Un codec a basso jitter per reti CAN

Il meccanismo di bit stuffing definito dal protocollo Controller Area Network (CAN) è causa di variabilità nella durata della trasmissione dei messaggi sul bus e può peggiorare l'accuratezza con cui i comandi vengono attuati dai dispositivi nei sistemi di controllo distribuiti in tempo reale. Un semplice meccanismo di codifica, implementato in software tramite un codec ad alte prestazioni, può ridurre drasticamente tale problema e rendere CAN adatto a sistemi che richiedono elevata precisione temporale.

Gianluca Cena
Ivan Cibrario Bertolotti
Tingting Hu
Adriano Valenzano

A più di vent'anni dalla sua introduzione il protocollo **Controller Area Network** (CAN) [1] ha conosciuto un'ottima diffusione anche al di fuori del settore veicolistico, ambito applicativo per il quale era stato inizialmente concepito. CAN è correntemente utilizzato negli ambienti industriali e, soprattutto, nei sistemi embedded distribuiti, anche grazie alla sua semplicità e ai bassi costi implementativi, che ne rendono vantaggioso l'uso nonostante le sue prestazioni non siano particolarmente elevate (se confrontate, ad esempio, con soluzioni basate su Ethernet).

A livello fisico, CAN utilizza una codifica di tipo **non-return to zero** (NRZ) con **bit stuffing** (BS). In particolare, ogni qualvolta sono rilevati 5 bit consecutivi allo stesso valore nella sequenza di bit trasmessa sul bus, i controller CAN introducono automaticamente un bit al valore opposto. Tali bit, denominati **stuff bit**, servono a creare un numero adeguato di transizioni sul bus, in modo da consentire ai ricevitori di sincronizzare i propri circuiti e decodificare correttamente il segnale ricevuto. Gli stuff bit sono rimossi dai controller CAN in fase di ricezione prima di decodificare il messaggio.

Il meccanismo di bit stuffing permette un'elevata efficienza di codifica del segnale. Purtroppo, esso introduce anche una variabilità indesiderata dei tempi di trasmissione dei messaggi, effetto che, in teoria, può determinare nel caso peggiore fluttuazioni temporali (**jitter**) pari alla durata di **24 bit**. Infatti, il numero di stuff bit inseriti in ogni singolo messaggio dipende dal contenuto dello stesso, contenuto che non può essere noto in fase di progetto in quanto tipicamente dipendente dal valore delle variabili di processo.

Se, come accade nella maggior parte dei sistemi basati su CAN, l'**interrupt** relativo alla ricezione di un messaggio viene utilizzato dai dispositivi per pilotare azioni a livello applicativo (per esempio, attuare segnali in uscita, effettuare campionamenti sugli ingressi, ecc.), la presenza di stuff bit può peggiorare la qualità del controllo, introducendo di fatto un'incertezza nella valutazione dei tempi.

In molti tipi di applicazione questa variabilità può essere trascurata, ma nel caso di sistemi ad alta precisione essa può risultare particolarmente fastidiosa e difficile da eliminare in quanto peculiare del protocollo. Per ovviare a questo problema nel decennio passato sono state proposte diverse soluzioni [2][3]. In questo articolo descriviamo brevemente una tecnica di codifica, denominata **8B9B** [4][5], che è di fatto un'evoluzione delle proposte precedenti. Essa è infatti in grado di offrire prestazioni notevolmente superiori tanto in termini di efficienza di codifica quanto di velocità di esecuzione, e rappresenta attualmente il meglio dello stato dell'arte.

A titolo di esempio si tenga presente che, in una rete CAN operante a 500 kb/s, 8B9B è in grado di ridurre i jitter di trasmissione dovuti al bit stuffing da circa 50 µs a meno di **10 µs**. In determinati contesti applicativi (sistemi di misura e controllo ad elevata accuratezza) il beneficio è dunque più che sensibile.

Nell'ambito delle attività di ricerca su 8B9B è stato sviluppato un **codec** software ottimizzato per piattaforme **embedded** in grado di effettuare codifica e decodifica in tempi dell'ordine di pochi microsecondi e con un **footprint** di circa **1 kB**. Tale soluzione può quindi essere incorporata direttamente in nuovi progetti e nei dispositivi esistenti con estrema facilità.

Formato delle trame CAN

Ogni trama CAN contiene un certo numero di campi. Come mostrato nella parte inferiore della ►figura 1, il messaggio comincia con un bit di inizio trama (SOF) a valore 0, seguito dal campo

GLI AUTORI
G. Cena, Senior Member, IEEE; I. C. Bertolotti, Member, IEEE; T. Hu, Member, IEEE; A. Valenzano, Senior Member, IEEE.

di arbitraggio composto dall'identificatore del messaggio (ID) e dal bit di richiesta remota (RTR). Seguono poi 2 bit riservati, il campo lunghezza dei dati (DLC) codificato su 4 bit, il campo dati (DATA) nel quale sono memorizzate le informazioni che devono essere scambiate e un campo di controllo (CRC) di 15 bit. Concludono la trama il delimitatore di CRC (CDEL), il campo di acknowledgement (ACK slot) e il relativo delimitatore (ADEL), ognuno codificato su un bit, e infine il campo di fine trama (EOF) di 7 bit.

È importante notare che il meccanismo di bit stuffing opera solo sulla prima parte della trama, ovvero sulla porzione compresa tra il bit SOF e il campo CRC compreso. Le uniche parti della trama il cui valore può essere effettivamente manipolato dalle applicazioni che si interfacciano al controller CAN sono il campo di arbitraggio (ed in particolare l'identificatore) e il campo dati. Infatti, il campo CRC è calcolato autonomamente dal controller mentre il campo DLC dipende direttamente dalla lunghezza (in byte) dei dati. Ne consegue che l'unico modo per ridurre il numero di stuff bit è selezionare in modo opportuno l'identificatore del messaggio (configurato staticamente) e adottare una codifica dinamica per il campo dati.

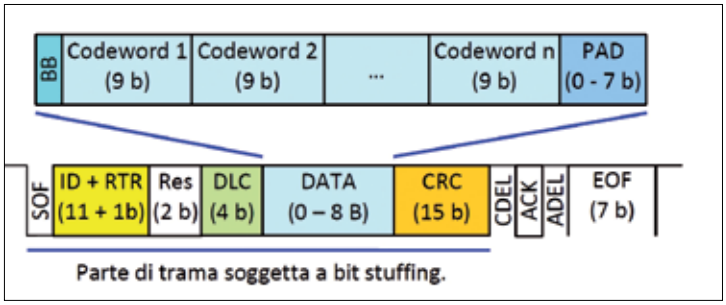

Figura 1 – Formato della trama CAN e 8B9B.

Codifica 8B9B

La codifica 8B9B è particolarmente semplice. Ogni singolo byte del payload originale del messaggio viene convertito separatamente in una sequenza di 9 bit (**codeword**), Tali codeword, opportunamente concatenate, costituiranno il contenuto finale del campo dati da trasmettere. Ovviamente, questo procedimento non si applica nel caso molto particolare (ma lecito) in cui il payload non sia presente, poiché nessuna conversione è di fatto necessaria.

Le codeword utilizzate per la conversione devono soddisfare i due requisiti seguenti:

1) esse non devono contenere sequenze di 5 bit consecutivi allo stesso livello. Per esempio, la sequenza 010000011 è illegale.
2) Durante il processo di concatenazione, la condizione precedente non deve verificarsi neppure a cavallo di codeword adiacenti. A tal fine, è sufficiente scartare anche tutte quelle sequenze di 9 bit che iniziano o terminano con 3 bit dello stesso valore. Per esempio, la sequenza 010101000 è inadatta poiché, se dovesse essere seguita dalla sequenza (valida) 001010101, darebbe luogo alla stringa complessiva 010101000 001010101 (che include 5 bit uguali in successione).

Le sequenze valide possono essere agevolmente determinate mediante un semplicissimo algoritmo. I risultati indicano la disponibilità di 258 codeword. Di queste, 256 sono utilizzate per codificare ogni possibile valore esprimibile su un byte, mentre le due rimanenti, denominate J (001000010 e K(110111101) possono essere utilizzate come sequenze di escape. L'insieme ordinato delle codeword costituisce una tabella di traduzione diretta, altrimenti denominata **forward lookup table** (FLT), che può essere impiegata per il processo di conversione (codifica) dei byte del payload. Un possibile esempio di FLT è riportato nella ►**tabella 1**.

FLT gode per costruzione di una proprietà di **simmetria**, ovvero se Y = FLT[X] allora *not* Y = FLT[*not* X]. Questa proprietà può essere utilizzata per dimezzare la dimensione della tabella e la relativa occupazione di memoria senza impatto alcuno sulle prestazioni del codec.

X_{16}	Y_2	X_{16}	Y_2	X_{16}	Y_2	X_{16}	Y_2
00	001000011	20	001101101	40	010100001	60	011001101
01	001000100	21	001101110	41	010100010	61	011001110
02	001000101	22	001110001	42	010100011	62	011010001
03	001000110	23	001110010	43	010100100	63	011010010
04	001001001	24	001110011	44	010100101	64	011010011
05	001001010	25	001110100	45	010100110	65	011010100
06	001001011	26	001110101	46	010101001	66	011010101
07	001001100	27	001110110	47	010101010	67	011010110
08	001001101	28	001111001	48	010101011	68	011011001
09	001001110	29	001111010	49	010101100	69	011011010
0a	001010001	2a	001111011	4a	010101101	6a	011011011
0b	001010010	2b	010000100	4b	010101110	6b	011011100
0c	001010011	2c	010000101	4c	010110001	6c	011011101
0d	001010100	2d	010000110	4d	010110010	6d	011011110
0e	001010101	2e	010001001	4e	010110011	6e	011100001
0f	001010110	2f	010001010	4f	010110100	6f	011100010
10	001011001	30	010001011	50	010110101	70	011100011
11	001011010	31	010001100	51	010110110	71	011100100
12	001011011	32	010001101	52	010111001	72	011100101
13	001011100	33	010001110	53	010111010	73	011100110
14	001011101	34	010010001	54	010111011	74	011101001
15	001011110	35	010010010	55	010111100	75	011101010
16	001100001	36	010010011	56	010111101	76	011101011
17	001100010	37	010010100	57	011000010	77	011101100
18	001100011	38	010010101	58	011000011	78	011101101
19	001100100	39	010010110	59	011000100	79	011101110
1a	001100101	3a	010011001	5a	011000101	7a	011110001
1b	001100110	3b	010011010	5b	011000110	7b	011110010
1c	001101001	3c	010011011	5c	011001001	7c	011110011
1d	001101010	3d	010011100	5d	011001010	7d	011110100
1e	001101011	3e	010011101	5e	011001011	7e	011110101
1f	001101100	3f	010011110	5f	011001100	7f	011110110

Tabella 1 - Forward Lookup Table usata per la codifica 8B9B.

Break bit e campo di Padding

Il campo dati di ogni messaggio CAN è preceduto dal campo DLC, il cui contenuto non può essere in alcun modo ricodificato in quanto usato per indicare al ricevitore la dimensione dei dati stessi. Questo implica che, in particolari condizioni, potrebbero apparire 5 bit adiacenti uguali a cavallo fra i campi DLC e dati. Per evitare che ciò accada è possibile utilizzare il primo bit del campo dati ridefinendolo come **break bit** (BB). BB viene impostato con un valore complementare a quello del bit meno significativo di DLC. In pratica, BB è 1 quando DLC è pari mentre è 0 in caso contrario.

Un secondo aspetto da considerare nel processo di codifica è che la sequenza di bit ottenuta tramite concatenazione di codeword non occupa un numero intero di byte. Si può porre rimedio a questo problema ridefinendo la porzione inutilizzata dell'ultimo byte del campo dati come campo di **padding** (PAD). Per evitare l'aggiunta di stuff bit, il contenuto del campo PAD viene impostato dal trasmettitore ad una sequenza alternata di bit (ad esempio, 0101…).

Il formato completo della codifica 8B9B è illustrato nella parte superiore della ►**figura 1**.

Dimens. payload originale (B)	Campo dati nel messaggio codificato 8B9B					
	Dimens. campo dati (B)	DLC	Break Bit (BB)		Dimens. seq. concatenata codeword (b)	Dimens. campo PAD (b)
			Valore	Dimens. (b)		
0	0	0000	—	0	0	0
1	2	0010	1	1	9	6
2	3	0011	0	1	18	5
3	4	0100	1	1	27	4
4	5	0101	0	1	36	3
5	6	0110	1	1	45	2
6	7	0111	0	1	54	1
7	8	1000	1	1	63	0
8	—	—	—	—	—	—

Tabella 2 - Confronto fra payload originale e codifica 8B9B.

Esempio di codifica 8B9B

Un breve esempio permette di chiarire il funzionamento pratico del codec e di evidenziarne l'intrinseca semplicità. Si consideri il valore **0xf0**, che in CAN verrebbe codificato su un solo byte (quindi con DLC = 1). La sequenza di bit trasmessa sul bus che corrisponde ai campi DLC e dati in questo caso è **0001 1111000001**. Come si può vedere, in fase di trasmissione il controller CAN aggiunge 2 stuff bit (sottolineati).

Dalla ►tabella 2 si ricava che, per la codifica 8B9B dello stesso dato, DLC = 2. Essendo il valore di DLC pari, BB viene impostato a 1. Ogni singolo byte del payload (solo uno, in questo caso) viene quindi codificato usando la FLT. Poiché 0xf0 è maggiore di 127, e si intende sfruttare la simmetria della FLT della ►tabella 1, occorre complementare tale valore (not 0xf0 → **0x0f**) prima di usarlo come indice nella tabella (FLT**[0x0f]** → **001010110**). La codeword 8B9B relativa al byte 0xf0 si ottiene tramite ulteriore complementazione (**110101001**). Per i valori minori o uguali a 127 la codifica è ottenuta effettuando direttamente l'accesso in tabella.

Concatenando poi BB, la (singola) codeword su 9 bit e il campo PAD, si ottiene il contenuto del campo dati che dovrà essere passato al controller CAN per la trasmissione. La sequenza di bit trasmessa sul bus che corrisponde ai campi DLC e dati è **0010 1 110101001 010101**. Come si può vedere, la dimensione del messaggio è aumentata a causa della codifica 8B9B. Tuttavia, grazie ad essa nessuno stuff bit viene inserito nel campo dati in fase di trasmissione, ottenendo quindi una notevole riduzione dei jitter di comunicazione. Questo significa anche che la durata del messaggio può essere stimata in fase di progetto con maggiore accuratezza.

Prestazioni

Occorre separare chiaramente le prestazioni della **codifica** da quelle del relativo **codec**. La codifica 8B9B permette di ridurre il numero di stuff bit aggiunti in fase di trasmissione dal controller CAN. L'entità del miglioramento dipende anche dal contenuto dei messaggi scambiati. In [6] sono state considerate 3 leggi di generazione per il traffico, che modellano rispettivamente dispositivi di I/O digitali (D) e analogici (A), nonché una generazione casuale di valori (R).

La ►figura 2 mostra la distribuzione statistica delle latenze di trasmissione per CAN con e senza 8B9B. Come si può notare, il jitter (larghezza della distribuzione) si riduce in modo tangibile grazie ad 8B9B, soprattutto con un traffico di tipo analogico (nel qual caso si passa da fluttuazioni massime di 18 bit a soli 4 bit). Ridurre il numero di stuff bit è tuttavia scarsamente utile se il codec introduce esso stesso una variabilità nella codifica e decodifica del campo dati. È quindi necessario utilizzare soluzioni hardware o, in alternativa, codec software altamente **ottimizzati** come quello da noi realizzato e testato. Occorre considerare inoltre l'ulteriore jitter introdotto a livello dell'**interfaccia** fra il software e il controller CAN, problema affrontato e risolto in [7].

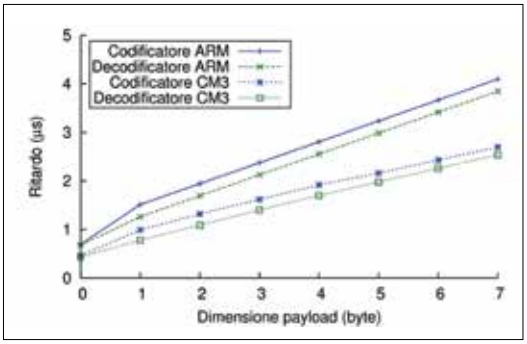

Figura 3 - Ritardo di elaborazione del codec 8B9B.

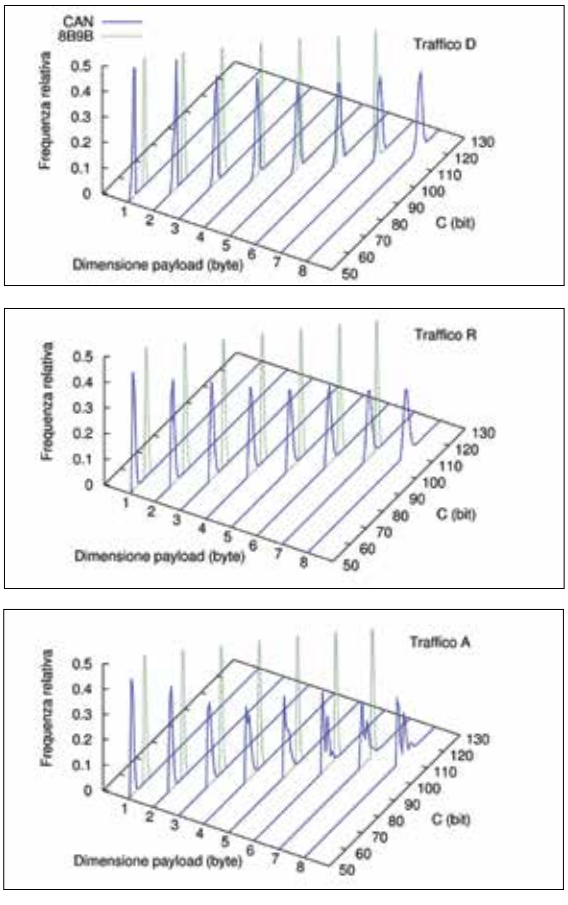

Figura 2 - Distribuzione delle latenze in CAN e 8B9B.

Il codec, più diffusamente descritto in [4], è costituito da un modulo software realizzato in linguaggio ANSI C. Le sue caratteristiche, in termini di occupazione di memoria e ritardo di elaborazione, sono state analizzate sui microcontrollori NXP LPC2468 (basato su processore ARM7 a 72 MHz) e LPC1768 (basato su processore Cortex-M3 a 100 MHz), entrambi rappresentativi della classe di sistemi embedded cui la tecnica di codifica qui descritta prevalentemente è rivolta.

La ►figura 3 illustra le prestazioni del codec per le due architetture considerate, dopo le opportune ottimizzazioni, mostrando il ritardo di codifica e decodifica (in ordinata) in funzione della dimensione del payload originale (in ascissa). Si osservi come il ritardo complessivo di codifica non superi mai gli 8 µs per il processore ARM7 e i 5 µs per il più veloce processore CM3. Tuttavia, aspetto di ancor maggiore rilevanza è che, sempre grazie alle ottimizzazioni effettuate sul codice sorgente, il ritardo del codec è

	Codice (B)	Dati e stack (B)
Codificatore ARM	234	160
Decodificatore ARM	164	284
Totale ARM	**398**	**444**
Codificatore CM3	160	156
Decodificatore CM3	112	276
Totale CM3	**272**	**432**

Tabella 3 - Dimensione del codec 8B9B.

costante per una data lunghezza del payload e non dipende in alcun modo dai dati trasmessi, a meno della precisione con cui le misure sono state effettuate (14 ns per il processore ARM7 e 10 ns per il processore CM3).

I risultati mostrati nella ►**tabella 3** riguardano invece l'occupazione di memoria, per entrambe le architetture e facendo distinzione fra dimensione del codice e dimensione dello spazio dati e stack da esso richiesto. Questa valutazione è particolarmente importante nei sistemi embedded, in quanto codice e dati sono normalmente allocati in aree di memoria diverse.

È utile infine osservare che, per entrambe le architetture, la dimensione del codec 8B9B è molto limitata e non supera, nel complesso, 1 kB. Ciò rende l'integrazione del codec agevole anche su piattaforme con memoria limitata senza necessità di dover modificare in modo significativo il software già esistente.

Riferimenti

[1] ISO, ISO 11898-1 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, International Organization for Standardization, 2003.

[2] T. Nolte, H. Hansson, C. Norström, S. Punnekkat, "Using bit-stuffing distributions in CAN analysis," in *Proc. IEEE/IEE Real-Time Embedded Systems Workshop*, 2001.

[3] M. Nahas, M. J. Pont, M. Short, "Reducing message-length variations in resource-constrained embedded systems implemented using the CAN protocol," *J. of Systems Architecture*, vol. 55, n. 5–6, pp. 344–354, 2009.

[4] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, "Fixed-Length Payload Encoding for Low-Jitter Controller Area Network Communication", *IEEE Transactions on Industrial Informatics*, vol. 9, n. 4, pp. 2155–2164, 2013.

[5] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, "On a family of run length limited, block decodable codes to prevent payload-induced jitter in Controller Area Networks," *Computer Standards & Interfaces*, vol. 35, n. 5, pp. 536–548, 2013.

[6] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano, "Performance comparison of mechanisms to reduce bit stuffing jitters in Controller Area Networks," in *Proc. 17th IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2012, pp. 1–8.

[7] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "Performance evaluation and improvement of the CPU–CAN controller interface for low-jitter communication," in *Proc. 17th IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2012, pp. 1–8.