POLITECNICO DI TORINO

Tesi di laurea magistrale in
INGEGNERIA INFORMATICA

Sviluppo di un sistema di comunicazione intraveicolare

Docenti:

Prof. Guido Marchetto
Prof. Enrico Masala

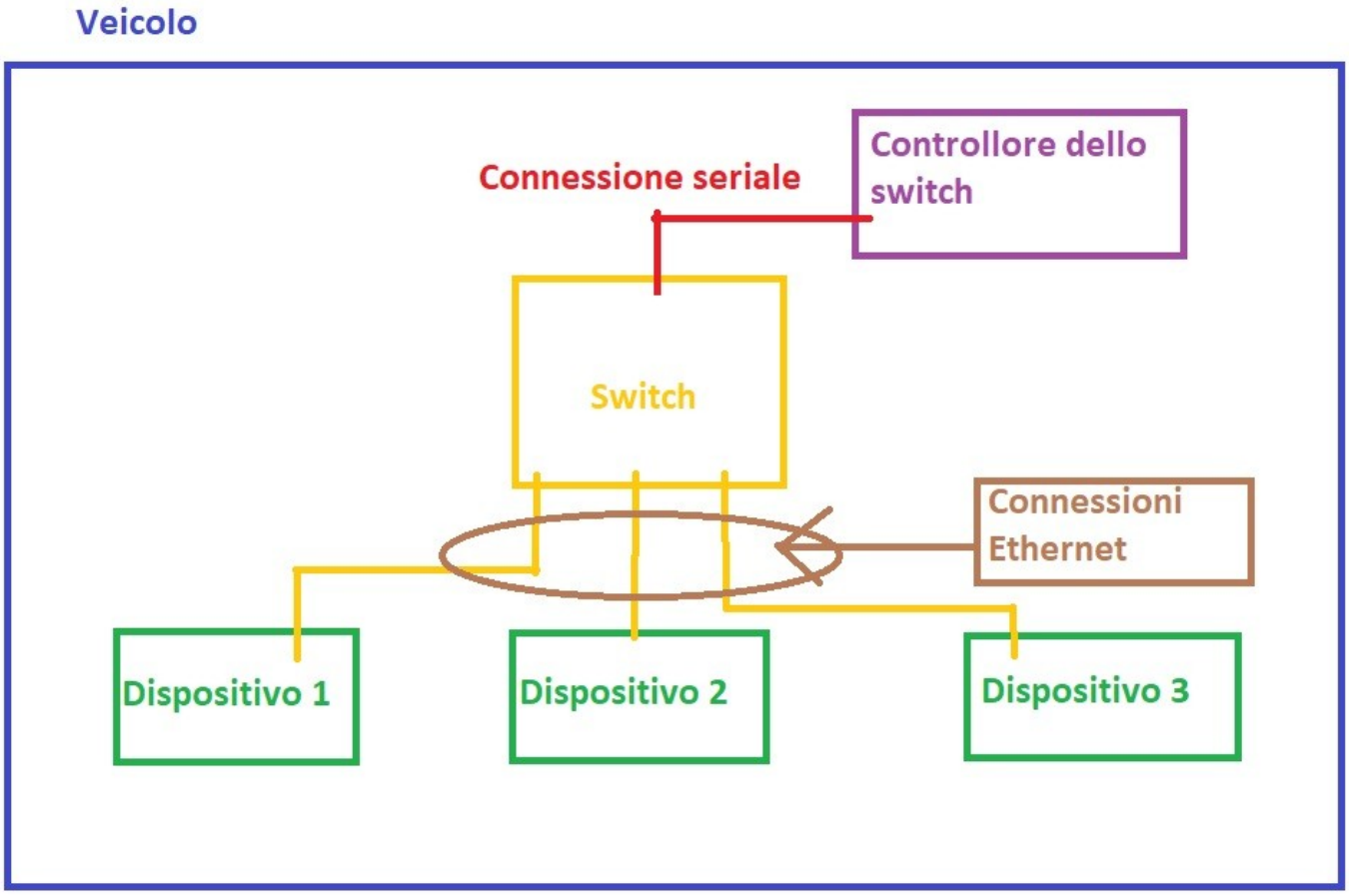
Candidato:

Davide Carrera

Indice:

1. Introduzione

L'elaborato seguente descrive mezzi e modalità di comunicazione a bassa latenza utilizzati da dispositivi informatici connessi ad una rete, in modo che questi possano interagire propriamente tra di loro.
Un valido esempio di questa situazione può essere quello di una comunicazione intraveicolare, nella quale due o più attori devono poter comunicare in modo veloce e con ritardi minimi, ad esempio per il controllo di un dispositivo gps, una videocamera, l'intervento della frenatura di emergenza ecc...
Per far ciò si possono utilizzare soluzioni facenti capo a diversi standard usati per le comunicazioni.
Uno di questi (quello da me utilizzato in questo percorso) è DDS (Data Distribution Service), standardizzato dalla OMG (Object Management Group), il cui scopo è proprio quello di permettere la distribuzione di dati in tempo reale.
Si tiene a precisare che il fine dell'attività svolta non è stato soltanto quello di apprendere concettualmente il funzionamento dei dispositivi utilizzati, ma di verificalo in pratica con programmi che andassero a testare alcune funzionalità.
Questo fa sì che il mio lavoro si sia dovuto svolgere inevitabilmente su più fronti, che saranno distinti e spiegati dettagliatamente nelle pagine seguenti.
Mi limito al momento a indicare i 2 filoni seguiti:
il primo è quello di verificare il funzionamento di "basso livello" dei vari componenti, in modo di avere la certezza di utilizzare strumenti tra di loro pienamente interoperanti; il secondo quello di implementare, nelle entità apposite, una logica che permettesse il raggiungimento degli scopi sopra citati.
Qui sotto è riportato uno schema che dia l'idea di quanto descritto.



Al centro troviamo uno switch, mediante il quale si intendono connettere diversi dispositivi ad una rete locale interna al veicolo.
In alto troviamo un dispositivo di controllo che, mediante una connessione seriale, modifica il comportamento dello switch in modo desiderato.
In basso troviamo appunto i vari dispositivi collegati via Ethernet al suddetto switch.
La prima parte del lavoro riguarda la connessione seriale, e quindi il controllo dello switch, la seconda parte riguarda l'implementazione di un protocollo di trasferimento dati (messaggi) tra i dispositivi riportati in basso in figura.

1.1. Strumenti utilizzati (Hardware e Software)

Switch Ethernet Microchip KSZ9477 a cinque porte programmabile mediante
1) Utilizzo di una Evaluation Board associata
2) Scrittura di valori nei registri, ciascuno associato ad un comportamento desiderato da parte di un componente dello switch stesso (es. apertura o chiusura di una porta).

L'utilizzo della Evaluation Board ha l'obiettivo di prendere confidenza con lo switch a cui è associata. Mediante questa si possono impostare direttamente i parametri dei registri mediante variabili di ambiente precaricate e notare le modifiche che vengono apportate.

La scrittura dei valori nei registri invece è il vero e proprio scopo dell'attività svolta: infatti mediante questa seconda opzione è possibile controllare lo switch da remoto attraverso un protocollo di comunicazione supportato (ad esempio spi, i2p, ecc...). Potenzialmente è dunque possibile programmare lo switch da qualunque piattaforma collegata ad esso.

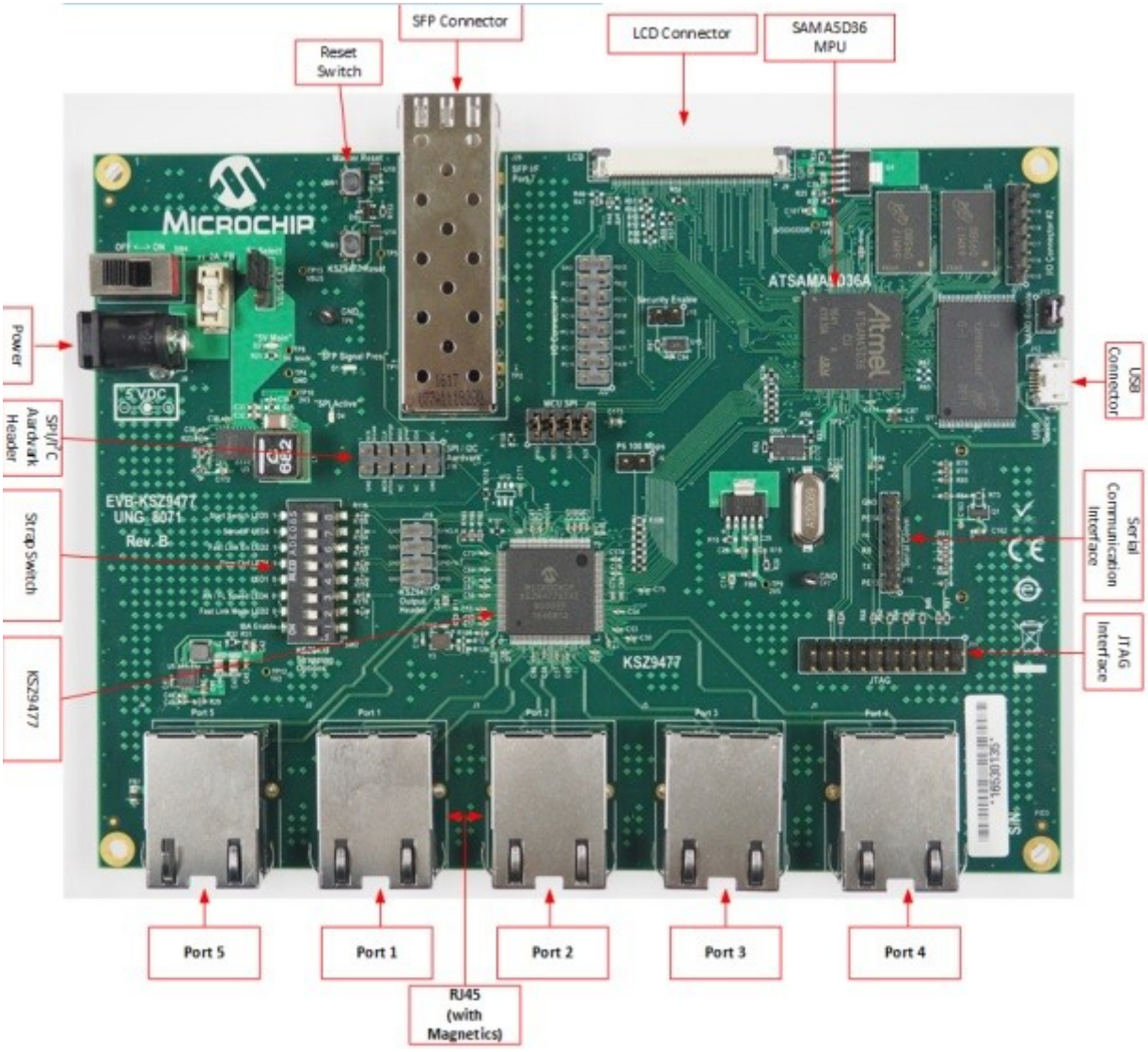


Immagine Switch KSZ9477 a 5 porte

Scheda SecoBoards-I.Mx-6_arm con installata a bordo una verisone di Linux in modalità CLI.
Questo è lo strumento utilizzato per:
1) Impartire i comandi allo switch
2) Eseguire la componente “intelligente” del software che è fondamentale per il corretto scambio di informazioni real-time.

I comandi vengono impartiti mediante programma C che esegue operazioni di read/write su bus spi (Serial Peripheral Interface), che è lo strumento fisico di comunicazione tra sistema esterno e switch.

La componente così detta “intelligente” viene gestita da un protocollo di tipo DDS, nella fattispecie dalla sua implementazione software OpenDDS.

Da queste considerazioni si deduce facilmente che da un punto di vista software si è deciso di utilizzare:

- Linguaggio C, per l’interazione switch-i.mx
- OpenDDS (C++) per l’elaborazione dei dati acquisiti dallo switch.

Immagine I.Mx6-SecoBoard

1.2. Compilazione Kernel i.mx con drivers switch

La prima parte del lavoro si è svolta cercando di compilare il kernel dell'apparato esterno i.mx in modo che in esso fossero inseriti i drivers necessari alla comunicazione via bus spi con lo switch.
Per far ciò come prima cosa bisogna scaricare tutte e 3 le componenti interessate:
- kernel Linux dell' i.mx
- drivers dello switch per il suddetto kernel
- cross-compiler per Linux X86-64 → Linux Arm-32

Di seguito elenco i link e il contenuto di ciascuno di essi

Kernel Linux:
https://git.seco.com/arm/nxp/imx6/bsp5/release/linux-3-14-28-secoboards-imx6-rel/-/archive/master/linux-3-14-28-secoboards-imx6-rel-master.tar.gz
A questo link si trova il codice sorgente del kernel che andrà compilato e la cui immagine verrà copiata sulla scheda i.mx

Drivers Switch:
https://github.com/Microchip-Ethernet/EVB-KSZ9477/archive/master.zip
A questo link si trovano i drivers dello switch che dovranno essere inseriti nel kernel scaricato al punto precedente, prima della sua compilazione.

Cross-compiler:
http://imx6-binary.seco.com/ToolChain/iMX6_arm-fsl-linaro-gnueabi-4.9.1.tar.gz
http://imx6-binary.seco.com/FILE_SYSTEM/iMX6_FS_ubuntu-minimal-5.1.tar.gz
In questa terza parte troviamo due link:

il primo scarica la vera e propria toolchain di compilazione, debug ecc..., il secondo scarica una copia del filesystem utilizzata da i.mx, che sarà indispensabile per la compilazione di programmi per quest'ultimo.

In *Appendice1* si trovano i comandi necessari per eseguire le operazioni preliminari per la compilazione.
A questo punto ci si pone nella condizione di avere tutti i file e le cartelle nei percorsi giusti e si descrivono i passi per la compilazione.

Come prima opzione si genera il makefile corretto mediante una simil-interfaccia grafica su terminale.
Aprire terminale nella cartella [Root] e lanciare il comando:

```
make menuconfig
```

in questo modo apparirà una interfaccia in cui si potranno selezionare le varie componenti del sistema da inserire (come moduli esterni o interni(built-in)).
A questo punto occorre seguire il percorso seguente:
`Device Drivers / Network Device Support / Ethernet driver support` e spuntare con [*] la voce `Micrel Devices`; fatto ciò si prosegue con il percorso in
`Drivers for Micrel switches` e si spuntano con [*] le voci `SPI driver for Micrel KSZ9897 switch`, `1588 PTP support`, `STP Support`.

Ci si sposta in nella barra in basso con il tasto *Tab* fino a premere con *Invio* sul tasto *Save*.
Viene richiesta un'ulteriore conferma di salvataggio nel file nascosto *.config* e il kernel sarà pronto per essere compilato.
Mediante il tasto exit ora è possibile uscire dalla schermata di configurazione e tornare alla CLI.
A questo punto è sufficiente eseguire il comando

```
LOADADDR=0x12000000 make
```

per completare la compilazione.

Ora bisogna spostare la ZImage generata ed il file contenente il Device-Tree (.dtb) nella partizione richiesta.

In questo caso la partizione è una scheda sd indirizzata dal file-descriptor */dev/sdcard1sp1,* che dovrà dunque essere montata nel sistema.

L'elenco dei comandi è il seguente:

Ora accediamo al dispositivo e montiamo la partizione desiderata:

```
ssh root@192.168.1.10

mount /dev/sdcard1sp1 /mnt/

exit
```

Ora copiamo i file del device-tree e della zImage:

```
scp                [root]/arch/arm/boot/dts/imx6q-
seco_quadmo747_928.dtb
root@192.168.1.10:/mnt/imx6q-
seco_quadmo747_928.dtb

scp                    [root]/arch/arm/boot/zImage
root@192.168.1.10:/mnt/zImage
```

Riaccediamo al dispositivo e smontiamo la partizione

```
ssh root@192.168.1.10

umount /dev/sdcard1sp1

exit
```

A questo punto occorre scollegare dalla corrente elettrica il dispositivo i.mx e ricollegarlo dopo un’attesa di circa 10s.

Se tutto è andato liscio, al riavvio, eseguendo l’accesso via terminale con il comando
`ssh root@192.168.1.10`,
spostandosi nella cartella */dev/*, si dovrebbero vedere i file-descriptor di due dispositivi spi, (*spidev1.0* e *spidev1.1*): quello da considerare sarà il primo.

2. Comunicazione Scheda I.Mx ↔ Switch

Come annunciato nel paragrafo precedente la comunicazione scheda I.Mx ↔ Switch è effettuata mediante bus spi (Serial Peripheral Interface).

Come dice il nome la trasmissione su questo tipo di bus avviene in modalità seriale (ovvero i bit vengono inviati sul canale uno dopo l'altro e giungono a destinazione nello stesso ordine in cui sono partiti).
Questo tipo di bus ha altre 2 importanti caratteristiche:
- la velocità di trasmissione è determinata in modalità sincrona da un clock: questo per coordinare trasmissione e ricezione
- l'invio e la ricezione di messaggi possono avvenire simultaneamente, dunque è un bus di tipo Full-Duplex

Poiché l'implementazione hardware è abbastanza semplice, il sistema viene utilizzato principalmente per mettere in comunicazione apparati vicini tra loro, in quanto non vi sono particolari accorgimenti per la possibile presenza di disturbi nel canale di comunicazione (cosa tipica nella condizione in cui le due entità si trovino ad una debita distanza).

L'invio e la ricezione dei messaggi avviene mediante dei registri a scorrimento, tipicamente della dimensione di 8 bit. Questo ha lo scopo di parallelizzare in un certo senso la ricezione dei dati: finché non vengono ricevuti 8bit il registro a scorrimento non si svuota e i dati non vengono memorizzati.

Da un punto di vista software la comunicazione avviene mediante un programma in C che legge/scrive su indirizzi specifici dello switch (corrispondenti agli indirizzi di inizio dei registri sui quali si desidera effettuare la scrittura/lettura).

Per far ciò si deve utilizzare un set di funzioni dichiarate negli header file `linux/spi/spidev.h` , `sys/ioctl.h` , `unistd.h` , e `fcntl.h` .

Il primo contiene sostanzialmente delle macro e delle strutture dati apposta definite per la comunicazione su bus spi,
il secondo contiene la dichiarazione della funzione `ioctl` (su cui effettuerò poco dopo una descrizione),
mentre il terzo e il quarto contengono le funzioni e le macro per l'accesso ai file in modalità read/write.

La funzione `int ioctl(int fd, int request, …)` permette di effetture input/output di basso livello su un qualsiasi dispositivo. Più in generale mediante questa funzione si può controllare una periferica.
Il primo parametro è il file-descriptor della periferica a cui si vuole accedere, il secondo è la richiesta che si vuole fare alla periferica e il terzo è il campo della richiesta (se necessario).
Questo vale in generale poiché, non esistendo uno standard per i parametri di questa funzione, ogni produttore implementa questa funzione a modo proprio.

N.B. Affinché le operazioni desiderate vadano a buon fine il file deve essere aperto in modalità read-write (macro `O_RDWR` definita in fcntl.h).

Per quanto riguarda in particolare il caso di un bus spi, prima di effettuare il trasferimento dei dati (quindi la scrittura e la lettura nei/dai regitri dello switch) occorre impostare alcune proprietà di trasferimento dati tipici del prodotto utilizzato.

Come prima cosa il file descriptor aperto è quello in posizione /dev/spidev1.0, e, come detto, deve essere aperto in lettura/scrittura.

Una volta effettuato il controllo sulla corretta apertura si procede con il settaggio dei parametri per i trasferimenti successivi, usando sin da ora la funzione `ioctl`.
Si tiene presente che un errore viene identificato da un valore restituito pari a -1.

Per prima cosa si setta la modalità di lettura/scrittura.
Questo avviene mediante la seguente chiamata:

```
ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
```

in cui il valore di mode è inizializato a 0.
Altri valori di mode possono essere assegnati mediante l'operatore or bit-a-bit |=
Ad esempio con il comando `mode|=SPI_CS_HIGH` si forzeranno le operazioni rw sul fronte di salita del clock, e così via.
Nel mio caso il tutto ha funzionato tenendo il campo mode a 0 sia in lettura che in scrittura.

La seconda coppia di impostazioni da effettuare è quella che riguarda i bit per word trasferiti, ovvero dopo quanti bit deve considerarsi effettuato il trasferimento.
Siccome il registro a scorrimento ambo i lati è da 8bit, questo sarà il valore da impostare.
Le due chiamate sono dunque:

```
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
```

in cui appunto il campo bits è settato a 8.

Resta ancora da impostare la massima velocità di trasferimento sul bus.
Anche qua bisogna effettuare una operazione simile alle due precedenti.

In questo caso però c'è da considerare il fatto che la velocità deve essere impostata in modo che entrambe le estremità ne siano compatibili.
Dunque il valore impostato sarà il minimo tra le velocità massime di trasferimento (uguali in lettura e scrittura) di switch e i.mx.
In questo caso il minimo è quello dello switch ed è di 2,4MHz, che equivale a 2400000Hz.

Fatte queste considerazioni veniamo alle 2 chiamate:

```
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
```

in cui speed è la frequenza espressa in Herz e dunque è impostata al valore 2400000.

Finita tutta questa pappardella di setup iniziale, possiamo proseguire con il trasferimento vero e proprio dei dati su canale spi.
Per prima cosa vorrei mettere subito in evidenza una considerazione sul funzionamento dell'input-output del dispositivo switch; questo presenta infatti una caratteristica peculiare: siccome i trasferimenti da/verso registri avvengono al massimo a gruppi di 1Byte, nel momento in cui si desidera scrivere valori su di un registro da 2Byte è chiaramente necessario effettuare 2 operazioni di scrittura (la prima all'indirizzo x e la seconda all'indirizzo x+1, considerando x l'indirizzo del byte meno significativo).
Questo pone un problema: l'operazione di aggiornamento di un registro non avviene nel momento della scrittura di un suo byte, poiché questo porterebbe a delle inconsistenze durante le operazioni di scrittura.
Supponiamo che si voglia scrivere il valore esadecimale
05 0E in un registro collocato agli indirizzi x e x+1, con valore iniziale 00 00.
Se ad ogni operazione di scrittura venisse aggiornato lo stato dello switch con i paramentri nuovi, si passerebbe

attraverso gli stati 00 0E o 05 00, a seconda che venga scritto prima il byte all'indirizzo x o x+1. E questa è una operazione certamente non voluta (e soprattutto sbagliata).

La soluzione a questo problema è stata quella di considerare un registro aggiornato quando sono stati scritti tutti i byte di quel registro, non esistendo comandi espliciti per inizio e fine transazione.
Dunque nel momento della scrittura di un byte, il sistema tiene conto delle operazioni di scrittura sugli altri bytes del registro. Quando tutti sono stati scritti almeno una volta, quello è il valore effettivo del registro stesso.
In questo modo, considerando sempre l'esempio di prima, lo switch vedrà il valore del registro passare da 00 00 a 05 0E, senza gli step intermedi.

Questo ci pone difronte ad un altro piccolo (e di facile soluzione) problema.
Supponiamo di avere il registro precedente inizializzato con i valori 05 00, e di voler scrivere il valore 0E nel byte all'indirizzo x (lasciando inalterato quello in x+1).
Siccome a priori il valore contenuto nel registro non è noto, e non possiamo soltanto scrivere 0E all'indirizzo x (poiché, come detto prima, non verrebbe considerato aggiornato in quanto non tutti i byte del registro lo sono) è necessario scrivere anche il valore contenuto in x+1.
Per far ciò l'unica soluzione e quella di memorizzare il valore in x+1 mediante una lettura e successivamente effettuare una write sia in x+1 (con il valore letto) che in x (con il valore che si desiderava impostare sin dall'inizio).

Questo esempio ci permette di capire che dato un registro di n bytes, sul quale voglio scrivere m (con m<=n) bytes, l'unico modo per effettuare una scrittura corretta è di effettuare un numero di operazioni pari a (m-n) read e n write: le write comprendono sia i valori da scrivere "nuovi" sia quelli appena letti che devono essere riscritti tali e quali.

Ora che sappiamo il modo in cui avvengono la scrittura e la lettura su un registro possiamo addentrarci nella struttura dati che descrive l'operazione.

Intanto i parametri da considerare sono sostanzialmente tre:
1) Tipo di operazione (read/write)
2) Indirizzo del registro in cui effettuare l'operazione del punto 1
3) Dato da leggere/scrivere

La struttura dati che contiene tutte queste informazioni è di tipo `struct spi_ioc_transfer` e contiene i seguenti campi:

- `tx_buf`: contiene l'indirizzo (sotto forma di unsigned int) del buffer usato in trasmissione.
- `rx_buf`: contiene l'indirizzo (sotto forma di unsigned int) del buffer usato in ricezione.
- `len`: contiene la dimensione (int) dei buffer sopra descritti (N.B. entrambi i buffer devono essere di uguale dimensione)
- `delay_usecs`: il tempo di ritardo (espresso in microsecondi, int) tra una operazione di scrittura e una successiva di lettura (valore default = 0).
- `speed_hz`: la frequenza alla quale far avvenire il trasferimento (valore default quello impostato in precedenza).
- `bits_per_word`: il numero di bit per word trasferita (vedere spiegazione sopra, valore di default quello impostato in precedenza).

Nel mio caso si è deciso di indicare espressamente tutti i campi, compresi quelli con un valore di default.
Pertanto si imposta a 0 il campo `delay_usecs` a 2400000 il campo `speed_hz` e a 8 il campo `bits_per_word`.

Volevo adesso spendere due parole per quanto riguarda i primi due campi, poiché questi sono i punti in cui ho riscontrato i maggiori problemi durante il proseguo del lavoro.

Il buffer di invio permette di effettuare le operazioni di lettura e scrittura sui registri dello switch. Esso infatti contiene al suo interno sia il flag read/write, sia l'indirizzo del registro (o del byte nel registro, nel caso di registri da 16 bit in su), sia il dato da scrivere (nel caso di una read questo terzo campo sarà trascurato).

Il buffer è espresso nel seguente formato (esadecimale):
xx yy yy yy zz , in cui xx vale 04 per il write e 06 per il read, yy yy yy è l'indirizzo del registro moltiplicato per 32, e zz è il valore da scrivere oppure don't care in caso di read (tipicamente 00).
N.B.: Considerato che lo switch utilizza valori a 16 bit per indirizzare i registri, il valore massimo assunto dal campo yy yy yy sarà pari a (2^16-1)*32 = 1F FF E0, quindi non tutto il range di valori viene utilizzato, (stessa cosa peraltro valida anche per il campo xx, in cui abbiamo solo 2 valori possibili).

Nel caso il campo xx sia posto a 06, e quindi si voglia effettuare una lettura, entra in gioco il buffer di ricezione, che, poiché deve essere della stessa dimensione di quello di trasmissione, sarà del tipo 00 00 00 00 ww, dove ww è l'unico campo significativo e rappresenta gli 8 bit letti dal registro desiderato.

Per rendere chiare le idee, supponiamo di voler scrivere il valore 1C nel registro avente indirizzo 05 0E. In questo caso xx = 04, yy yy yy = (05 0E)*32_{10} = 00 A1 C0 e
zz = 1C.
Nel caso poi voglia leggere il valore appena scritto, allora dovrò impostare xx = 06, yy yy yy = 00 A1 C0 e zz=00

In questo caso, dopo la chiamata ioctl, nel buffer di ricezione avrò i valori 00 00 00 00 1C. Dunque ww, ovvero il valore che mi interessa leggere, vale 1C.

Si presti attenzione al fatto che non tutti i bit dei vari registri sono scrivibili, quindi nel caso in cui si tenti di scrivere su memoria non scrivibile, il bit considerato verrà semplicemente ignorato.
Ad esempio se un dato registro da 8 bit è di tipo xxxxxyyx dove, con x indico i bit scrivibili e con y quelli riservati, supponendo che il valore iniziale del registro sia 00000000 e io voglia scrivere il valore 00000101, il registro conserverà soltanto il valore 1 meno significativo e non quello precedente.

2.1. Esempio codice C di scrittura/lettura

Supponiamo di voler scrivere il valore 0x1C nel registro da 8bit avente indirizzo 0x10A3 e, successivamente, leggere il contenuto del registro.

Come prima operazione calcoliamo i valori xx, yy yy yy e zz, per costruire il campo `tx_buf` della struct sopra descritta.

xx=0x04, in quanto si tratta di una operazione di scrittura;
yy yy yy =(0x10A3)*32_{10}=0x021460, indirizzo del registro;
zz=0x1C, il valore che voglio scrivere.

A questo punto possiamo procedere con il codice C:

```
char tx[]={0x04, 0x02, 0x14, 0x60, 0x1C},
     rx[]={0x00, 0x00, 0x00, 0x00, 0x00};
```

in questa parte dichiaro i buffer di trasmissione e ricezione, così come descritti testualmente prima.

A questo punto dichiaro la struttura dati completa e la inizializzo con i vari campi:

```
struct spi_ioc_transfer tr =
{
   .tx_buf=(unsigned long) tx,
   .rx_buf=(unsigned long) rx,
   .len=5,
   .delay_usecs=0,
   .speed_hz=2400000,
   .bits_per_word=8
}
```

Si noti che gli indirizzi dei buffer tx e rx sono convertiti ad unsigned long, perché così è richiesto dai campi della struct.

A questo punto possiamo effettuare la chiamata alla funzione `ioctl` nel seguente modo:

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

in cui il secondo campo è una marco a cui si passa il numero di elementi del tipo del terzo parametro passati (nel nostro caso 1), e il terzo parametro è un puntatore alla struct costruita prima.
Se l'operazione di scrittura è andata a buon fine il valore di `ret` sarà diverso da -1 e il registro avrà il valore corretto.

La seconda metà dell'esempio riguarda invece la lettura del valore appena scritto.
Scriviamo nuovamente i valori xx yy yy yy zz.

xx=0x06, in quanto si vuole effettuare una lettura;
yy yy yy=0x021460, lo stesso registro di prima;
zz=0x00, in quanto non si vuole scrivere nulla.

Detto ciò, creiamo i due buffer:

```
char tx[]={0x06, 0x02, 0x14, 0x60, 0x00},
     rx[]={0x00, 0x00, 0x00, 0x00, 0x00};
```

inizializiamo la struct (esattamente come prima), e chiamiamo la funzione `ioctl`:

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

Se anche sta volta va tutto a buon fine e il valore di `ret` è diverso da -1, allora nel buffer di ricezione, in posizione 4, troveremo il valore 0x1C, ovvero `rx[4]` conterrà `0x1C`.

Da questo esempio si nota facilmente il fatto che possono essere scritti/letti tutti i registri dello switch, e dunque, seguendo il manuale dello stesso si può controllare da remoto (nel nostro caso via bus spi) lo switch in tutte le sue funzionalità

Con queste considerazioni possiamo considerare conclusa la parte di comunicazione switch-i.mx. Nel prossimo paragrafo vedremo come rendere utili le informazioni reperite dallo switch e come manipolarle mediante un protocollo di più alto livello.

Qui di seguito si trova il link al sorgente del programma poc'anzi descritto:

https://drive.google.com/file/d/17w8TmOkH8RDgLCmGvc670sQI-f3EPowl/view

2.2. Cenni aggiuntivi

Per completezza viene riportato di seguito un secondo metodo utilizzabile per la lettura e scrittura da/sui registri.

In questo caso non viene utilizzata la struttura dati `struct spi_ioc_transfer` ma semplicimente un buffer di bytes (char).

L'operazione consiste nel riempire il buffer con valori stabiliti, in maniera analoga alla precedente.

Per la scrittura di dati il buffer (in binario) da passare alla `ioctl` è il seguente:

`010 00000000`$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$ `XXXXX` $D_7D_6D_5D_4D_3D_2D_1D_0$

mentre per la lettura è il seguente:

`010 00000000`$A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$ `XXXXX` $D_7D_6D_5D_4D_3D_2D_1D_0$

in cui i bit A_x sono l'indirizzo del registro, gli X sono dei don't care e i bit D_x sono i bit di dato nel caso di scrittura o don't care nel caso di lettura.

Per riprendere l'esempio di prima, nel caso in cui voglia scrivere il valore 0x1C nel registro 0x10A3, il buffer da inviare sarà in questo caso composto dai seguenti campi:

comando: 010(scrittura) 011(lettura)

registro (A_{15} .. A_0): 0001 0000 1010 0011

dato (D_7 .. D_0): 0000 0000 0001 1100 (scrittura)
: don't care (lettura)

3. Analisi protocollo DDS

La seconda parte del lavoro si è svolta utilizzando un protocollo di tipo DDS, in modo da permettere il raggiungimento degli obiettivi prefissati descritti nell'introduzione.

Intanto si elencano alcune caratteristiche di questo protocollo, per le quali può essere preferito rispetto ad altri suoi simili:

1)Indipendenza dalla piattaforma di esecuzione:
Utilizza un linguaggio descrittivo (*IDL) per la rappresentazione delle classi che è indipendente dal linguaggio di programmazione e dall'hardware sottostante

2)Scalabilità:
Poiché l'accoppiamento tra le entità è lasco (in quanto il paradigma utilizzato è di tipo publish/subscribe topic-based), si può definire un numero indefinito di attori e di topics, cosa che rende estensibile il software scritto.

3)Efficienza:
Resa possibile grazie alla comunicazione diretta tra publisher e subscriber.

4)Determinismo nella consegna dei dati e nella corretta ricezione:
DDS utilizza un protocollo di livello 5 che può funzionare sia utilizzando UDP sia TCP come protocollo di livello 4. Nel secondo caso la consegna corretta dei pacchetti è sempre garantita.

5)Flessibilità dell'architettura di rete:
Grazie al discovery automatico DDS presenta un punto di forza nella grande adattabilità che offre dal punto di vista degli attori presenti nella rete.

3.1. API e livelli strutturali

Uno degli scopi principali di DDS è quello di nascondere l'implementazione della comunicazione al programmatore, consentendogli solamente di definire alcune proprietà della stessa (ad esempio TCP/UDP, Unicast/Multicast ecc…).
Per far ciò vengono definite delle API (Application Programming Interface) che aiutano il programmatore nell'interfacciarsi con il sistema stesso.

Esse sono principalmente suddivise su due livelli, di cui uno opzionale (il primo descritto in seguito) e uno facoltativo:

1)Data Centric Publish/Subscribe(DCPS):
E' la parte che definisce sostanzialmente le proprietà della comunicazione con la rete.
In essa infatti sono contenute le definizioni di strutture dati (mediante interfacce IDL), livello di Quality of Service, metodo di discovery utilizzato (ne parleremo dopo) e altri ambiti sempre comunque inerenti alla mera comunicazione tra le varie entità.

2)Data Local Reconstruction Level(DLRL):
E' quella componente che, come dice il nome, è svolta localmente su ogni entità (cioè senza interazioni con l'esterno) il cui scopo principale è quello di estrarre gli oggetti provenienti dalla rete (e quindi da DCPS) e inserirli all'interno di strutture dati locali che mappino gli stessi.
In questo modo qualsiasi aggiornamento ad un oggetto in rete verrà reso effettivo anche sulla singola macchina in modo del tutto trasparente al programmatore e all'utilizzatore.

Nel proseguo della tesi ci si concentrerà maggiormente sul primo livello, che è quello in cui si interviene di più.

A questo punto ci soffermiamo un momento sulle entità che compongono il primo strato dei due descritti sopra (DCPS) in modo da aver le idee chiare quando dopo verrà presentato un esempio di programma, con relativi commenti al codice scritto (in maniera simile a quanto fatto per la parte1).

1)Domain Partecipant Factory (DPF):
Classe singleton (che può essere instanziata una volta sola) e che funge da factory verso lo spazio dei domini di comunicazione DDS (chiamato GDS, Global Data Space), ciascuno dei quali è identificato da un nome.

2)Domain Partecipant (DP):
E una classe che viene instanziata dal DPF, ed è, a sua volta, una factory che viene utilizzata per instanziare Publishers, Subscribers e Topics.

3)Topic:
E' una classe che viene instanziata da un DP, il cui scopo è quello di definire una componente tipizzata del dominio di comunicazione in cui possono essere scambiati messaggi di un certo tipo. Il tipo di messaggi scambiati viene dichiarato attraverso una struttura dati C-Like, alla quale però vi è l'aggiunta del fatto di poter dichiarare uno o l'insieme di alcuni campi come chiave primaria, in modo del tutto simile a quanto avviene nei Database relazionali.
Ciascun Topic è identificato univocamente (all'interno di uno stesso GDS) da un nome.

4)Publisher-DataWriter:
E' la coppia di componenti predisposte alla creazione e all'invio dei messaggi verso la rete.

In particolare il Publisher è un'entità che svolge la funzione di factory per la creazione di uno o più DataWriter, ovvero i processi effettivi di invio dei messaggi.

Il DataWriter è associato univocamente ad un singolo topic, e dunque rimane un'entità astratta fin tanto che non viene tipizzata con il tipo di dato del topic a cui si riferisce (si capirà meglio nell'esempio di seguito).

5)Subscriber-DataReader:
Entrambe le componenti svolgono il lavoro complementare a quelle del punto 4. Dunque il subscriber instanzia dei DataReader, i quali verrano resi concreti tipizzandoli con i topic che vogliono essere associati a ciascuno di essi, e fungeranno da ricettori dei messaggi provenienti dalla rete.

Per quanto concerne la Quality of Service, il protocollo DDS la implementa soprattutto a livello di qualità di comunicazione tra i diversi nodi della rete, che è la parte che maggiormente può risentire di perdite di dati e perciò quella su cui si è ritenuto più opportuno agire.

Le 5 proprietà che definiscono la QoS in DDS sono:

1)Ownership:
Viene applicata ai Topic e indica che quest'ultimo è aggiornabile soltanto dal DP che lo ha creato (in questo caso sarà di tipo "Exclusive") oppure da qualunque DP ne abbia accesso (in questo caso sarà di tipo "Shared")

2)Strength:
Fortemente legata ad una Ownership di tipo Exclusive; indica la priorità di aggiornamento di un Topic assegnata ai DataWriter creati a partire da uno stesso DP (l'unico avente diritto di aggiornamento di un Topic).
Può essere assegnata anche a Topic di tipo Shared, ma, statisticamente parlando, è meno comune.

3)Destination Order:
Si applica ai DataReader, ciascuna per ciascun Topic;
indica l'ordine con cui devono essere posizionati i dati ricevuti.
Gli ordini possibili sono fondamentalmente 2:
- in base al tempo di immissione in rete da parte del publisher creante
- in base al tempo di ricezione del DataReader stesso.
Quest'ultima è l'impostazione predefinita.

4)Liveliness:
Stabilisce un criterio per determinare la presenza o meno di ognuno dei partecipanti alla comunicazione DDS in corso.
In particolare viene utilizzata per stabilire se una delle entità abbia o meno "tagliato la corda".
Quest'espressione può sembrare sgarbata, ma in realtà descrive esattamente quanto avviene.
Quando un partecipante vuole scollegarsi può farlo chiamando delle procedure opportune che avvisino della sua dipartita.
Se invece ciò non viene fatto, l'unico modo che ha DDS per accorgersene è quello di settare un timer per ciascun partecipante e reimpostarlo ogni qual volta questo si fa vivo. Nel caso non si faccia sentire per un tot di tempo preimpostato, lo si considera non più connesso e le risorse ad egli dedicate vengono liberate.
I valori possibili sono 3:
-Automatico (di default): viene definito un parametro che indica ogni quanto il partecipant invia un messaggio di conferma della sua presenza.
-Manuale lato Topic/Partecipant: il partecipant viene considerato attivo fin tanto che sono attivi il partecipant stesso e/o il topic a cui fa riferimento.

N.B. In questo caso l’avviso di chiusura deve essere inviato manualmente dal partecipant.

5)Reliability:
Indipendentemente dal protocollo di livello 4 utilizzato (UDP o TCP) specifica se una determinata connessione deve essere di tipo affidabile oppure best-effort.
Se si imposta la proprietà come “affidabile”, si dovrà anche specificare il tempo oltre il quale, in mancanza di ricezione di uno o più ack, vengono rispediti il/i pacchetto/i mancante/i.

3.2. OpenDDS: l’implementazione opensource di DDS

Adesso ci si addentra nella descrizione di OpenDDS, sviluppato da Object Computing, Inc, e attualmente alla versioneo 3.13.

Come prima cosa, si presenta la modalità di compilazione dell’ambiente sia per pc (dunque per processore x86) che per i.mx (arm).

La versione da me utilizzata si trova al seguente link:
http://download.objectcomputing.com/OpenDDS/previous-releases/OpenDDS-3.13.2.tar.gz

Per quanto riguarda arm, l’idea è quella di riutilizzare il cross-compiler utilizzato nella parte1, questa volta per compilare l’ambiente OpenDDS per arm.

In *Appendice2* si trovano le operazioni preliminari per far sì che i file siano già in posizione giusta e si possano qui di seguito descrivere i passi della compilazione vera e propria.

N.B.:
Si tiene a precisare che se si esegue questo elenco di operazioni subito dopo aver eseguito la parte1 (quella relativa a i.mx), occorre effettuare l’unset della variabile di ambiente *CROSS_COMPILE*, con il seguente comando:

```
unset CROSS_COMPILE
```

Compilazione per x86:

Spostarsi nella cartella *[root-dds-x86]* e ivi aprire un terminale.

Ora si esegue la configurazione per la piattaforma desiderata (in questo primo caso quella su cui si trovano i sorgenti, ovvero il PC):

```
./configure
```

Dopo un po’ di operazioni eseguite la configurazione è effettuata e si può lanciare il comando

```
make
```

che esegue il build vero e proprio.

Finite queste 2 operazioni la compilazione OpenDDS lato x86 è finita.

Compilazione per arm:

Per quanto riguarda l'istanza per I.Mx, qui le cose sono leggermente diverse, in quanto l'ambiente deve essere compilato per una piattaforma che risulta diversa da quella in cui ci troviamo.

Dunque le operazioni da eseguire sono sempre le 2 di prima, ma con istruzioni in più per indicare la piattaforma di esecuzione e compilazione.

Spostarsi nella cartella *[root-dds-arm]* e aprire qui un terminale.

Eseguire la configurazione dell'ambiente per la compilazione con target linux per arm, con il seguente comando:

```
./configure –target=linux-cross –target-compiler=/
opt/yocto/fsl-release-bsp/build_seco/tmp/sysroots/
x86-64-linux/usr/bin/arm-poky-linux-gnueabi/arm-
poky-linux-gnueabi-g++
```

Ora compilare separatamente le componenti Host e Target del sistema:
Host:

```
cd [root-dds-arm]build/host

make
```

Target:

```
cd [root-dds-arm]build/target

CXXFLAGS=”-mfloat-abi=hard”     CPPFLAGS=”-mfloat-
abi=hard” make
```

Una delucidazione sui termini Host e Target:

Nella cartella Host si trovano le componenti del sistema che devono essere compilate per x86, mentre in Target quelle per arm.
Una cosa apparentemente strana è che, pur volendo compilare l'ambiente per arm, devo anche avere delle "parti" compilate per x86, cosa che a prima vista risulta inutile.
In realtà il problema non si porrebbe se io compilassi i programmi OpenDDS direttamente su I.Mx, e quindi avessi qui un compilatore C++. Dato che così non è, e quindi le operazioni per una e l'altra piattaforma vengono eseguite sempre su PC, che svolge il ruolo di sistema Host, si riesce a capire il motivo di questa apparente contraddizione.

A questo punto abbiamo il sistema compilato per entrambe le piattaforme desiderate.

Ora la seconda istanza compilata, quella che si trova in [root-dds-arm], deve essere copiata sulla scheda I.Mx, in modo da renderla operativa nel suo luogo di appartenenza.

Per far questo ci si reca da terminale nella cartella padre di [root-dds-arm] e si esegue una operazione di compressione, con il comando:

```
tar -czvf OpenDDS-3.13.2.tar.gz  OpenDDS-3.13.2/
```

Tale operazione è richiesta per poter spostare il tutto su I.Mx come unico file compresso, che verrà poi scompattato sulla scheda.

Ora copiamo il file compresso appena creato su I.Mx, con il seguente comando:

```
scp                              OpenDDS-3.13.2.tar.gz
root@192.168.1.10:/OpenDDS-3.13.2.tar.gz
```

Ora occorre estrarre il file copiato collegandoci via ssh all'I.Mx per poter impartire i comandi a quest'ultimo.

```
ssh root@192.168.1.10
```

E adesso procediamo con l'estrazione

```
tar -xzvf /OpenDDS-3.13.2.tar.gz
```

e la rimozione del file compresso copiato

```
rm /OpenDDS-3.13.2.tar.gz
```

Chiudiamo la connessione ssh per finire questa parte:

```
exit
```

3.3. Descrizione programma di esempio

In questa parte descrittiva, si assume una configurazione DDS di tipo centralizzato, ovvero in cui publisher e subscriber forniscono dati e attingono dati da una entità comune, chiamata broker.

In seguito vedremo un'altra configurazione in cui ogni attore apre una connessione in multicast mode per la discovery degli altri partecipanti.

La configurazione più semplice è dunque quella rappresentata da un processo publisher, un processo subscriber, e un processo broker, quest'ultimo atto a gestire gli scambi tra i primi due.

Il programma da cui lanciare il processo broker è già disponibile da ambiente OpenDDS compilato e si trova in *[root-dds-x86]/dds/InfoRepo/DCPSInfoRepo*
per x86 e in
[root-dds-arm]/build/target/dds/InfoRepo/DCPSInfoRepo
per arm.

I programmi che gestiscono i processi publisher e subscriber devono invece essere scritti in C++ e in seguito compilati. Ora vedremo come si svolge tutto ciò, dettagliatamente.

Questo esempio di programma permette di gestire l'invio e la ricezione di messaggi di un determinato tipo tra un publisher e un subscriber, per mezzo del già citato broker.
Il topic è singolo, così come la struttura dati dei messaggi scambiati.

Il primo passaggio è quello di definire, in un file con estensione .idl, la struttura dati dei messaggi.
Il file ha nome *Esempio.idl*.

```
module Esempio {

#pragma DCPS_DATA_TYPE "Esempio::Prova"
#pragma DCPS_DATA_KEY "Esempio::Prova id"

  struct Prova {
    int id;
    string str;
  };
};
```

Il nome del module è l'equivalente del namespace c++, in cui ci sono le diverse classi (struct) utilizzate.

La prima direttiva #pragma invece serve ad indicare il nome assegnato alla struttura dati contenente i campi relativi al tipo del messaggio.

La seconda direttiva #pragma invece indica quale/i campi della struttura è/sono da intendersi come chiave/i.

Infine abbiamo la definizione della struttura dati vera e propria.

Anticipo in questa sezione che tale file verrà analizzato prima che il compilatore esegua il build del sistema, e verranno generati automaticamente 3 file a partire da questo: *EsempioTypeSupport.idl*, *EsempioTypeSupportImpl.h*, *EsempioTypeSupportImpl.cpp*.
Il primo file continene le interfacce utilizzate dai DataReader e DataWriter (approfondiremo questo aspetto dopo).
Il secondo e il terzo contengono le definizioni C++ dei tipi di dato descritti nel file .idl iniziale.

Questa operazione, precedente a quella di compilazione di publisher e subscriber, viene configurata all'interno di un file di configurazione globale del progetto.

Tale file ha estensione .mpc e nel nostro caso si chiama *Esempio.mpc* ed è composto di 3 parti: una comune a publisher e subscriber, una solo per il publisher e una solo per il subscriber. Le analizzerò nell'ordine in cui le ho elencate.
Parte comune:

```
project(*idl): {

  TypeSupport_Files {
    Esempio.idl
  }
}
```

La prima riga indica al sistema OpenDDS che il nome di quel sotto-progetto si chiama *idl*.

La parte importante è quella identificata da `TypeSupport_Files`, che contiene l'elenco dei file idl richiesti dal progetto (nel nostro caso solo quello descritto sopra).

Ora veniamo alla parte esclusiva del publisher:

```
project(*publisher) : {
  exename   = publisher
  after    += *idl

  TypeSupport_Files {
    Esempio.idl
  }

  Source_Files {
    Publisher.cpp
  }
}
```

Dinuovo ritroviamo degli elementi già descritti prima, quindi mi limiterò, e così farò d'ora in poi, a descrivere le parti "nuove".
`exename = publisher` serve ad indicare il nome del file eseguibile generato, in questo caso "publisher", mentre `Source_Files` contiene un elenco di file sorgenti da compilare.
`after += *idl` indica un elenco di project che devono essere conclusi per poter iniziare a analizzare quello corrente. In sostanza serve per dare un ordine all'ambiente sui vari sotto-progetti di cui fare il build.

La parte del subscriber è invece questa:

```
project(*subscriber) :  {

  exename   = subscriber
  after    += *publisher

  TypeSupport_Files {
    Esempio.idl
  }


 Source_Files {
    DataReaderListenerImpl.cpp
    Subscriber.cpp
  }
}
```

L’unica differenza è che in questo caso il progetto è composto da 2 file sorgenti cpp, che sono elencati nella apposita struttura.

Veniamo ora ai sorgenti C++ partendo da Publisher.

```
try {

    DDS::DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs(argc, argv);


DDS::DomainParticipant_var participant =
dpf->create_participant(42,
PARTICIPANT_QOS_DEFAULT,0,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!participant) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l:main() -")
   ACE_TEXT(" create_participant failed!\n")),-1);

}
```

In questo blocco di istruzioni si crea il partecipant.
Come prima cosa si crea una istanza concreta della Factory (`dpf`) usata per creare un partecipant, passando alla funzione i valori argc e argv ricevuti dal main (vediamo dopo a cosa servono).

Dopo di ché si crea il partecipant mediante il metodo `create_participant` richiamato appunto su `dpf`.
Il metodo sopra citato prende 4 parametri, di cui quello che ci interessa principalmente è il primo, ovvero il domainID a cui questo partecipant è iscritto. Ciò sta ad indicare che esso potrà leggere e scrivere messaggi da/per utenti associati a questo ID.

Infine vi è un controllo di errore per vedere che tutto abbia funzionato. Se partecipant contiene 0 allora è tutto ok.

A questo punto entra in scena la struttura stilata nel file .idl, in quanto viene qui richiamata.

```
Esempio::ProvaTypeSupport_var ts =
  new Esempio::ProvaTypeSupportImpl;

if (ts->register_type(participant, "")
    != DDS::RETCODE_OK) {

  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" register_type failed!\n")),
  -1);
}
```

Si dichiara una variabile del tipo descritto nel file idl, mediante il tipo `Esempio::ProvaTypeSupport_var` e la si alloca nell'heap mediante la `new` chiamata sul tipo `Esempio::ProvaTypeSupportImpl`.

A questo punto mediante il metodo `register_type` richiamato sull'oggetto appena creato, passando come primo parametro la variabile `partecipant` creata prima, si associa quel tipo di dato al partecipant corrispondente.
Anche in questo caso la funizone restituisce un codice, per indicare la presenza o meno di errori.

Adesso siamo arrivati al punto di creare il topic.
Questa operazione viene effettuata attraverso il seguente segmento di codice:

```
CORBA::String_var type_name = ts->get_type_name();

DDS::Topic_var topic = participant->create_topic(
"Topic di esempio",type_name, TOPIC_QOS_DEFAULT,0,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!topic) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" create_topic failed!\n")),-1);

}
```

Come prima cosa si chiama il metodo `get_type_name` sulla variabile `ts` dichiarata precedentemente.

A questo punto si crea il topic a partire dal partecipant, mediante la funzione `create_topic(…)`,passandogli come parametro anche il type_name, che deriva a sua volta dal ProvaType. Quindi ecco che in questo modo si riescono a legare topic e ProvaType. Dunque, come avevamo già detto nella descrizione generale di un ambiente DDS, il tipo del messaggio (ProvaType) è uno dei 2 identificativi del Topic, mentre l'altro è il nome assegnato, che è il primo parametro passato.

Anche qui si fa il solito controllo di errore andando a vedere il valore restituito dalla `create_topic(…)`.

Ora creiamo il publisher, a partire sempre dal partecipant creato all'inizio:

```
DDS::Publisher_var publisher =
participant->create_publisher(PUBLISHER_QOS_DEFAULT,0,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!publisher)
{
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:  %l: main() -")
  ACE_TEXT(" create_publisher failed!\n")),-1);
}
```

In questo caso i vari parametri sono considerati standard, per il nostro scopo esemplificativo, dunque non spendo parole per descriverli.

Successivamente, a partire dal publisher, creiamo il DataWriter, che è ciò che si occupa di effettuare la scrittura vera e propria.

```
DDS::DataWriter_var writer =
publisher->create_datawriter(topic,
DATAWRITER_QOS_DEFAULT,0,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!writer) {
   ACE_ERROR_RETURN((LM_ERROR,
   ACE_TEXT("ERROR: %N:%l: main() -")
   ACE_TEXT(" create_datawriter failed!\n")),-1);

}



Esempio::ProvaDataWriter_var prova_writer =
    Esempio::ProvaDataWriter::_narrow(writer);
```

```
if (!message_writer) {
 ACE_ERROR_RETURN((LM_ERROR,
 ACE_TEXT("ERROR: %N:%l: main() -")
 ACE_TEXT(" _narrow failed!\n")),-1);
}
```

Si chiama il metodo `create_datawriter(…)` passandogli come parametro il topic creato in precedenza; ottenuto il DataWriter lo si passa come parametro al metodo statico `_narrow` della classe `Esempio::ProvaDataWriter`. Questo crea una variabile di tipo `Esempio::ProvaDataWriter_var`, che specifica al writer il tipo di messaggio che può utilizzare.
Si noti che qui vengono richiamati dei metodi e dei tipi che sono generati automaticamente dall'interprete IDL prima che vengano compilati i sorgenti veri e propri.

A questo punto il file contiene una sequenza di istruzioni volte a bloccare il programma fin tanto che un subscriber non si è ancora connesso.
Io le salterò di proposito, supponendo che al momento dell'inizio dell'esecuzione del processo publisher, il processo subscriber sia già inizializzato ed in esecuzione.

Ora creiamo una istanza concreta del messaggio da inviare coerentemente con quanto dichiarato nella struct Prova, contenuta nel file Esempio.idl .

```
 Esempio::Prova prova;
    prova.id  = 20;
    prova.str = "Testo di prova";
```

Mediante un ciclo for, inviamo 10 messaggi di questo tipo.

```
for (int i = 0; i < 10; ++i) {

DDS::ReturnCode_t error = prova_writer->write(
prova, DDS::HANDLE_NIL);

prova.id++;

  if (error != DDS::RETCODE_OK) {
    ACE_ERROR((LM_ERROR,
    ACE_TEXT("ERROR: %N:%l: main() -")
    ACE_TEXT(" write returned %d!\n"),error));
    }
}
```

Ad ogni iterazione viene inviato un messaggio diverso, in quanto il campo `id`(la chiave) viene incrementato di 1.

Un ulteriore parte di codice, quella relativa alla verifica all'acknowledgement da parte del subscriber, viene volontariamente saltata, in quanto appesantirebbe la descrizione della sequenza di operazioni da fare.

```
participant->delete_contained_entities();
dpf->delete_participant(participant);

TheServiceParticipant->shutdown();

} catch (const CORBA::Exception& e) {
  e._tao_print_exception("Exception caught in main():");
  return -1;
}
return 0;

}
```

Si arriva alla conclusione, in cui viene liberata la memoria, come sempre a partire dagli oggetti allocati più internamente, fino a quelli più esterni.

Successivamente si chiude il ServicePartecipant ed infine si cattura una generica eccezione di tipo `CORBA::Exception`, ovvero il tipo di eccezione lanciato per qualunque errore inerente a funzioni OpenDDS.
Questa `catch` è riferita al blocco `try` aperto ad inizio codice.

In questo modo è conclusa la descrizione della componente publisher del progetto considerato.

Adesso veniamo alla componente subscriber, che, come già visto nel file Esempio.mpc, si compone di 2 file, Subscriber.cpp e DataReaderListenerImpl.cpp.

Li analiziamo a partire dal primo.

```
try {

DDS::DomainParticipantFactory_var dpf =
  TheParticipantFactoryWithArgs(argc, argv);

DDS::DomainParticipant_var participant =
dpf->create_participant(
  42,PARTICIPANT_QOS_DEFAULT,0,
  OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!participant) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR:  %N:%l:  main()  -")
  ACE_TEXT(" create_participant failed!\n")), 1);
}
```

Analogamente a quanto effettuato sul Publisher, in questa prima sequenza di operazioni si effettua la creazione in sequenza di DomainPartecipantFactory e partecipant, con controllo di errore corrispondente.

In un secondo momento viene istanziata la struttura dati contenente il tipo di messaggi da ricevere a cui, in seguito, viene assegnato un topic.

```
Esempio::ProvaTypeSupport_var ts =
      new Esempio::ProvaTypeSupportImpl;
```

```
if (ts->register_type(participant, "") !=
    DDS::RETCODE_OK) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" register_type failed!\n")),1);

  }
```

Qui si crea la variabile (ts) che conterrà il messaggio da ricevere

```
CORBA::String_var type_name = ts->get_type_name();

DDS::Topic_var topic =
participant→create_topic(
"Topic di esempio",type_name,
TOPIC_QOS_DEFAULT,0,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!topic) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" create_topic failed!\n")),1);

}
```

e la si associa al topic appena creato (utilizzando come collegamento il campo type_name della struttura ts creata in precedenza).

```
DDS::Subscriber_var subscriber =
participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,0,
   OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

```
if (!subscriber) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" create_subscriber failed!\n")),1);

}
```

In questa sezione viene creato il subscriber a partire dal partecipant creato all’inizio.

A questo punto cerchiamo di approfondire maggiormente la parte seguente, in quanto si differenzia notevolmente da quanto visto lato Publisher.

Da questo momento in poi infatti entrerà in gioco il secondo file sorgente elencato all’inizio della descrizione della componente Subscriber.

Questo secondo file contiene il codice con cui gestire il contenuto del messaggio proveniente da un Publisher, e verrà mostrato successivamente al file in analisi.

Vediamo ora come continua la creazione del datareader:

```
DDS::DataReaderListener_var listener(
new DataReaderListenerImpl);

DDS::DataReader_var reader =
subscriber->create_datareader(
topic,
DATAREADER_QOS_DEFAULT,
listener,
OpenDDS::DCPS::DEFAULT_STATUS_MASK);

if (!reader) {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" create_datareader failed!\n")),1);

}
```

In questa parte vengono creati il listener (ovvero la componente che riceve i messaggi) e il datareader vero e proprio.

A questo punto si blocca il reader fino alla ricezione di un nuovo messaggio.

```
DDS::StatusCondition_var  condition  =
  reader->get_statuscondition();


condition->set_enabled_statuses(
DDS::SUBSCRIPTION_MATCHED_STATUS);

DDS::WaitSet_var ws = new DDS::WaitSet;


ws->attach_condition(condition);
```

```
while (true) {
  DDS::SubscriptionMatchedStatus matches;

if (
reader->get_subscription_matched_status(matches)
!= DDS::RETCODE_OK) {

  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -"
  ACE_TEXT("get_subscription_matched_status failed\n")),
  1);
  }

if(matches.current_count==0&&matches.total_count>0){
  break;
  }

  DDS::ConditionSeq conditions;
  DDS::Duration_t timeout = { 10, 0 };

if(ws->wait(conditions,timeout)!= DDS::RETCODE_OK)
  {
  ACE_ERROR_RETURN((LM_ERROR,
  ACE_TEXT("ERROR: %N:%l: main() -")
  ACE_TEXT(" wait failed!\n")),1);

  }

}

ws->detach_condition(condition);
```

Descriviamo ora un momento questa fase.

Il ciclo while che contiene il codice per gestire la ricezione dei messagi è preceduto da una condizione di attesa che permette un ingresso alla volta nella gestione dei messaggi ricevuti. Questo crea una sorta di coda di attesa.

Una volta entrati nel ciclo entra in gioco la variabile `matches` che viene modificata dal metodo `get_subscription_matched_status` richiamato sul reader.

Questa variabile ci serve per capire se la lettura dei messaggi è terminata o meno.
Il modo per farlo è un duplice confronto posto in AND logico.

```
if(matches.current_count==0 && matches.total_count>0)
```

In questo modo riesco a vedere sia se ho già iniziato a leggere qualcosa (`matches.total_count > 0`) sia se l'attuale lettura ha registrato dati nuovi in ingresso
(`matches.current_count == 0`).

Poiché appena dopo questo controllo vi è l'attesa di n secondi (10) sul WaitSet creato prima di entrare nel ciclo, una volta scaduto il timeout, viene verificata la condizione descritta poco sopra.

Se non ci sono dati nuovi, ma avevo già ricevuto qualcosa prima, allora suppongo la ricezione dati terminata.

```
participant->delete_contained_entities();
dpf->delete_participant(participant);
```

```
TheServiceParticipant->shutdown();

} catch (const CORBA::Exception& e) {
    e._tao_print_exception("Exception caught in main():");
  return 1;
}
```

A questo punto si fa una sorta di free di tutte le componenti e si chiude il Subscriber.

Adesso analizziamo il file DataReaderListenerImpl.cpp, che contiene i metodi che devono essere invocati nei vari casi. Questo include il file DataReaderListenerImpl.h, che contiene la dichiarazione dei suddetti metodi, e di cui ne viene riportato il contenuto qua sotto.

```
class DataReaderListenerImpl: public virtual
OpenDDS::DCPS::LocalObject<DDS::DataReaderListener> {
public:

virtual void on_requested_deadline_missed(
DDS::DataReader_ptr reader,
const DDS::RequestedDeadlineMissedStatus& status);

virtual void on_requested_incompatible_qos(
DDS::DataReader_ptr reader,
const DDS::RequestedIncompatibleQosStatus&status);

virtual void on_sample_rejected(
DDS::DataReader_ptr reader,
const DDS::SampleRejectedStatus& status);

virtual void on_liveliness_changed(
DDS::DataReader_ptr reader,
const DDS::LivelinessChangedStatus& status);

virtual void on_data_available(
DDS::DataReader_ptr reader);

virtual void on_subscription_matched(
DDS::DataReader_ptr reader,
const DDS::SubscriptionMatchedStatus& status);

virtual void on_sample_lost(
DDS::DataReader_ptr reader,
const DDS::SampleLostStatus& status);

};
```

Di questi metodi, a titolo esemplificativo, viene implementato in maniera attiva soltanto il `on_data_available` che contiene appunto le operazioni da effettuare alla ricezione di un dato.

Nel file .cpp corrispondente troviamo infatti la sua implementazione:

```
Esempio::ProvaDataReader_var reader_i =
    Esempio::ProvaDataReader::_narrow(reader);

if (!reader_i) {
   ACE_ERROR((LM_ERROR,
   ACE_TEXT("ERROR: %N:%l: on_data_available() -")
   ACE_TEXT(" _narrow failed!\n")));
   ACE_OS::exit(1);
}
```

Innanzitutto viene istanziato un puntatore al reader passato come parametro.

```
Esempio::Prova prova;
DDS::SampleInfo info;

DDS::ReturnCode_t error =
reader_i->take_next_sample(prova, info);
```

Viene creata la variabile che conterrà il messaggio e viene catturato il messaggio vero e proprio.

```
if (error == DDS::RETCODE_OK) {
  std::cout  <<  "SampleInfo.sample_rank  =  "  <<
  info.sample_rank << std::endl;

  std::cout << "SampleInfo.instance_state = " <<
  info.instance_state << std::endl;

  if (info.valid_data) {
    std::cout << " id   = " << prova.id <<
    std::endl<<"testo="<<prova.str.in()<<
    std::endl;
    }

  } else {
   ACE_ERROR((LM_ERROR,
   ACE_TEXT("ERROR: %N:%l: on_data_available() -")
   ACE_TEXT(" take_next_sample failed!\n")));
  }
```

Se la lettura dei dati è andata a buon fine si procede con l’acquisizione e, nel caso del nostro esempio, alla stampa di questi a video.

Con questo si chiude sostanzialmente il programma di esempio di un possibile utilizzo di OpenDDS.

Una volta conclusosi il discorso sui sorgenti, vediamo un momento come eseguire il programma.

3.4. Modalità di esecuzione

In questa ultima sezione verranno descritte le due principali modalità di esecuzione di OpenDDS dal punto di vista della configurazione di rete che si intende utilizzare.
Queste modalità sono impartite al DDS mediante un file di configurazione il cui nome viene passato via linea di comando sia al processo Publisher che Subscriber.

La prima è la così detta modalità centralizzata, ovvero nella quale c'è un processo che funge da gestore dello scambio di messaggi nella rete (broker).
Nel nostro esempio supponiamo che il publisher abbia indirizzo ip 192.168.1.20, il subscriber 192.168.1.30 e il broker 192.168.1.10 con processo di discovery attivo sulla porta 12345.

Il file di configurazione per il publisher è il seguente:

```
#config_file_p.ini:

[common]

DCPSInfoRepo=corbaloc::192.168.1.10:12345/
DCPSInfoRepo

DomainId=1

DCPSGlobalTransportConfig=config1

[config/config1]

transports=tcp1

[transport/tcp1]

transport_type=tcp

local_address=192.168.1.20

[domain/DomainId]

DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]

RepositoryIor=192.168.1.10:12345
```

mentre per il subscriber il file è questo:

```
#config_file_s.ini:

[common]

DCPSInfoRepo=corbaloc::192.168.1.10:12345/
DCPSInfoRepo

DomainId=1

DCPSGlobalTransportConfig=config1

[config/config1]

transports=tcp1

[transport/tcp1]

transport_type=tcp

local_address=192.168.1.30

[domain/DomainId]

DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]

RepositoryIor=192.168.1.10:12345
```

L’esecuzione del programma, in questa modalità, avviene lanciando i seguenti comandi in sequenza:

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBListenEndpoints iiop://192.168.1.10:12345 -d 1

./subscriber -DCPSConfigFile ./config_file_s.ini

./publisher -DCPSConfigFile ./config_file_p.ini
```

In questo caso nel file di configurazione sono indicati l’indirizzo ip dell’interfaccia su cui agisce DDS per entrambe le componenti e l’ip e la porta del broker.

Il broker viene eseguito per primo. Successivamente si mette in ascolto il subscriber e, infine, il publisher inizia ad inviare i dati.

Una seconda modalità di esecuzione è quella senza l'utilizzo di un broker, nella quale le componenti comunicano in modalità multicast, mediante protocollo RTPS (Real-Time Publish-Subscribe).

In questo caso gli unici 2 processi ad essere eseguiti sono il subscriber e il publisher.

Vengono qui riportati i file di configurazione in questo caso:

```
#rtps_file_p.ini
[common]

DomainId=1

DCPSDefaultDiscovery=config2

DCPSGlobalTransportConfig=config1

[domain/DomainId]

DiscoveryConfig=config2

[rtps_discovery/config2]

SedpMulticast=0

SedpLocalAddress=192.168.1.20:13556

SpdpLocalAddress=192.168.1.20

SpdpSendAddrs=239.192.3.1:7900

[config/config1]

transports=transport1

[transport/transport1]

transport_type=multicast

group_address=239.192.3.1:7900

local_address=192.168.1.20
```

```
#rtps_file_s.ini
[common]

DomainId=1

DCPSDefaultDiscovery=config2

DCPSGlobalTransportConfig=config1

[domain/DomainId]

DiscoveryConfig=config2

[rtps_discovery/config2]

SedpMulticast=0

SedpLocalAddress=192.168.1.30:13656

SpdpLocalAddress=192.168.1.30

SpdpSendAddrs=239.192.3.1:7900

[config/config1]

transports=transport1

[transport/transport1]

transport_type=multicast

group_address=239.192.3.1:7900

local_address=192.168.1.30
```

In questo caso nei file di configurazione indichiamo l’indirizzo ip multicast a cui inviare i messaggi e la porta che desideriamo utilizzi il protocollo RTPS.

In questo modo ho illustrato anche le due principali modalità di configurazione di un sistema di apparati comunicanti mediante OpenDDS.

3.5. Emulazione comunicazione intraveicolare

In questo paragrafo descriveremo una emulazione effettuata su macchine virtuali in VirtualBox rappresentanti 3 dispositivi che inviano notifiche ad una centralina di bordo.

Nel caso specifico abbiamo i seguenti dispositivi:
-Brakes_PSM, per il controllo di attivazione/disattivazione dell'impianto frenante;
-Power_PSM, per la gestione dell'energia e la modifica di modalità di caricamento delle batterie;
-Video_PSM, per il controllo di un dispositivo di acquisizione multimediale (Videocamera), in cui verrà emulata la capacità da parte di questo di inviare fotogrammi compressi (uno ogni 10 secondi).

I 3 dispositivi e la centralina sono connessi ad una rete locale (192.168.1.0/24) e comunicano in modalità multicast, previo discovery reciproco mediante protocollo rtps_udp (Real Time Publish Subscribe over UDP), mediante il quale ogni publisher invia i messaggi ad un indirizzo ip multicast e ogni subscriber in ascolto su tale indirizzo è in grado di riceverli.

I topic considerati sono stati:

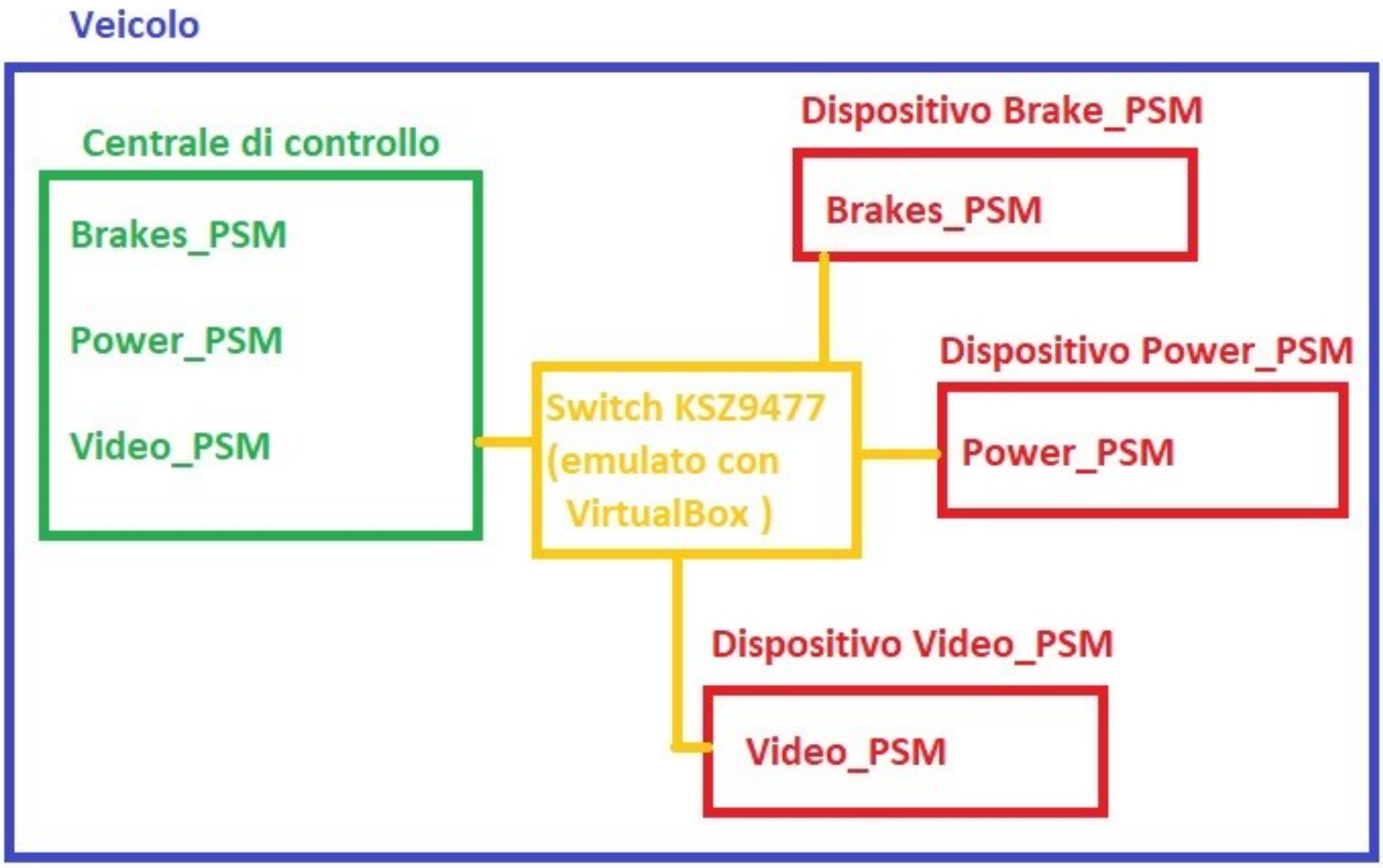
- C_Braking_System_commandEngineBrake (controllo attivazione/disattivazione freni), dal file Brakes_PSM.idl

- C_Charger (controllo cambio di modalità di caricamento batterie), dal file Power_PSM.idl

- C_DataBlock (contenitore di un frame compresso inviato da una videocamera), dal file Video_PSM.idl

La configurazione con cui è stata impostata la demo è quella di avere 4 macchine virtuali aperte (ognuna rappresentante uno specifico dispositivo), in cui vengono lanciati i subscriber (3 sulla vm che rappresenta la centralina) e successivamente i publisher (uno per ognuna delle 3 vm rimanenti).

Due dei topic considerati (quelli dai file Brakes_PSM e Power_PSM) sono implementati con messaggi di tipo testuale, che indicano la variazione del comportamento del dispositivo a cui si riferiscono.
Il terzo, Video_PSM, oltre a notificare testualmente il fatto che è pronto un nuovo frame, si occupa anche del suo trasferimento.



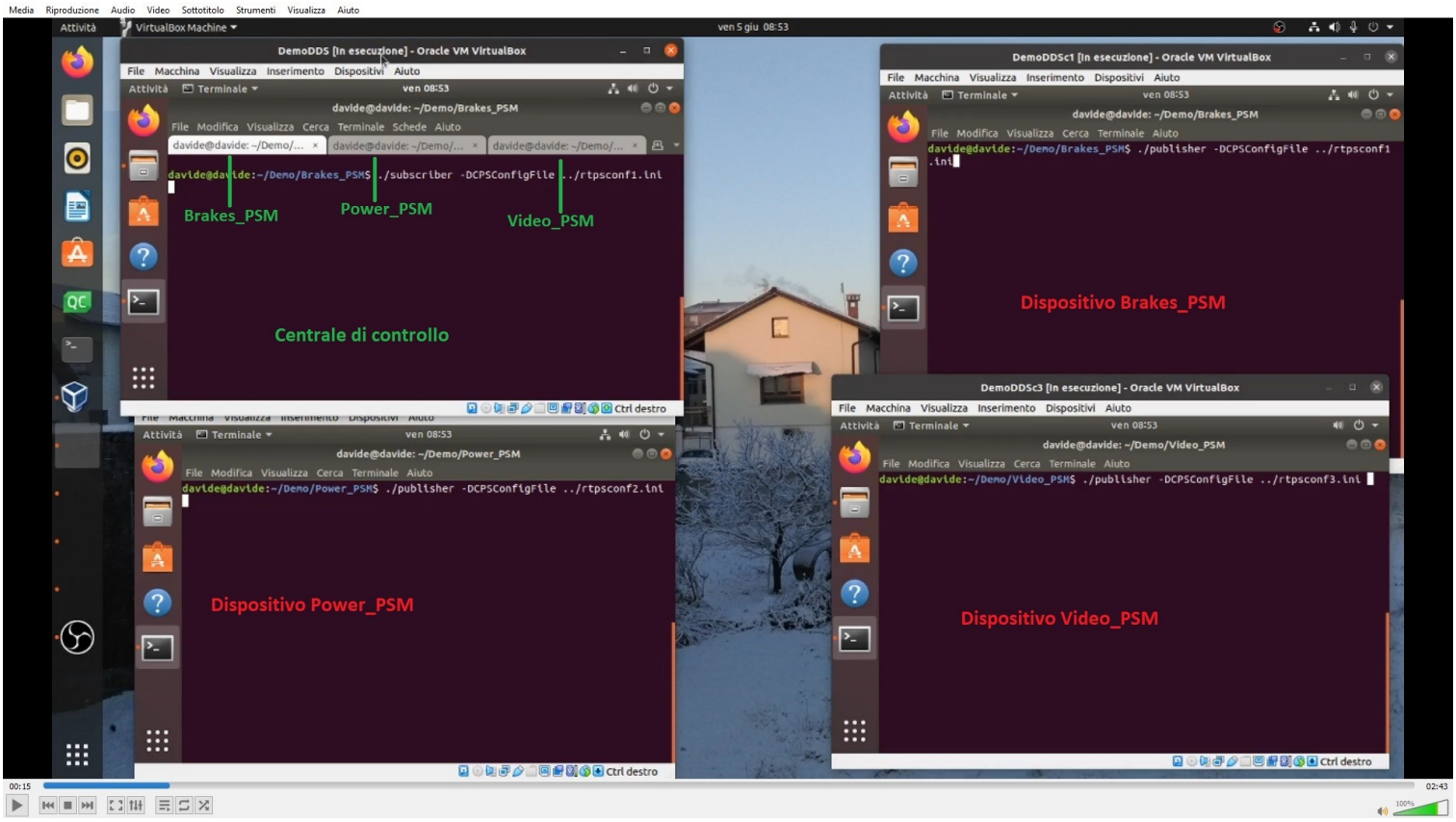
Tutti i file sorgenti e di configurazione si possono trovare qui:
https://drive.google.com/drive/folders/1dRnR70rSNdOR2WI_3-ePqO2HyNFHSp20

Esiste inoltre un video-demo del funzionamento del sistema. Il link per trovarlo è il seguente:
https://drive.google.com/file/d/1FdCJ8jT80B7Y_sRLwBowbY6Kru4RfSzN/view

Per la compilazione dei sorgenti rimando all'appendice 3, in cui descrivo la compilazione di un generico semplice progetto OpenDDS.

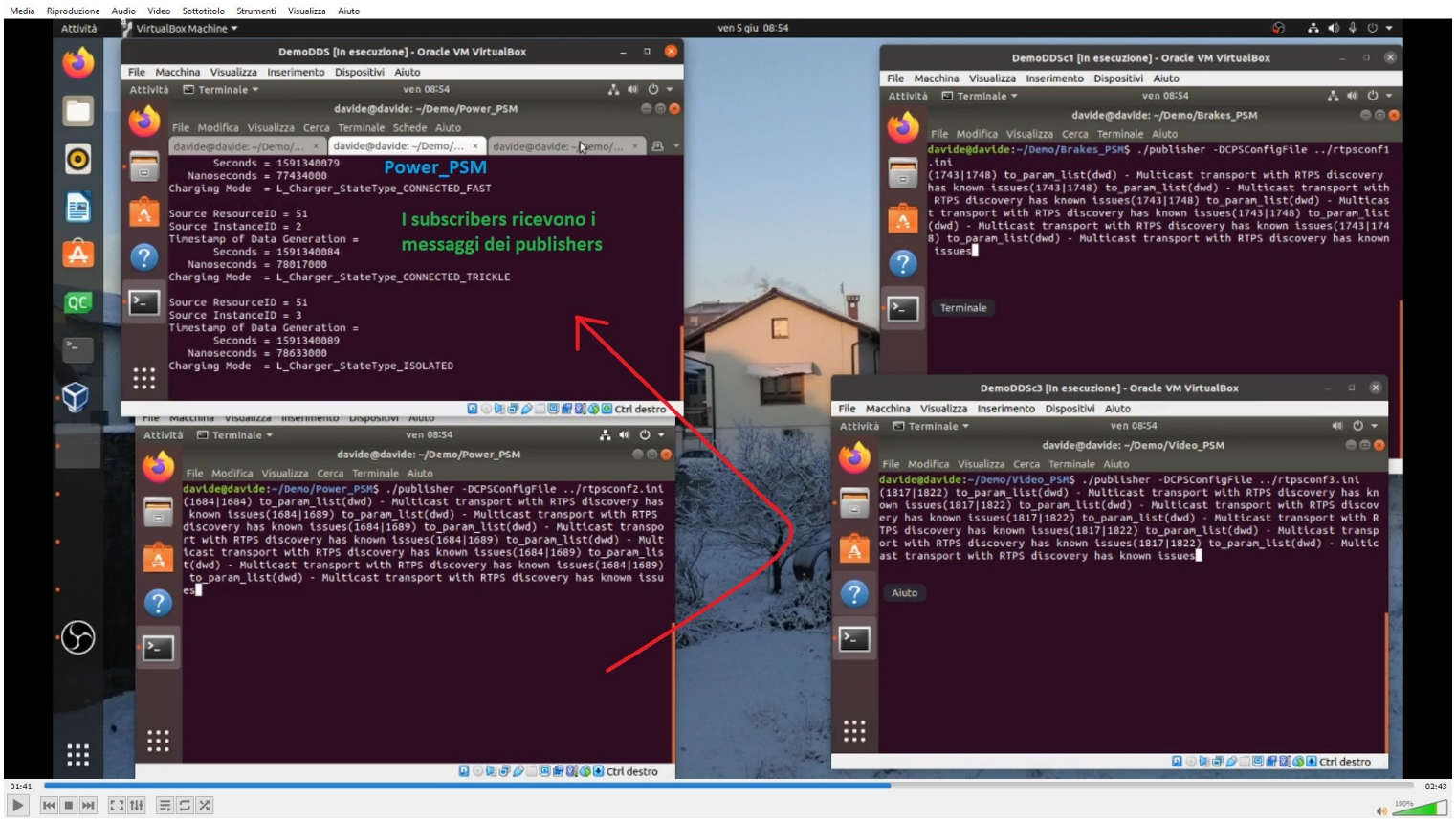
Di seguito vengono riportati delle immagini tratte dalla video-demo linkata nella pagina precedente, in cui si evidenziano i punti salienti.

1) Inizializzazione del sistema



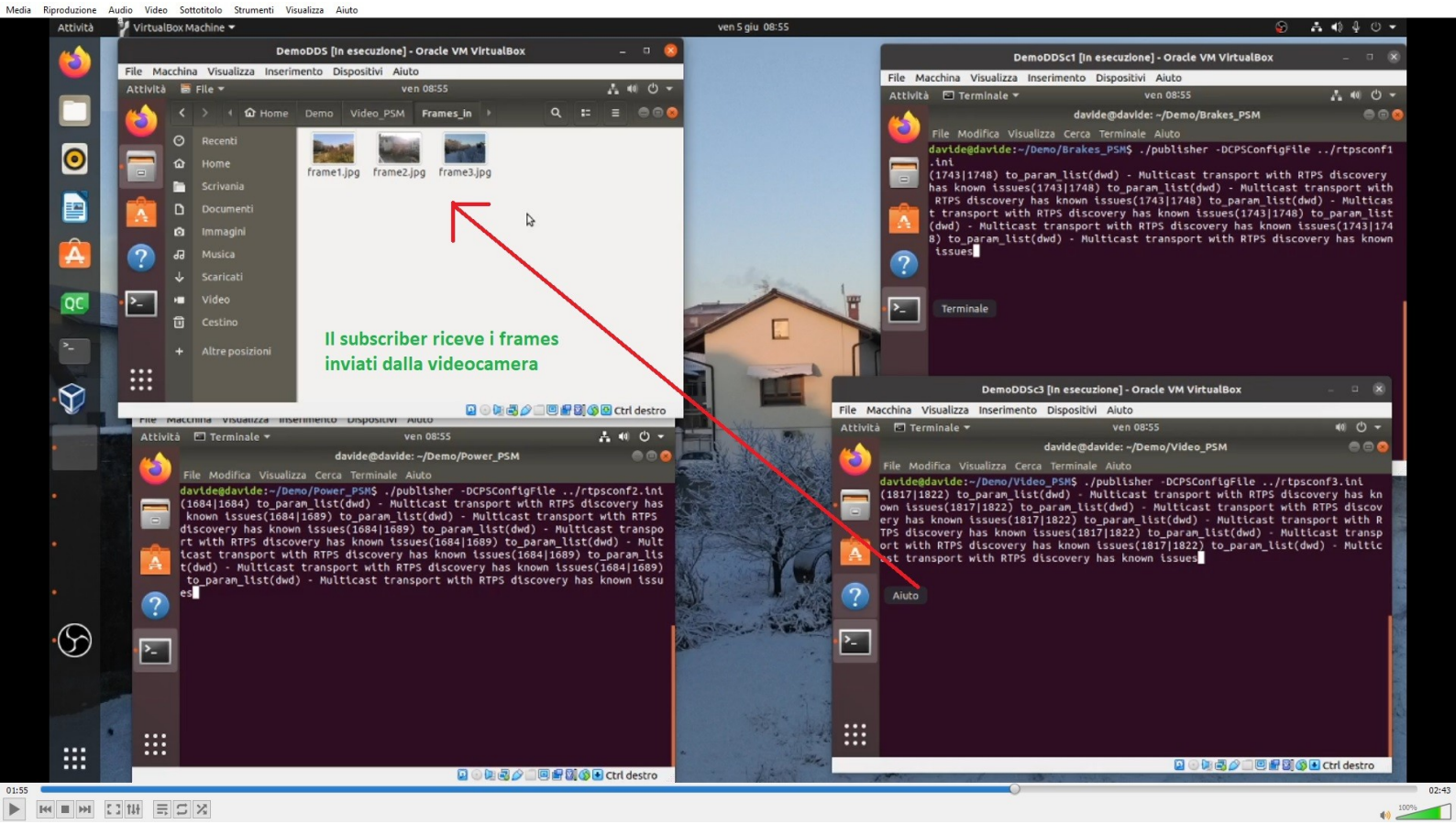
In alto a destra troviamo la centrale di controllo, che attiva 3 processi subscriber, uno per ogni dispositivo connesso
Le altre 3 VM invece emulano un dispositivo a testa, attivando su ognuno un processo publisher.

2) Avvio dei processi subscriber e publisher



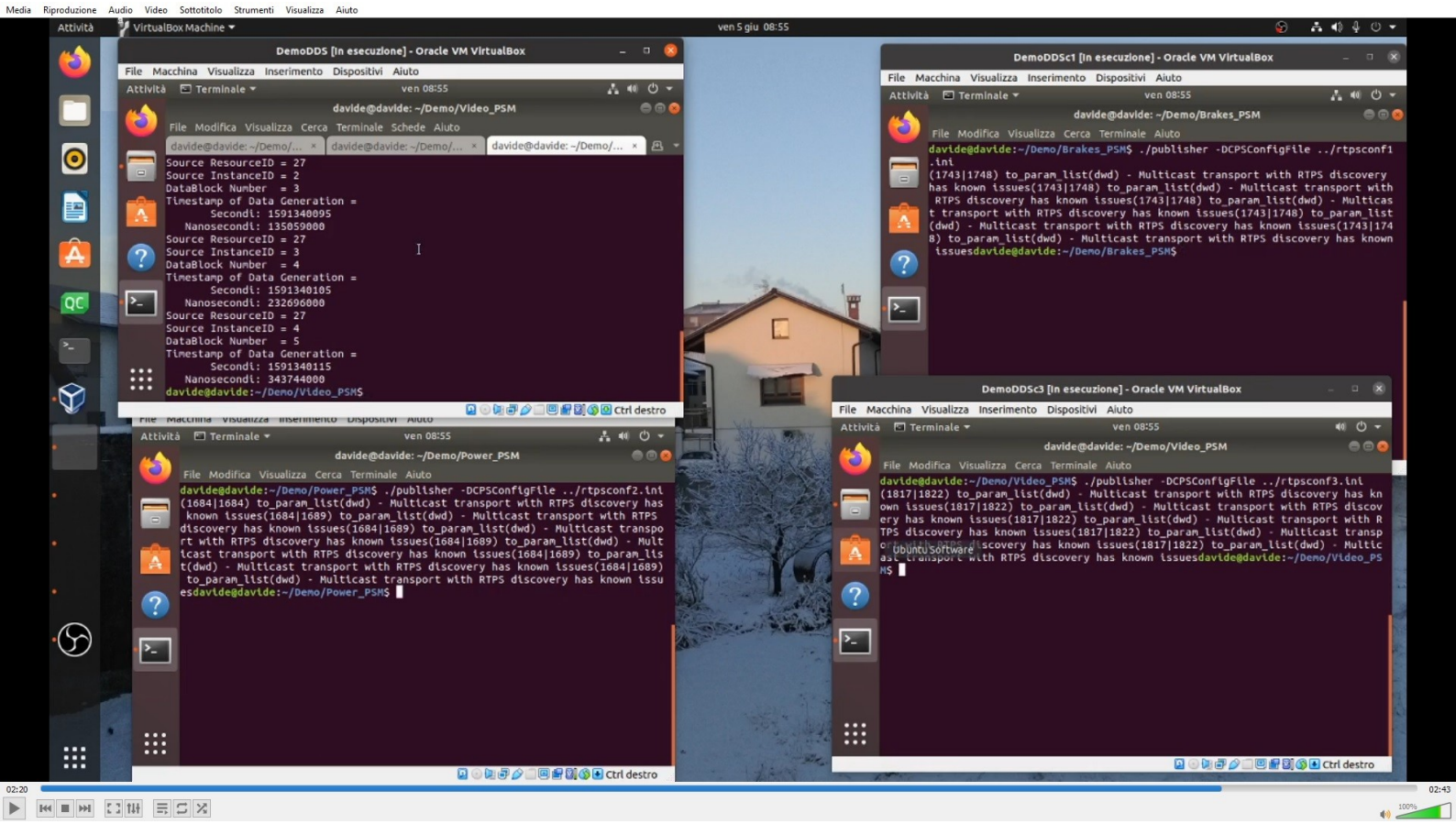
Vengono avviati i 3 processi subscriber e, successivamente, i 3 publisher.
Nell'immagine si può notare che il subscriber relativo a Power_PSM sta ricevendo (e stampando a video) i dati inviatigli dal publisher relativo

Qui sotto troviamo una schermata relativa al comportamento della componente Video_PSM del sistema



Notiamo che il subscriber relativo a Video_PSM sta ricevendo i frames inviatigli dal publisher corrispondente e li memorizza in una sua cartella locale (Frames_in)

3) Chiusura della comunicazione



Una volta che i publisher hanno finito di inviare i loro dati, i subscriber attendono un tempo prestabilito nel programma, dopodiché chiudono la connessione.

4. Comandi utilizzati

In questa sezione vengono presentati i comandi utilizzati per la configurazione iniziale degli strumenti.

Nella Appendice1 è mostrata l'inizializzazione degli strumenti per la compilazione del kernel da installare su i.mx, compresi i driver dello switch e il cross-compilatore per arm.

Nella Appendice2, invece, si parlerà della configurazione dell'ambiente OpenDDS, sia in merito alla sua installazione (per processori x86 e arm) sia per la compilazione di programmi su questo ambiente per entrambe le piattaforme.

N.B.: E' importante evidenziare il fatto che i comandi espressi sono validi per la verisone di Ubuntu 14.04_LTE, e che i programmi scaricati sono compatibili con la verisone a 64bit.

Su altre distribuzioni Linux ci potrebbero essere delle variazioni poco significative.

4.1. Appendice1 - configurazione i.mx

Di seguito si riportano i comandi con una breve descrizione di questi (i file da scaricare sono indicati nella descrizione sopra).

Configurazione Cross-Compilatore:

Dalla cartella in cui si è scaricato il cross-compilatore digitare il comando:

```
sudo tar -xzvf iMX6_arm-fsl-linaro-gnueabi-4.9.1.tar.gz /opt/
```

verrà richiesta la password per accesso alle funzionalità di super-user.

Ora occorre settare la variabile di ambiente `CROSS_COMPILE`, di modo che ogni qual volta viene chiamato un comando `gcc` o equivalente, il sistema capisca che deve andare a cercare nella directory indicata nella suddetta variabile e non in `/usr/bin`, dove si trovano di solito i file eseguibili, tra cui appunto quelli della famiglia `gcc`.
Per far ciò si chiudono tutti i terminali aperti, si apre un terminale “nuovo” e si digita il comando:

```
CROSS_COMPILE=/opt/yocto/fsl-release-bsp/build_seco/tmp/sysroots/x86-64-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-
```

Ora il Cross-Compilatore è inizializzato e pronto all’uso. Si ricorda che tale variabile resta valida fino alla chiusura del sistema. Perciò è necessario reimpostarla al riavvio.

Configurazione kernel con drivers:

Dalla cartella in cui si è estratto il kernel Linux aprire un terminale e digitare il comando:

```
tar -xzvf linux-3-14-28-secoboards-imx6-rel-master.tar.gz
```

La directory creata dall'estrazione verrà chiamata *[root]*.
Se il file viene scaricato nella cartella *Scaricati* di Ubuntu, allora [root] = home/*nome_utente*/Scaricati/linux-3-14-28-secoboards-imx6-rel-master/

Adesso bisogna estrarre i file contenenti i drivers dello swtich.
Dalla cartella in cui si è scaricato il file .zip dei drivers, digitare il comando:

```
unzip EVB-KSZ9477-master.zip -d EVB-KSZ9477-master
```

Questo crea una cartella con i files estratti.
Poi si inseriscono i files dei drivers nel sorgente del kernel estratto sopra.

```
cd EVB-KSZ9477-master/KSZ/linux-drivers/ksz9897/linux-3.14

cp -r * [root]
```

A questo punto la configurazione iniziale è completata e si può proseguire con la descrizione della compilazione del kernel, nella parte di descrizione presente sopra.

4.2. Appendice2 – configurazione OpenDDS

Qui si riportano i passaggi preliminari la compilazione di OpenDDS.

Estrarre il file scaricato dal link della descrizione sopra con il comando

```
tar -xzvf OpenDDS-3.13.2.tar.gz
```

Tale operazione creerà una directory di nome `OpenDDS-3.13.2` .

Possiamo, e per il proseguo delle operazioni è consigliato, rimuovere il file compresso scaricato con il comando:

```
rm OpenDDS-3.13.2.tar.gz
```

Ora si crea una copia della cartella sopra estratta con il comando

```
cp -r OpenDDS-3.13.2 OpenDDS-3.13.2-x86
```

Per comodità questa cartella appena creata verrà chiamata `[root-dds-x86]`.

La cartella `OpenDDS-3.13.2`, invece la chiameremo `[root-dds-arm]`.

Questi sono i passi preliminari che ci consentono di avere 2 istanze identiche di sorgente OpenDDS, pronte alla compilazione sui due diversi ambienti.

4.3. Appendice3 - compilazione progetto OpenDDS

Un progetto OpenDDS minimale, ovvero quello che andremo a compilare, è costituito fondamentalmente da un singolo workspace, definito all'interno di un file .mwc.

Questo file contiene un elenco di File .mpc, di cui discusso nella parte riguardante l'analisi di un programma OpenDDS.

Nell'esempio di prima il file si chiamerà *Esempio.mwc* e conterrà il seguente codice:

```
workspace {

  Esempio.mpc

}
```

Per compilare il progetto, dopo aver settato le variabili di ambiente con il seguente comando:

```
source <DDS_ROOT>/setenv.sh
```

dove DDS_ROOT è la cartella principale in cui si trova OpenDDS,

si eseguono i seguenti passi:

```
$ACE_ROOT/bin/mwc.pl -type gnuace Esempio.mwc

make
```

Se non ci sono errori nei files il progetto viene compilato e vengono generati gli eseguibili `publisher` e `subscriber`.

5. Bibliografia

Bus SPI:
https://www.vincenzov.net/tutorial/RaspberryPi/spi-c.htm

Switch 9477 :
http://ww1.microchip.com/downloads/en/DeviceDoc/KSZ9477S-Data-Sheet-DS00002392C.pdf

U-Boot:
https://developer.toradex.com/software-resources/arm-family/linux/features/uboot

OpenDDS:
https://opendds.org/about/

DDSMulticastMode:
https://objectcomputing.com/resources/publications/mnb/using-reliable-multicast-for-data-distribution-with-opendds