

Guardians of the CAN bus – come usare il CAN bus in C e C++ – pt.4

Stakar Ogord: Uniformi della OrgoCorp. Dovrete indossarle per muovervi nell'Orgosfera senza attirare l'attenzione.

Drax: Questo non è il mio colore.

Stakar Ogord: Cosa hai detto?

Drax: Stona con i miei occhi.

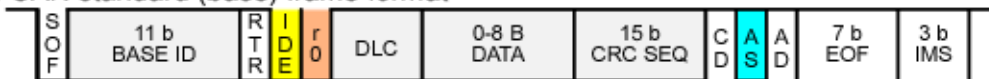
Ed eccoci alla quarta (e ultima!) parte di **Guardians of the CAN bus**, un ciclo di articoli ispirati al bellissimo **Guardiani della Galassia Vol.3** (ripassatevi **la prima**, **la seconda** e **la terza** parte, please...). Nel dialogo qui sopra c'è, di nuovo, tutta la vena comica del film, che viaggia come al solito in parallelo alla parte seria. Mi ripeterò: è po' come il **CAN bus**, che è così ben studiato ed efficace che rende divertente scrivere Software che lo usa, salvo quando devi risolvere un problema molto complicato e allora entri nella fase seria e drammatica... In questa quarta parte scopriremo gli ultimi segreti che ci mancano per diventare dei veri e propri **Guardians of the CAN bus**!



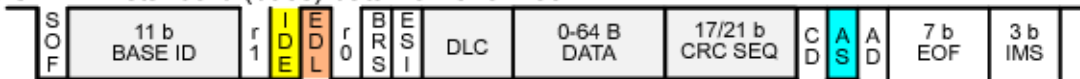
...ma anche il CAN bus stona coi tuoi occhi?...

Come ho fatto notare nella chiusura **dell'ultimo articolo** oramai ci manca solo il **CAN FD** per ultimare il nostro viaggio nei segreti del CAN bus. E allora veniamo subito al dunque: cosa è il CAN FD? Il CAN FD (CAN Flexible Data Rate) è un'espansione del CAN bus "classico" (Classical CAN) ed è definito dalla norma ISO 11898-1:2015. Il CAN FD facilita non solo l'invio di volumi di dati più grandi, visto che il limite di 8 byte del CAN classico è stato aumentato a 64 byte. Inoltre, offre la possibilità di una velocità di trasmissione più elevata nella fase dei dati, raggiungendo agevolmente i 5 Mbit/s. Anche del CAN FD esiste una versione extended, quindi, alla fin fine, abbiamo quattro tipi di frame, quelli mostrati in questa figura riassuntiva:

a) CAN standard (base) frame format



b) CAN FD standard (base) data frame format



c) CAN extended frame format



d) CAN FD extended data frame format



Si può commentare che, con questa versione FD, il CAN bus si snatura un poco, passando da semplice e affidabilissimo “*bus di campo*” a un rango superiore, quasi quello dei protocolli di rete sofisticati (tipo Ethernet) che permettono di trasferire notevoli moli di dati: comunque, anche se “*snaturato*”, è assolutamente pronto all’uso e quando è il caso si può/deve usare, perché offre delle grandi prestazioni.

Ok, con ben quattro tipi disponibili sembrerebbe abbastanza complicato scrivere Software CAN “*compatibile*” (o *quasi*) passando da un tipo all’altro senza fare troppi cambi... Ma per questo c’è il nostro amico **SocketCAN**! Come abbiamo già visto **nell’articolo precedente**, passare da CAN classic a CAN extended è decisamente semplice, e sono felice di comunicarvi che, grazie alla flessibilità di SocketCAN, anche il CAN FD si usa in maniera molto semplificata. Vediamo come: non so se ricordate, ma nel **primo articolo** del ciclo avevo mostrato come è descritto (nell’*header can.h*) un *frame* CAN di SocketCAN, ve lo ripropongo:

```
/** * struct can_frame - basic CAN frame structure * @can_id:
the CAN ID of the frame and CAN_*_FLAG flags, see above. *
@can_dlc: the data length field of the CAN frame * @data: the
CAN frame payload. */
struct can_frame { canid_t can_id; /* 32 bit CAN_ID +
EFF/RTR/ERR flags */ __u8 can_dlc; /* data length code: 0 .. 8
*/ __u8 data[8] __attribute__((aligned(8)));
};
```

Ecco, questa sopra è una versione abbastanza datata (**Linux 2.6**) che non trattava ancora CAN FD. In **Linux 3.7** troviamo la prima versione “moderna” che include il CAN FD:

```
/** * struct can_frame - basic CAN frame structure * @can_id:
CAN ID of the frame and CAN_*_FLAG flags, see canid_t
definition * @can_dlc: frame payload length in byte (0 .. 8)
aka data length code * N.B. the DLC field from ISO 11898-1
Chapter 8.4.2.3 has a 1:1 * mapping of the 'data length code'
to the real payload length * @data: CAN frame payload (up to 8
byte) */
struct can_frame { canid_t can_id; /* 32 bit CAN_ID +
EFF/RTR/ERR flags */ __u8 can_dlc; /* frame payload length in
byte (0 .. CAN_MAX_DLEN) */ __u8 data[CAN_MAX_DLEN]
__attribute__((aligned(8)));
}; /** * struct canfd_frame - CAN flexible data rate frame
structure * @can_id: CAN ID of the frame and CAN_*_FLAG flags,
see canid_t definition * @len: frame payload length in byte (0
.. CANFD_MAX_DLEN) * @flags: additional flags for CAN FD *
@__res0: reserved / padding * @__res1: reserved / padding *
@data: CAN FD frame payload (up to CANFD_MAX_DLEN byte) */
struct canfd_frame { canid_t can_id; /* 32 bit CAN_ID +
EFF/RTR/ERR flags */ __u8 len; /* frame payload length in byte
*/ __u8 flags; /* additional flags for CAN FD */ __u8 __res0;
/* reserved / padding */ __u8 __res1; /* reserved / padding */
__u8 data[CANFD_MAX_DLEN] __attribute__((aligned(8)));
};
```

Poi, in **Linux 5.11**, il *can_frame* è diventato così (mentre il *canfd_frame* è rimasto invariato):

```
/** * struct can_frame - Classical CAN frame structure (aka
CAN 2.0B) * @can_id: CAN ID of the frame and CAN_*_FLAG flags,
see canid_t definition * @len: CAN frame payload length in
byte (0 .. 8) * @can_dlc: deprecated name for CAN frame
payload length in byte (0 .. 8) * @__pad: padding * @__res0:
reserved / padding * @len8_dlc: optional DLC value (9 .. 15)
at 8 byte payload length * len8_dlc contains values from 9 ..
15 when the payload length is * 8 bytes but the DLC value (see
ISO 11898-1) is greater then 8. * CAN_CTRLMODE_CC_LEN8_DLC
flag has to be enabled in CAN driver. * @data: CAN frame
payload (up to 8 byte) */
```

```

struct can_frame { canid_t can_id; /* 32 bit CAN_ID +
EFF/RTR/ERR flags */ union { /* CAN frame payload length in
byte (0 .. CAN_MAX_DLEN) * was previously named can_dlc so we
need to carry that * name for legacy support */ __u8 len; __u8
can_dlc; /* deprecated */ }; __u8 __pad; /* padding */ __u8
__res0; /* reserved / padding */ __u8 len8_dlc; /* optional
DLC for 8 byte payload length (9 .. 15) */ __u8
data[CAN_MAX_DLEN] __attribute__((aligned(8)));
};

```

Come si nota il campo *can_dlc* è stato deprecato (ma non cancellato) e il nome è stato allineato a quello usato nel CAN FD, proprio per permettere la scrittura semplificata di Software per entrambi i tipi senza troppi cambi!. Insomma, alla fine della fiera, ora è possibile riscrivere gli esempi del primo articolo (*cansend.c* e *canrecv.c*) per usare “Classic” o FD definendo una semplice variabile di compilazione (che ho chiamato CAN_FD). Cominciamo con *canfdsend.c*, vai col codice!

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h> // definizione del tipo CanFrame
#ifdef CAN_FD
typedef struct canfd_frame CanFrame; // CAN FD
#else
typedef struct can_frame CanFrame; // CAN classic
#endif // canfdsend - funzione main()
int main(int argc, char *argv[])
{ // creo il socket int sockfd; if ((sockfd = socket(PF_CAN,
SOCK_RAW, CAN_RAW)) == -1) { // errore socket() printf("%s:
errore socket() (%s)\n", argv[0], strerror(errno)); return
EXIT_FAILURE; } // set degli attributi di i/o struct ifreq
ifr; memset(&ifr, 0, sizeof(ifr)); snprintf(ifr.ifr_name,
sizeof(ifr.ifr_name), "%s", "vcan0"); if (ioctl(sockfd,

```

```

SIOCGIFINDEX, &ifr) == -1) { // errore ioctl() printf("%s:
errore ioctl() (%s)\n", argv[0], strerror(errno));
close(sockfd); return EXIT_FAILURE; } #ifdef CAN_FD // set del
modo CAN_FD int enable_canfd = 1; if ( setsockopt(sockfd,
SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable_canfd,
sizeof(enable_canfd)) == -1) { // errore setsockopt()
printf("%s: errore setsockopt() (%s)\n", argv[0],
strerror(errno)); close(sockfd); return EXIT_FAILURE; } #endif
// set attributi dell'indirizzo struct sockaddr_can addr;
memset(&addr, 0, sizeof(addr)); addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex; // assegna l'indirizzo al
socket if (bind(sockfd, (struct sockaddr *)&addr,
sizeof(addr)) == -1) { // errore bind() printf("%s: errore
bind() (%s)\n", argv[0], strerror(errno)); close(sockfd);
return EXIT_FAILURE; } // compongo il frame CanFrame frame;
memset(&frame, 0, sizeof(CanFrame)); frame.can_id = 0x100U;
frame.len = 8; snprintf(frame.data, sizeof(frame.data),
"0123456"); // invio il frame if (write(sockfd, &frame,
sizeof(CanFrame)) != sizeof(CanFrame) == -1) { // errore
write() printf("%s: errore write() (%s)\n", argv[0],
strerror(errno)); close(sockfd); return EXIT_FAILURE; }
close(sockfd); return EXIT_SUCCESS;
}

```

ed ora passiamo a *canfdrecv.c*!

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h> // definizione del tipo CanFrame
#ifdef CAN_FD
typedef struct canfd_frame CanFrame; // CAN FD
#else
typedef struct can_frame CanFrame; // CAN classic
#endif // canfdrecv - funzione main()
int main(int argc, char *argv[])

```

```

{ // creo il socket int sockfd; if ((sockfd = socket(PF_CAN,
SOCK_RAW, CAN_RAW)) == -1) { // errore socket() printf("%s:
errore socket() (%s)\n", argv[0], strerror(errno)); return
EXIT_FAILURE; } // set degli attributi di i/o struct ifreq
ifr; memset(&ifr, 0, sizeof(ifr)); snprintf(ifr.ifr_name,
sizeof(ifr.ifr_name), "%s", "vcan0"); if (ioctl(sockfd,
SIOCGIFINDEX, &ifr) == -1) { // errore ioctl() printf("%s:
errore ioctl() (%s)\n", argv[0], strerror(errno));
close(sockfd); return EXIT_FAILURE; } #ifdef CAN_FD // set del
modo CAN_FD int enable_canfd = 1; if ( setsockopt(sockfd,
SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable_canfd,
sizeof(enable_canfd)) == -1) { // errore setsockopt()
printf("%s: errore setsockopt() (%s)\n", argv[0],
strerror(errno)); close(sockfd); return EXIT_FAILURE; } #endif
// set attributi dell'indirizzo struct sockaddr_can addr;
memset(&addr, 0, sizeof(addr)); addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex; // assegna l'indirizzo al
socket if (bind(sockfd, (struct sockaddr *)&addr,
sizeof(addr)) == -1) { // errore bind() printf("%s: errore
bind() (%s)\n", argv[0], strerror(errno)); close(sockfd);
return EXIT_FAILURE; } // ricevo il frame CanFrame frame; if
(read(sockfd, &frame, sizeof(CanFrame)) == -1) { // errore
read() printf("%s: errore read() (%s)\n", argv[0],
strerror(errno)); close(sockfd); return EXIT_FAILURE; } //
mostro il frame ricevuto printf("id:0x%x dlc:%d data: ",
frame.can_id, frame.len); for (int i = 0; i < frame.len; i++)
printf("%c ", frame.data[i]); printf("\n"); close(sockfd);
return EXIT_SUCCESS;
}

```

Visto come è facile? Nel codice ci sono gli appositi “*#ifdef CAN_FD*” che decidono che parte di codice usare. Quindi per ottenere la versione “Classic” si può compilare con:

```

aldo@Linux $ gcc canfdsend.c -o canfdsend
aldo@Linux $ gcc canfdrecv.c -o canfdrecv

```

Mentre, per ottenere la versione FD si deve compilare con:

```

aldo@Linux $ gcc canfdsend.c -o canfdsend -DCAN_FD
aldo@Linux $ gcc canfdrecv.c -o canfdrecv -DCAN_FD

```

E i cambi nel codice sono veramente limitatissimi, grazie alla flessibilità dell'interfaccia SocketCAN: basta definire un nuovo tipo (che ho chiamato *CanFrame*) da usare al posto della *struct can_frame* (o della *struct canfd_frame*), così i cambi nel codice sono minimi, anche grazie al fatto che il campo *can_dlc* ora si chiama *len* in entrambe le strutture, e quindi, in pratica, l'unico vero cambio nel codice è questo:

```
#ifdef CAN_FD
// set del modo CAN_FD
int enable_canfd = 1;
if ( setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES,
&enable_canfd, sizeof(enable_canfd)) == -1) { // errore
setsockopt() printf("%s: errore setsockopt() (%s)\n", argv[0],
strerror(errno)); close(sockfd); return EXIT_FAILURE;
}
#endif
```

che forza l'uso, nel *socket* aperto precedentemente, dei *frame* di tipo FD. Insomma, come avevo promesso è veramente semplice scrivere codice “*compatibile*” con entrambi i tipi “*Classic*” e FD. E il CAN FD extended? Vabbé, ve lo lascio come utile esercizio, basti sapere che è sufficiente modificare il codice per FD con le minime modifiche che avevo mostrato nello **scorso articolo** al riguardo del CAN extended.

Ok, penso che con questo possa bastare. Il nostro viaggio nei misteri del CAN bus e del suo uso attraverso SocketCAN è terminato, ora possiamo qualificarci come **Guardians of the CAN bus**! Giuro che non ci sarà (*per il momento...*) una quinta parte, quindi nel prossimo articolo parleremo d'altro. E non trattenete il fiato nell'attesa, mi raccomando!

Ciao, e al prossimo post!

<https://italiancoders.it/guardians-of-the-can-bus-come-usare-il-can-bus-in-c-e-c-pt-4/>