.

Una volta che i sensori laterali si saranno spenti e si sarà acceso il sensore **S3** allora il rover andrà avanti fino ad una nuova interruzione. Se durante una curva particolarmente complicata il rover dovesse perdere il percorso sotto di lui perchè lo avrà superato, nel rover si spegneranno tutti i sensori ed il rover si sposterà all'indietro fino a quando non avrà ritrovato il percorso. Questo però implica che il rover, quando non troverà un percorso, lo cercherà sempre andando all'indietro. L'ultimo **else** che troviamo nella funzione non è verificabile, infatti prima sono gestiti tutte le condizioni, e l'azione che farebbe al non verificarsi delle condizioni precedenti (che darebbero il controllo alle istruzioni di questo **else**) sarebbe molto forte, infatti bloccherebbe il rover inesorabilmente. Cosa mai successa durante i test effettuati. È stata comunque inserita solo per completezza.

```
char driveRover(char cmd);
```

Questa funzione si occuperà di muovere il rover sulla base dei comandi che arriveranno dalla seriale e che a lei saranno passati come semplice parametro. Riguardo la concorrenza delle variabili globali in questa funzione non servirà gestirla, in quanto la variabile che leggerà non sarà globale ma locale. Si preoccuperà il task COMMUNICATION di gestire la concorrenza. Questa funzione ritornerà infatti un carattere per permettere al task COMMUNICATION la gestione della concorrenza. Nell'implementazione vediamo che i comandi CMD_MOTORUP e CMD_MOTORDOWN non saranno ritornati, questo perchè nell'applicazione che useremo nel computer non servirà avere questa informazione.

```
void sendAction(char cmd);
```

Scrive sulla seriale la variabile *actualCommand*

lasciando il compito della gestione della concorrenza al task chiamante. Anche in questo caso la condizione di *default* non sarà verificabile, è stata inserita per completezza.

3 SOFTWARE PC

La GUI realizzata, abbastanza minimale, è riportata nella Figura 3. Nella stessa figura troviamo tre tonalità di colore, che denotano i tre thread che la controllano. Nella figura il rover si trova nella modalità manuale.

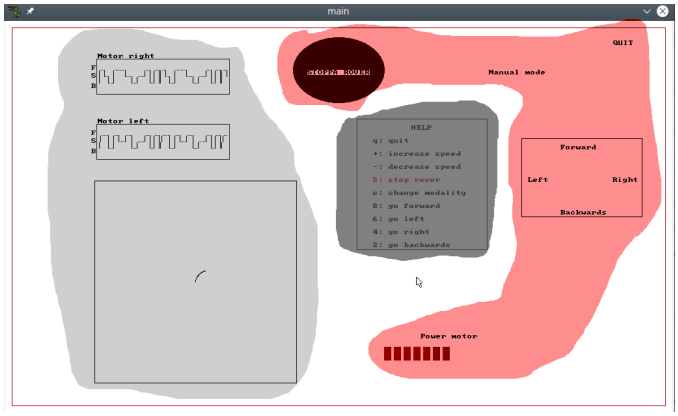


Figura 3. GUI e suoi task di gestione

- Rosso: Lo chiamiamo τ_1, si occuperà della gestione dei bottoni e della visualizzazione (ma non gestione) dell'indicatore di velocità.
- Grigio scuro: Lo chiamiamo τ_2, si occuperà della lettura e gestione dei comandi dalla tastiera.
- Grigio chiaro: Lo chiamiamo τ_3, si occuperà della lettura e estrapolazione dei dati dalla seriale, grafico dei motori e disegno della traiettoria.

Questi thread verranno memorizzati in uno specifico array dichiarato nel file *firmwarePc/mainFunction/headerFile.h*.
Tale array è:

```
pthread_t tid[3];
int n_tid;
```

Inoltre, anche se sappiamo che i thread saranno tre, teniamo il conto dei thread su una apposita variabile per maggiore ordine. Queste semplici variabili ci faciliteranno il compito di arresto dei thread.

Il main, una volta lanciata l'applicazione, resta in attesa solo del thread che in figura abbiamo chiamato τ_1 e che sarà salvato su *tid[0]*, per questo è considerato il thread principale. Nonostante sia il thread principale vediamo dal codice che gli è stata assegnata la priorità più bassa, tra i thread. Questo perchè il thread resterà semplicemente in attesa di una interazione con l'utente.
Osserviamo adesso una possibile criticità nel codice, ovvero il seguente codice:

```
while (run) {
    createScreen();
    pthread_join(tid[0], NULL);
}
```

dove la *createScreen()*, dopo aver creato graficamente la schermata, lancia i thread. Se il thread τ_1 non dovesse partire immediatamente, quando verrebbe valutata la *pthread_join(tid[0], ...)* su *tid[0]* potrebbe non essere presente nessun valore, e questo porterebbe sicuramente a comportamenti non aspettati o a violazioni della memoria con conseguente arresto del programma. Per evitare questo si è ricorso alla tecnica del *busy waiting*.
Infatti la funzione *primary_thread()*, chiamata dalla funzione *createScreen()* utilizza la variabile contenuta nella struttura, dichiarata globalmente, *task_par* settandola a *FALSE*. Adesso la funzione *primary_thread()*, tramite la seguente istruzione, si metterà in attesa che il thread che ha fatto partire venga effettivamente eseguito e mandato in esecuzione:

```
while (!task_par.arg) ;
```

Il thread, quando gli verrà assegnata la CPU, si occuperà di settare a *TRUE* la variabile situata nella struttura condivisa e controllata dalla funzione *primary_thread()*. E da quel momento, la funzione precedentemente bloccata potrà continuare la sua esecuzione e, quando torneremo nel *main*, saremo sicuri che dentro *tid[0]* ci sia un valore consistente, e che quindi la funzione *pthread_join(tid[0], ...)* si comporterà come ci aspettiamo che faccia, ovvero bloccando l'esecuzione del main a questa linea. Intanto saranno stati lanciati gli altri thread.
Le routine dei thread sono spiegate in una

sezione a loro dedicata, l'uso della variabile *run* pure.

3.1 Organizzazione software

Il codice è diviso in più file salvati nelle relative cartelle. Possiamo vederlo per intero nella Figura 3, dentro la directory **firmwarePc**.

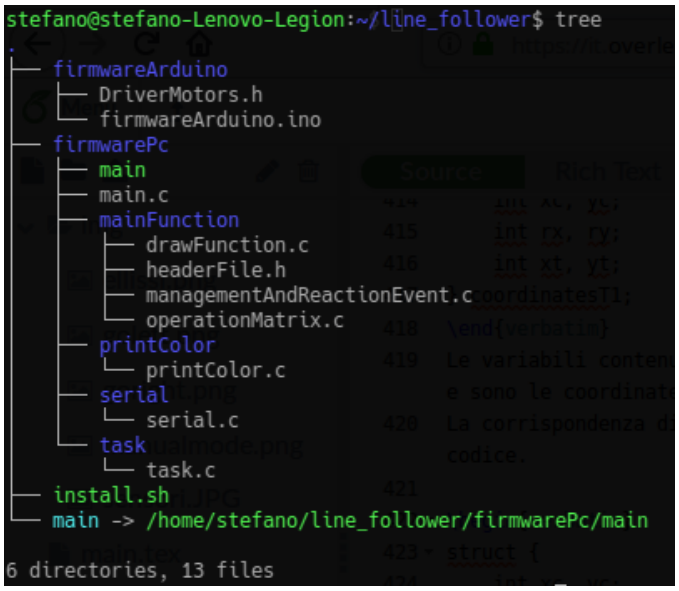


Figura 4. Organizzazione software

3.2 serial/serial.c

In questo file sono state raccolte le funzioni relative alla comunicazione seriale. Due di queste funzioni sono state scritto con il supporto di una discussione trovata sul noto sito StackOverflow, ovvero le funzioni *set_interface_attribs(...)* e la funzione *set_blocking(...)*. Le altre sono invece state scritte per intero dallo studente.

```
int init_serial(char *portname);
```

Questa funzione si occupa di inizializzare la comunicazione seriale con le impostazioni di default, in quanto è stata scritta ad hoc per l'applicazione. Il parametro che richiede è il nome della porta a cui è stato collegato il cavo usb. È quindi sempre un parametro del tipo **/dev/tttyACMx** dove x è un numero. Se si passa un parametro sbagliato la funzione fallisce, scriverà un messaggio di errore e ritornerà il numero -1 al chiamante. Vedremo che il *main* farà un controllo su questo numero per assicurarsi che l'inizializzazione abbia avuto successo. Le impostazioni di default che useremo in questa applicazione saranno:

boudrate: 9600 bps.
bit di parità: disattivato.

Inoltre in questa funzione, prima della compilazione, se viene trovata la macro ALLEGRO_H (definita nella libreria */usr/include/allegro.h*) in caso di errore userà la funzione *allegro_message(...)* per comunicare l'errore, altrimenti si limiterà ad effettuare una stampa sull'output. Un altro motivo per cui questa funzione può fallire è se la porta non gode dei privilegi di amministratore.
Se invece tutto andasse bene verrà ritornato il descrittore alla porta, e sarà quindi possibile utilizzare la comunicazione seriale.

```
void printFromSerial(int fd);
```

Questa funzione legge i dati in arrivo sulla seriale e li stampa a video fino a quando non riceve il carattere $'\backslash n'$.

```
char getCommandFromSerial(int fd);
```

Questa funzione ritorna l'ultimo carattere letto dalla seriale. Nel farlo sappiamo che Arduino invierà un tipo di dato statico, mandando sempre prima un carattere e poi il carattere $'\backslash n'$.

3.3 printColor/printColor.c

Questo file contiene solo una funzione per modificare il colore delle stampe dell'output. Questo ha reso più veloce il debug.

3.4 task/task.c

Questo file contiene invece le funzioni trattate durante il corso, con qualche aggiunta poi neanche utilizzate.
Queste funzioni, e le relativi variabili con cui operano, sarebbero servite ad implementare un cronometro. Queste funzioni sono poi state abbandonate lasciando ad un task di arduino il compito di scandire il tempo.

3.5 mainFunction

In questa directory troviamo invece più file contenenti diverse funzioni. Le funzioni sono state raggruppate secondo il loro scopo.

3.5.1 headerFile.h

In questo file troviamo tutte le costanti, variabili globali principali e strutture dati utilizzate. Le costanti sono, già nel file, divise in base all'utilizzo a cui sono destinate.
Abbiamo poi le variabili globali principali, ovvero:

```
int fd;
```

Questa variabile è semplicemente il descrittore della seriale, ritornato dalla funzione *init_serial(.)*.

```
int run;
```

Questa variabile può determinare il riavvio dell'applicazione o l'arresto. Avendo due modalità e volendo avere due schermate diverse in base all'attuale modalità si è dovuto pensare un modo per intercambiare le due schermate. E questo senza dimenticarsi del fatto che la GUI è una vera e propria risorsa condivisa su cui operano più task. Si è quindi pensato di stoppare tutti i thread, cancellare la schermata, ricrearla e far ripartire i task.
L'implementazione di ciò è stato fatto anche grazie a questa variabile. Il *main()* contiene infatti un ciclo *while(.)* che ha per condizione proprio questa variabile e per istruzioni il riavvio dei task, la cancellazione e creazione della nuova schermata. Mettere a *FALSE* questa variabile significa quindi il voler arrestare il programma. Vedremo che saranno più task a poter far ciò.

```
int mode;
```

Su questa variabile salviamo semplicemente l'informazione riguardo l'attuale modalità in cui si trova il rover. Per far questo è prevista un handshake tra il rover ed il pc. Quando il firmware su Arduino inizia la sua esecuzione aspetterà sempre di ricevere il comando riguardante la modalità. Questo comando è quindi inviato dal *main*.

```
int closeSw;
```

Questa variabile mette in comunicazione τ_1 con τ_2. Infatti τ_2 potrà assegnare due valori a questa variabile. Se τ_2 settasse questa variabile al valore della costante SHUTDOWN lo farebbe perchè l'utente vorrà arrestare il programma, allora τ_1 invierà il comando CMD_STOP al rover, setterà la variabile *run* a *FALSE* e terminerà la sua esecuzione. Da questo momento la *pthread_join(tid[0], ...)* che abbiamo nel *main()* farà proseguire l'esecuzione, quindi verrà verificata la condizione su *run* che risulterà essere falsa. Quindi il ciclo *while(run)* non verrà rieseguito e il *main()* proseguirà con il proprio codice, che consisterà nello stoppare i due thread rimasti in esecuzione, chiudere la comunicazione seriale e arrestare il programma.
Se invece τ_2 dovesse settare la variabile al valore della costante REBOOT sarà perchè l'utente vorrà cambiare modalità e quindi il software dovrà essere riavviato per aggiornare la schermata. Quindi τ_2 invierà il comando CHANGE-MODALITY e si arresterà senza modificare la variabile *run*, stavolta *while (run)* verrà rieseguito e il software si riavvierà. Chiaramente i due precedenti thread verranno prima stoppati.

```
int motorPower;
int oldMotorPower;
```

Queste due variabili si riferiscono ad una stessa informazione, e sono variabili di supporto per la gestione dell'indicatore di velocità. La seconda è utilizzata solo da τ_1, perciò possiamo non usare protezione per lei. Nella prima variabile, su *motorPower*, salveremo invece il nuovo valore che verrà modificato dall'utente. Perciò *motorPower* verrà letta dal thread τ_1 ma modificata da τ_2. Per questa variabile sono quindi stati usati gli opportuni accorgimenti per gestire la mutua esclusione alla risorsa.

```
    struct task_par tp1;
    struct task_par tp2;
    struct task_par tp3;
```

Queste strutture sono strutture utilizzate per la creazione dei task e sono quindi usate rispettivamente da un unico task.

```
struct {
    int xq, yq;
    int xc, yc;
    int rx, ry;
    int xt, yt;
} coordinatesT1;
```

Le variabili contenute in questa struttura sono scritte dal *main()* e vengono lette solo da τ_1, e sono le coordinate degli oggetti grafici che τ_1 monitora. La corrispondenza di queste variabili con gli oggetti visualizzati nella GUI è riportata direttamente nel codice.

```
struct {
    int xc, yc;
    int xf, yf;
    int xr, yr;
    int xl, yl;
    int xb, yb;
} coordinatesT2;
```

Analogamente, queste sono le coordinate che sono scritte sempre dal *main()* e che vengono lette da τ_2.
Per τ_3 la scelta di usare una struttura globale non è stata fatta, questo per semplicità di sintassi all'interno del task stesso. Inoltre le coordinate su cui sui lavora τ_3 vengono modificate in ogni periodo, a differenza degli altri task, e questo invalida l'idea di tenere delle strutture dati su cui memorizzare semplicemente delle coordinate statiche.
Alla fine di questo fie troviamo i prototipi delle funzioni scritte. Nell'intestazione di ogni funzione nei relativi file sono sempre riportate delle spiegazioni sulla singola funzione.

3.5.2 drawFunction.c

In questo file troviamo solo funzioni che operano sulla GUI tramite la libreria **Allegro**. Tutte queste funzioni hanno il solo compito di calcolare coordinate e disegnare oggetti.
Notiamo adesso che sia τ_1 che τ_2 potranno scrivere sulla GUI, e per farlo ci sarà la variabile *screen* condivisa, poi in base all'operazione (disegno testo o figura) potrebbero esserci altre variabili (*font*). Per proteggere questa variabile da accessi multipli usiamo quindi un mutex. In alcune funzioni questa protezione manca, questo perchè quelle funzioni saranno utilizzate quando, sicuramente, saranno le uniche istanze in esecuzione, ovvero alla partenza.
Notiamo inoltre che tutte le coordinate calcolate dentro questo file sono in funzione di due costanti, ovvero WIDTH e HEIGHT, ovvero le dimensioni scelte arbitrariamente per la finestra. In questo modo se si volesse cambiare la dimensione della una finestra basterebbe agire su quelle costanti e il codice cercherà di adattarsi automaticamente. Chiaramente ci saranno dei limiti.
Tra le varie funzioni ci concentriamo sulle più complesse, ovvero:

```
    void drawTrajectory(...);
```

che è una funzione di τ_3. Il rovescio della medaglia per non aver usato una struttura su cui salvare le sue variabili lo paghiamo nel passaggio dei parametri a questa funzione: questi infatti risultano essere molti.
Quando questa funzione è stata chiamata tutte le coordinate relative alla traiettoria sono già state calcolate e a le spetta solo di effettuare il disegno sulla GUI. Quello che farà quindi questa funzione sarà di disegnare un segmento a partire dalle precedenti coordinate calcolate fino alle nuove coordinate. Ecco perchè questa funzione riceve le variabili *xg, yg, xg_old, yg_old*.
Vediamo inoltre che le variabili relative alle vecchie coordinate sono passate come puntatori. Questo perchè queste variabili verranno aggiornate direttamente da questa funzione. La funzione inizierà a disegnare dal centro del riquadro.

Le regole che dovrà seguire questa funzione sono le seguenti:

- Consistenza delle variabili *xg_old, yg_old*. Quando la funzione sarà chiamata per la prima volta dal task queste variabili saranno inizializzate a 0, in quel momento quindi decidiamo di non modificare la GUI, ma ci limiteremo ad aggiornare queste variabili in modo che, alla prossima chiamata, le variabili avranno consistenza e potremo così tracciare la traiettoria sulla GUI.
- Non superamento della cornice. Il riquadro dedicato alla traiettoria è infatti limitato da una cornice oltre la quale non vogliamo che il grafico sia disegnato.

```
drawGraphicsMotor(
int cmd,
int xp,
int yp);
```

Questa funzione permetterà la visualizzazione dei grafici relativi ai motori. I grafici che disegnerà saranno due e saranno simili. I due grafici saranno disegnati sulla base di una struttura globale condivisa solo da τ_3, perciò si limiterà a disegnare sulla GUI le due cornici ed a chiamare le funzioni che si occupano della gestione della struttura dati. Tali funzioni e relative strutture dati verranno trattate successivamente.

3.5.3 managementAndReactionEvent.c

In questo file troviamo le funzioni principali dell'applicazione. In questo file sono concentrate le funzioni che sono utilizzate dai task.

```
void analayzeKey(char key);
```

Questa funzione riceve in ingresso un comando proveniente dalla tastiera che è stato ricevuto direttamente dall'utente ed ha il compito di eseguire tale comando. Chiaramente l'utente potrebbe aver premuto un tasto a cui non è associato nessun comando, in questo caso si è deciso di non effettuare nessuna azione. Si è deciso di rendere tale funzione case unsensitive.

Tra i comandi che può dare l'utente c'è quello relativo all'arresto dell'applicazione, e chiaramente non sarà questa funzione a chiamare brutalmente la funzione *exit()*.
Dovendo in questa funzione gestire due variabili globali condivise, ovvero *closeSw* e *mode*, e per evitare sezioni critiche annidate si è usata una variabile di appoggio su cui salvare il valore di *closeSw*. La modifica di questa variabile è gestita da $\tau 1$ e la reazione a tale modifica dipende dal nuovo valore che gli verrà assegnato come precedentemente spiegato.
A tal proposito introduciamo le due costanti SHUTDOWN e REBOOT. Alla fine di questa funzione, se l'utente avrà dato il comando di arrestare l'applicazione allora la variabile *closeSw* avrà il valore di SHUTDOWN. Se invece l'utente avrà dato il comando di cambiare modalità la variabile *closeSw* assumerà il valore di REBOOT.

```
void managementIndicator();
```

Questa funzione gestisce l'indicatore della velocità. Anche in questo caso, onde evitare sezioni critiche annidate, usiamo una variabile temporanea per la variabile globale *motorPower*. La variabile *oldMotorPower*, anch'essa globale, è utilizzata solo da un task, quindi su di essa non troviamo meccanismi di protezione. Confrontando queste due variabili, sapendo che *motorPower* può essere modificata solo dalla funzione precedentemente descritta, se risulteranno essere uguali sarebbe inutile effettuare una nuova modifica alla GUI. Ecco perchè questa funzione modifica l'indicatore solo quando l'utente richiede una variazione alla velocità, e perciò le due variabili textitoldMotorPower e textitmotorPower saranno diverse.

```
int pressedEllipsis();
```

Nella GUI si è creato un particolare tasto di forma ellittica. Discriminare un click al suo interno non è immediato come nel caso in cui si ha un quadrato. Ricordandoci che la condizione di un punto appartenente ad una ellissi è data dall'equazione dell'ellissi calcolata nel punto

in esame, sappiamo che se si otterrà un valore minore del termine noto dell'ellissi stessa, allora il punto apparterrà all'ellissi, altrimenti no. Il problema informatico si è trasformato in un problema di natura matematica, e sta quindi nel trovare l'equazione dell'ellissi di nostro interesse. Avendo già i punti caratteristici dell'ellissi, ovvero semiassi e centro, usiamo il software **Geogebra** per calcolare facilmente l'equazione. L'implementazione in C è molto semplice, l'unico accorgimento da effettuare è stato il dover considerare ribaltato l'asse Y.

Le funzioni del tipo:

```
int pressed<OBJECT>();
```

hanno un compito simile, ma operano su oggetti rettangolari, quindi la matematica risulta essere ben più semplice.

```
void managementEvent
    (int mx, int my);
```

Tutte le funzioni che inviano i comandi relativi alla locomozione sono chiamate da questa funzione allo scaturirsi di un click sulla GUI.

```
void stopAllthread();
```

Quando serve arrestare tutti i thread che sono in esecuzione verrà chiamata questa funzione. Siccome non siamo sicuri che da quando la funzione *pthread_cancel(.)*, il thread venga effettivamente arrestato, è stata inserita una *pthread_join(.)* per aspettare che il thread si concluda. Notiamo che i thread saranno eliminati dall'ultimo fino al primo, questo per gestire al meglio la concorrenza. Come già visto nel *main()* abbiamo infatti una *pthtread_join()* proprio su *ted[0]*. Inoltre le stampe in questa funzione sono evidenziate in blue, questo ha semplificato il debug.

```
void signalRoutine();
```

Questa è principalmente una funzione di debug, per gestire la terminazione forzata dal programma. Questa è l'unica eccezione in cui troviamo la funzione *exit()* oltre che nel *main()*. Questo perchè, essendo una funzione pensata principalmente per il debug, se viene chiamata è probabile che il *main* possa essere anche in uno stato di blocco a causa di un errore creatosi nella fase di sviluppo.

```
int getClearCommand(char buf);
```

Questa funzione è chiamata da τ_3 dopo aver letto la seriale e serve a discriminare i comandi che si aspetta di ricevere τ_3 da qualunque altro comando possa arrivare dalla seriale. I comandi che sappiamo voler eliminare, dentro τ_3, sono i comandi che manda Arduino per confermare l'avvenuta modifica alla velocità.

```
void getangle(
    double *theta,
    int *oldCommand,
    int command);
```

Anche questa funzione è chiamata da τ_3, e ha il compito di calcolare l'angolo verso cui disegnare il prossimo segmento che sarà la traiettoria. In questa funzione ci preoccuperemo di far si che la variabile *theta* rispetti la condizione $\theta \in [0, 360]$.

```
void calculateCoordinates(
    double theta,
    int *x,
    int *y);
```

Questa funzione calcola le coordinate per il grafico della traiettoria. Nel farlo, per come abbiamo inizializzato le variabili, supponiamo che il rover parta con una direzione ortogonale al monitor del computer e che vada oltre lo schermo, piuttosto che verso l'utente. Dalla variabile *speed* vediamo che nel tracciare la

traiettoria consideriamo la velocità attuale del rover.

3.6 main.c

Nel main troviamo chiamate le funzioni precedentemente spiegate. Vediamo che la condizione necessaria affinchè l'applicazione parta è che l'inizializzazione della comunicazione seriale vada a buon fine. Abbiamo detto che questa funzione può fallire nelle seguenti condizioni:

- Cavo usb non collegato.
- Porta passata come argomento errata, potrebbe infatti non essere sempre */dev/ttyACM0*. Questo dipende dal sistema operativo e solitamente è possibile risalire alla corretta porta usando uno dei seguenti comandi:
  ```
  $ lsblk
  $ dmesg | grep tty
  ```
- Mancanza dei permessi al file descrittore della porta seriale. È possibile in questo caso risolvere il problema con il seguente comando:
  ```
  $ sudo chmod 777 /dev/ttyACM0
  ```
  Supposto chiaramente che la porta */dev/ttyACM0* sia la corretta porta.

Dopo questo, ricordandoci che è previsto un handshake all'avvio tra Arduino e l'applicazione, dobbiamo comunicare ad Arduino la modalità da cui vogliamo iniziare. La modalità con cui si inizia è indifferente.
Fatto questo troviamo il *while(run)* già descritto.

```
void startThread(
    void*(*func)(void*),
    struct task_par *tp,
    int priority);
```

Questa funzione permette di far partire task periodici secondo quanto detto durante il corso. Notiamo la gestione del caso in cui la funzione *pthread_create(.)* fallisca. Tale funzione può infatti fallire se l'applicazione non è stata fatta partire con i privilegi di amministratore. Questo è dovuto al fatto che stiamo usando un diverso scheduler da quello standard.

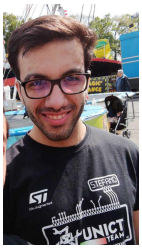
4 UTILIZZO

Il software scritto per Arduino è stato realizzando utilizzando l'editor testuale **Vim**, la compilazione e l'operazione di flash sul processore è stata effettuata tramite l'IDE **Arduino 9 - BETA**, come richiesto da ARTe. Il debug è stato effettuato sempre tramite l'IDE offerto da Arduino.
L'applicazione è stata sviluppata utilizzando sempre l'editor testuale **Vim**, la compilazione è stata effettuata con un software scritto sempre dallo studente che aveva il compito di generare il seguente comando:

```
gcc -Wall -Werror main.c -g
    -lpthread -lm -lrt -lalleg
    -o main
```

Il debug è stato effettuato tramite i software **gdb** e **valgrind**.
Per eseguire il software la prima cosa da fare è la compilazione di tutti i file. Per fare questo si può usare lo script *install.sh* situato nella cartella principale. Questo script può fallire se nel sistema in cui si sta provando la compilazione non è presente la libreria *allegro*. In tal caso lo script suggerirà il comando per installarlo.
A compilazione eseguita nella stessa directory è stato creato il link *main* da cui è possibile eseguire l'applicazione. Quindi per eseguirla bisognerà dare il comando

```
$ sudo ./main
```



Stefano Maugeri Studente del corso di Ingegneria Robotica e dell'Automazione, Pisa. Attuale matricola: 591511. Proveniente dal corso di Ingegneria Informatica, Catania.