Relazione Progetto Sistemi Operativi

Autori

Nome e Cognome	Matricola	E-mail
Simone Felici	7013439	simone.felici1@stud.unifi.it
Giulio Fringuelli	7026006	giulio.fringuelli@stud.unifi.it
Samuele Del Corto	7030872	samuele.delcorto@stud.unifi.it

Data di consegna

24-06-2023

Istruzioni per l'installazione

Per installare:

1. Recarsi all'interno della cartella *Progetto*.
2. Aprire il terminale.
3. Digitare *make -f makeFile.make install*

Istruzioni dettagliate per compilazione ed esecuzione

Per compilare:

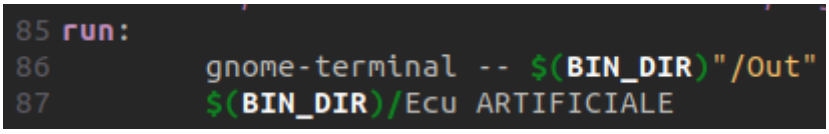
1. Recarsi all'interno della cartella *Progetto*.
2. Aprire il terminale.
3. Digitare *make -f makeFile.make all*.

Per eseguire:

1. Aprire un altro terminale.
2. Nel primo terminale digitare *./bin/Ecu NORMALE* se si vuole eseguire il programma in modalità normale, altrimenti digitare *./bin/Ecu ARTIFICIALE* se lo si vuole eseguire in modalità artificiale.
3. Nel secondo terminale digitare *./bin/Out*.

L'alternativa alla precedente esecuzione è digitare con un solo terminale aperto il comando *make -f makeFile.make run*, questo è possibile solo se si ha installato GNOME .

Per scegliere tra modalità normale o artificiale, modificare il make file alla voce run.



Caratteristiche SW

Distribuzione Linux	Compilatore	Editor di testo
Ubuntu 22.04.2 LTS Jammy	gcc version 11.3.0	Eclipse IDE C/C++ 2023-06
Lubuntu 22.04.2 LTS Jammy	gcc version 11.3.0	Eclipse IDE C/C++ 2023-03

Caratteristiche HW

Modello hardware	Memoria	Processore	Grafica	Capacità disco
Acer SWIFT 3	8.0 GB	Intel Core i5-8265U 1.60GHz x 4	NVIDIA GeForce MX150	SSD NVMe PCIe 250 GB
ASUSTeK COMPUTER INC. X555LJ	12.0 GB	Intel Core i7- 5500U 2.40GHz x 4	Nvidia GeForce 920M	SSD SATA 480 GB
HP Pavilion Laptop	16.0 GB	Intel Core i7-1165G7 2.80GHz x 8	Mesa Intel Xe Graphics (TGL GT2)	SSD NVMe PCIe 1 TB

Progettazione ed implementazione

Il progetto è suddiviso in svariate cartelle contenenti tutti i file utili nel progetto. La scissione è stata effettuata in base al tipo del file, ciò offre una migliore organizzazione del codice quindi una maggiore facilità di manutenzione e comprensione.
Le cartelle sono:

- **bin:** contiene i file binari generati durante la compilazione del programma. Inoltre, bin contiene pure una cartella che prende il nome di fileUtils.
- **fileUtils:** contiene frontCamera.data e urandomARTIFICIALE.binary.
- **include:** contiene tutti i file di intestazione (header).
- **log:** contiene tutti i file di log generati durante l'esecuzione del programma.
- **obj:** contiene tutti i file oggetto generati dal compilatore durante la compilazione del software per ogni file sorgente.
- **src:** contiene tutti i file con estensione sorgente, cioè con estensione (.c).

Come si può dedurre dalla suddetta suddivisione, l'intero progetto è stato partizionato in tanti piccoli moduli in maniera tale da riusare le funzioni ogni qualvolta ce ne sia bisogno includendo il file di intestazione correlato. In aggiunta, tale scissione ci permette di modificare i moduli separatamente senza dover ricompilare il programma completo. Notare che ogni modulo ha associata una responsabilità, quindi risulta immediato trovare il modulo che deve essere aggiornato conoscendo il suo compito.
Si è cercato di inserire più funzioni possibile assicurando modularità, leggibilità e riusabilità. Queste funzioni prendono il nome di funzioni di utilità, sono delle operazioni comuni che possono essere usate all'interno del programma dai diversi componenti.
Si può ulteriormente notare la presenza delle macro, utilizzate principalmente per accedere ad alcune risorse predefinite del sistema, tra cui la data nel nostro caso, e per evitare duplicazione di codice, fornendo quindi un codice ottimizzato.
Entriamo nei dettagli della struttura realizzata.
La comunicazione tra i diversi processi avviene tramite socket AF_UNIX iterative.
La struttura generale del progetto corrisponde ad una struttura gerarchica dove la Central ECU è un server che interagisce con gli altri componenti del sistema che corrispondono ai suoi figli esclusi Out e Surround View Cameras; quest'ultimo è infatti figlio del componente Park Assist.
Per instaurare le connessioni e chiuderle vengono utilizzate delle funzioni di utilità.
La funzione memset presente nei vari componenti permette di azzerare l'intero blocco di memoria utilizzato, impostando tutti i byte a zero.

1. **HMI**

   La HMI è stata rappresentata da due componenti: In e Out.
   L'elemento In è un client del server *ECU* e permette all'utente di inserire un messaggio e inviarlo al server sfruttando le funzioni di utilità.
   Il componente Out, anch'esso client del server *ECU*, riceve dal server determinate informazioni e le stampa a video. Se il messaggio ricevuto corrisponde alla stringa *ARRESTO TOTALE* o alla stringa *Park complete* allora il componente termina la sua esecuzione. Questi due controlli sono effettuati all'interno di una funzione definita col nome di *receiveBreak* che prende come parametro la stringa da controllare, tale stringa è quella ricevuta dal server.
   Infine in entrambi i file si chiude la connessione col server.

2. **Steer By Wire**

   Instaura la connessione con il server *ECU*, successivamente crea il file di log *steer.log* con la funzione di utilità *openLog* che permette appunto di creare o aprire file.
   Prima di entrare nel loop richiama il metodo *createNonBlockReadClient* che permette di impostare un descrittore di file in modalità non bloccante per la lettura. È possibile effettuare tale operazione tramite la system call *fcntl* che permette di prendere e settare lo stato dei flag.
   Riassumendo, le operazioni di lettura eseguite su quel descrittore di file non si bloccano, ma restituiscono immediatamente un valore disponibile o un valore di errore se non ci sono dati disponibili al momento della lettura.
   Entra nel loop dove si mette in ascolto col server e nel caso riceva il messaggio *STO GIRANDO A DESTRA* lo scrive nel file di log, lo stesso avviene per la stringa *STO GIRANDO A SINISTRA*. Nel caso in cui non riceva nessuna delle due stringhe sopra citate, scrive nel file di log *NO_ACTION*.
   Per scrivere nel file di log si utilizza la funzione *writeLog*.
   Tutte le scritture avvengono una volta al secondo.
   Infine si chiudono la connessione col server e il file di log.

3. **Throttle Control**

   Instaura la connessione con il server *ECU*, se andata a buon fine crea il file di log *throttle.log* con la funzione di utilità *openLog*.
   Entra nel loop dove si mette in ascolto col server e nel caso riceva il messaggio *INCREMENTO 5* esegue il metodo *probabilityOfBrake* (descritto successivamente nella relazione) e scrive nel file di log la data attuale tramite la macro predefinita *__DATE__* e la stringa *INCREMENTO 5*.
   Queste due scritture avvengono chiamando due volte consecutive la funzione *writeLog*, la prima chiamata scrive la data e la seconda la stringa richiesta.
   Infine si chiudono la connessione col server e il file di log.

4. **Brake By Wire**

   Instaura la connessione con il server *ECU* per poi creare il file di log *brake.log* con la funzione di utilità *openLog*.
   Usa la funzione *signal* che permette di gestire il segnale di interruzione nel caso venga inviato. Il segnale viene gestito dalla funzione *signalHandler* che si occupa di scrivere all'interno del file di log la stringa *ARRESTO*.
   Entra nel loop dove si mette in ascolto col server e nel caso riceva il messaggio *FRENO 5* scrive nel file di log la data attuale tramite la macro predefinita *__DATE__* e la stringa *FRENO 5*.
   Come nel caso precedente queste scritture avvengono con la doppia chiamata di *writeLog*.
   Per ultimo si chiudono la connessione col server e il file di log.

5. **Front Windshield Camera**

   Instaura la connessione con il server *ECU*, successivamente crea il file di log *camera.log* con la funzione di utilità *openLog*.
   Apre il file *frontCamera.data* in sola lettura definito dalla macro *DATA_FILE*.
   Entra in un ciclo in cui rimane finché c'è qualcosa da leggere in *frontCamera.data*.
   Toglie il carattere *\n* dal dato letto.
   Invia il dato letto al server tramite la funzione di utilità *sendMessageToServer* e scrive nel file di log il contenuto del dato con la solita *writeLog*.
   Successivamente vengono chiusi *frontCamera.data*, *camera.log* e la connessione con la *ECU*.

6. **Park Assist**

   Si deduce che Park Assist è al contempo client e server, questo perché i dati letti da Surround View Cameras vengono trasmessi alla ECU ma non in maniera diretta, questi infatti vengono

trasmessi al componente Park Assist che si occupa poi di inoltrarli alla ECU. In sostanza, Park Assist funge da ponte tra Surround View Cameras e la Central ECU.
Si connette al server *ECU* e crea la socket *ParkAssist*.
Crea il processo Surround View Cameras e quest'ultimo si connette al server *ParkAssist*.
Entra in un loop in cui legge il messaggio inviato da Surround View Cameras e lo inoltra alla *ECU* tramite la funzione *sendMessageToServer*.
Effettua la lettura da *dev/urandom* se il programma è stato avviato in modalità normale altrimenti legge da *urandomARTIFICIALE* con la funzione *readData (descritta successivamente)* che in aggiunta poi si occupa di inviare il contenuto letto alla *Central ECU* convertito in codice ASCII con la funzione *convertExToAscii*.
Legge il messaggio inviato da Surround View Cameras e lo inoltra alla ECU tramite la funzione *sendMessageToServer*.
Il codice inviato viene pure memorizzato nel file di log *assist.log* creato all'inizio del codice.
Infine si chiudono la connessione col server e il file di log.

7. **Central ECU**
   Nel file sono presenti svariati file descriptor, utili per la gestione delle varie connessioni e operazioni. Visto il numero elevato di componenti, per una migliore organizzazione, gestione e leggibilità si è deciso di utilizzare un vettore di descrittori di file. Lo stesso vale per i PID dei figli.
   Crea il file di log *ECU.log* usato per memorizzare le varie azioni attuate.
   Nel caso venga inviato il segnale di arresto da Throttle Control, questo viene gestito tramite la funzione *signalHandler* che provvede a inoltrare al file Out il messaggio di arresto.
   Si creano i vari processi figli e li si connettono al server centrale *ECU*.
   La ECU legge costantemente gli input inviati dal file In e sulla base di questi effettua le diverse operazioni.
   Si chiude la socket principale e il file di log.

I vari metodi utilizzati nella ECU, sono definiti nel file sorgente ecuUtils in modo da rendere più pulita la ECU e avere un file con tutte le funzioni utilizzate da questa.
I file sorgente logManagement e fileManagement forniscono rispettivamente le operazioni per gestire i file di log e per gestire le operazioni sui file.

Indicazione degli elementi facoltativi realizzati

1. **Componente facoltativo 1**
   Come già detto prima Throttle Control compara la stringa ricevuta dalla ECU e se questa corrisponde a *INCREMENTO 5* allora viene richiamata la funzione *probabilityOfBrake* definita all'interno del file *throttleControl.c*.
   Si deduce che questa operazione avviene ogni volta che si riceve un comando di accelerazione.
   Nella funzione *probabilityOfBrake* la prima istruzione presente è *srand(time(NULL))* che permette la dichiarazione del generatore di numeri casuali.
   Viene creata una variabile di tipo double con assegnata la probabilità di 10^(-5) e viene generato un valore compreso tra 0 e 1.
   Se il valore randomico è minore della probabilità prescritta allora viene stampata la stringa *ARRESTO TOTALE*, viene scritta la stringa *ACCELERAZIONE FALLITA* all'interno di *throttleControl.log* (motivo per cui si passa alla funzione il file descriptor di throttleControl.log) ed infine viene inviato il segnale di arresto alla Central ECU che lo gestisce.
   Riassumendo in breve, la funzione `probabilityOfBrake` genera un numero casuale e verifica se cade all'interno della probabilità di guasto al freno. Se ciò accade, viene segnalato un guasto, viene registrato un messaggio di errore nel file di log e viene inviato un segnale di interruzione al processo padre cioè Central ECU.

2. **Componente facoltativo 2**
   Instaura la connessione con la socket *ECU* utilizzando la funzione *connectTo*.
   Se la connessione va a buon fine, crea il file di log *radar.log* usando la funzione *openLog*.
   Entra in un loop infinito in cui legge da *dev/urandom* nel caso il sistema sia stato avviato in modalità normale, altrimenti legge da *urandom.BINARY* tramite la funzione *readData* che prende 5 diversi parametri. Il primo parametro è il file descriptor utile al fine di comunicare i

dati letti al server, il secondo è il puntatore al file *radar.log* dove si deve scrivere ciò che si è inviato alla ECU, il terzo corrisponde alla durata della lettura, il quarto serve per effettuare il controllo sul numero di byte letti, cosa richiesta nei componenti Forward Facing Radar e Surround View Cameras e il quinto parametro specifica la modalità di avvio.
All'interno della funzione *readData* la variabile *endTime* calcola il tempo di fine sommando al tempo in secondi dallo Unix epoch time la durata desiderata in secondi del ciclo e *currentTime* viene usata come condizione di uscita nel ciclo. Quest'ultima memorizza continuamente il tempo attuale in secondi e finché è minore o uguale a quello di *endTime* rimane nel ciclo, conclude altrimenti.
Una volta entrati nel ciclo di lettura se si leggono 8 byte, questi vengono convertiti da esadecimale in codice ASCII, si inviano in codice ASCII al server e li scriviamo nel file di log *radar.log*. Queste tre operazioni avvengono col richiamo di tre funzioni, vale a dire *convertExToAscii*, *sendMessageToServer* e *writeLog*.
Infine si chiudono la connessione col server e il file di log.

3. **Componente facoltativo 3**
   Quando la Central ECU riceve da In oppure da Front Windshield Camera il messaggio *PARCHEGGIO* viene chiamato il metodo *parkingProcedure,* che, oltre a ridurre ciclicamente la velocità fino a quando questa non risulta essere pari a zero, provvede a sospendere tramite un ulteriore ciclo i vari sensori e attuatori esclusi Park Assist e Surround View Cameras con la funzione di sistema *waitpid*.

4. **Componente facoltativo 4**
   A differenza della stragrande maggioranza dei componenti, Park Assist non viene generato all'avvio del sistema ma solo quando ce n'è bisogno, più precisamente viene generato all'interno della funzione *parkingProcedure*, vale a dire quando la ECU riceve il messaggio *PARCHEGGIO*.
   La metodologia di creazione del componente in questione è medesima agli altri componenti, viene infatti usata la *fork* per creare il processo figlio che a sua volta crea il processo Park Assist che, avviato, tenta di connettersi al server *ParkAssist*, nel frattempo il processo padre si mette in fase di accettazione.

5. **Componente facoltativo 5**
   Come già annunciato nel paragrafo della progettazione e implementazione Park Assist è un client e al contempo un server per i motivi sopra descritti.
   Park Assist invia i dati ricevuti da Surround View Cameras alla Central ECU prima leggendoli attraverso la funzione *receiveMessageFromClient*, salvandoli poi nel buffer *msg* ed infine inviandoli col metodo *sendMessageToServer*.

6. **Componente facoltativo 6**
   Instaura la connessione con la socket *ParkAssist* utilizzando la funzione *connectTo* definita nell'header *socketClient.h*, implementata nel file *socketClient.c*.
   Se la connessione va a buon fine, crea il file di log *cameras.log* usando la funzione *openLog* implementata nel file *logManagement.c*.
   Entra in un loop infinito in cui legge da *dev/urandom* sempre se è avviato in modalità normale, altrimenti la lettura avviene da *urandom.BINARY* tramite la funzione *readData*.
   Una volta entrati nel ciclo di lettura se si leggono 8 byte, questi vengono convertiti da esadecimale in codice ASCII, si inviano in codice ASCII al server, cioè a Park Assist e li scriviamo nel file di log *cameras.log*. Queste tre operazioni avvengono col richiamo di tre funzioni, vale a dire *convertExToAscii*, *sendMessageToServer* e *writeLog*.
   Infine si chiudono la connessione col server e il file di log.

7. **Componente facoltativo 7**
   - Non implementato.

8. **Componente facoltativo 8**
   - Per fare questa operazione la ECU utilizza la funzione definita all'interno di se stessa come *signalHandler* che provvede a porre la variabile *velocity* a 0 e scrivere nel file di log *ECU.log* la stringa *ARRESTO TOTALE* per poi inviarla al file Out.c e il programma termina la sua esecuzione.

Esempio di esecuzione di Park Assist e Surround View Cameras

Ripetiamo che il processo Park Assist non viene creato immediatamente dalla Central ECU ma solo quando occorre chiamando la funzione *parkingProcedure*, che a sua volta lo crea solamente quando questa ha terminato l'azzeramento della velocità.
A sinistra il terminale In e a destra Out.

```
FFR connesso
ARRESTO
Enter you message:
PARCHEGGIO
```

```
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: ARRESTO
```

Se si scrive ARRESTO nel terminale di sinistra, il terminale di destra mostra la stringa ricevuta, viene impostata la velocità a zero e ucciso il processo Brake By Wire.

```
forwardFacingRadar in esecuzione
FFR connesso
PARCHEGGIO
Enter you message:
parkAssist in esecuzione
PA connesso
surroundViewCamera in esecuzione
Kill all
SWC connesso
Park complete
```

```
Ricevuto da ECU.c: INCREMENTO 5
Ricevuto da ECU.c: INCREMENTO 5
Ricevuto da ECU.c: Parking...

Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
Ricevuto da ECU.c: FRENO 5
```

Per chiamare la funzione *parkingProcedure* digitiamo nel terminale la stringa *PARCHEGGIO* quando il sistema software è già avviato.
Quando si scrive PARCHEGGIO nel terminale di sinistra viene avviata la procedura di parcheggio, in quel momento sul terminale di destra viene mostrata la stringa *Parking….*
Sempre nel terminale di destra si visualizzano i vari impulsi di decremento velocità sotto forma di stringhe *FRENO 5*, quando la velocità raggiunge il valore 0 sul terminale di sinistra viene rappresentato l'avvio di Park Assist, la sua connessione col server *ECU*, l'avvio di Surround View Cameras e conseguentemente la sua connessione al server *ParkAssist*.
Arrivati a tal punto aspettiamo i 30 secondi di esecuzione di Park Assist in cui riceve pure i dati da Surround View Cameras. Terminati i 30 secondi possiamo visualizzare la stringa *Park complete* sul terminale di sinistra, l'esecuzione termina.
Infine possiamo andare ad aprire i file di log *cameras.log* e *assist.log* generati durante l'esecuzione e trovare al loro interno tutti i byte letti da *dev/urandom* oppure da *urandomBINARY* che rispettano le specifiche date.
La seguente immagine rappresenta parte dei dati presenti nei file *assist.log* e *cameras.log*.

1 1D10FF4D	1 FF3FFF27
2 4FFF0D7B	2 FFFF7FFF
3 FF66FFFF	3 0FFFFFFF
4 FFFFFFFF	4 7BFFFF41
5 58FF547A	5 FFFF730E