Preventing bit stuffing in CAN

Gianluca Cena, Ivan Cibrario Bertolotti, Tingting Hu, and Adriano Valenzano, CNR-IEIIT

Some of the drawbacks of CAN depend on bit stuffing. Stuff bits worsen both timing accuracy, because of jitters on transmission times, and data integrity, due to undesired interactions with CRC calculation. The Zero Stuff-bit (ZS) mechanism operates on conventional CAN controllers and prevents stuff bits completely all over the frame by suitably encoding the data field. ZS has been experimentally proven to decrease worst-case transmission jitters from more than 20μs to less than 0.5μs at a bit rate of 1Mb/s. It also achieves a reduction in the residual error probability of about two orders of magnitude, and ensures full coexistence with conventional CAN devices and applications. An industrial-grade ZS codec has been developed for embedded platforms, whose footprint is about 2.5KB only. Its contribution to end-to-end delays is below 12μs. This confirms that ZS can be adopted as an interim software solution to ease migration from CAN to CAN FD.

The signal on the bus in Controller Area Networks (CAN) [1] is encoded using a method known as bit stuffing (BS). While being quite efficient on the average, BS suffers from two main drawbacks. First, the actual duration of each message depends on the specific content of the data field, and not only on the nominal payload size. In turn, this causes jitter on message reception—in theory, up to 24 bit times for base frames—which worsens timing accuracy and, possibly, control quality. Second, as pointed out in [2], BS may interfere severely with the error detection mechanism of CAN, which is based on the Cyclic Redundancy Check (CRC). In particular, a pair of erroneous bits may trick receivers into believing that the frame is still valid. This implies that the residual error probability is noticeably higher than one may expect, hence impairing integrity of communications and consequently system reliability.

The first issue was tackled by the introduction of TTCAN [3]. By decoupling frame reception times from actuation and sensing instants—thanks to the time-triggered paradigm—jitters at the application level are avoided at the expense of increased software complexity. The second issue, instead, is dealt with by CAN FD [4]. In this protocol the CRC is calculated on the whole frame (including stuff bits as well) and encoded with stuff bits placed in fixed positions. Thanks (also) to its much higher speed, there is little doubt that CAN FD will eventually succeed to replace CAN completely.

Both solutions preserve a reasonable degree of backward compatibility with “legacy” CAN. However, a new breed of CAN controllers is required, which means that the existing ones are unsuitable. Much worse, their advantages cannot be exploited in networks where nodes based on conventional controllers are present. In the case of CAN FD, severe error conditions may arise when trying to do so. For these reasons, a solution is welcome that is able to run on unmodified legacy CAN hardware and coexist with existing devices and applications, yet permitting to overcome the drawbacks of CAN due to BS highlighted above.

To this extent, the Zero Stuff-bit (ZS) encoding scheme can be adopted. It acts as a presentation layer and is located between the data-link layer and either the application layer (e.g., CANopen) or applications programs directly. From a practical viewpoint, this corresponds to a codec, which can be implemented as a thin software layer running on the same microcontroller in charge of executing application programs.

The paper is organized as follows: the next section briefly describes the techniques that can be used to prevent stuff bit insertion in CAN, while the ZS encoding scheme is specifically described afterwards. Finally, the last sections describe in detail the ZS codec and its performance, and draw some conclusions.

Bit stuffing in CAN

As shown in Figure 1 (which refers to the base frame format), each CAN frame can be seen as made up of four distinct sections:

- Header (H): comprises the Start of Frame (SOF) bit, the arbitration field—including the identifier and the Remote Transmission Request (RTR) bit—and the control field—made up of reserved bits and the Data Length Code (DLC).
- Data field (D): carries user information as seen by the upper protocol layers (referred to in the following as the original payload (P)). This field is not interpreted by the CAN controller and is completely under control of the applications. The final content of D is obtained by encoding P suitably.
- CRC field (R): represents a signature, evaluated by the transmitter on all the preceding fields of the frame and used by receivers to detect transmission errors. Its content is calculated at runtime by the CAN controller, and hence, it cannot be modified directly by applications.
- Unstuffed trailer (U): includes the CRC delimiter, the ACK slot, the ACK delimiter, and the End of Frame (EOF) field. This section, located at the very end of the frame, is not encoded with BS. Hence, it is not a source of jitter.

Overall, a CAN frame (F) can be seen as

$$F = H \setminus D \setminus R \setminus U$$

where the "\" operator denotes concatenation between sections or bit sequences.
At the physical level, the signal transmitted on the CAN bus relies on a non-return to zero (NRZ) encoding with bit stuffing.

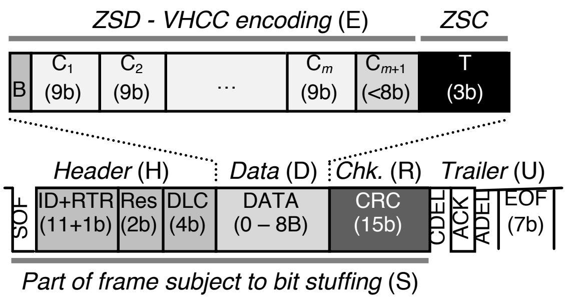

Figure 1: CAN and ZS frame format

Every time 5 consecutive bits at the same level are found in the bit sequence sent on the bus, the CAN controller in the transmitting device(s) automatically inserts a stuff bit at the opposite value. Stuff bits enable a proper synchronization of CAN controllers in receivers, which is essential for decoding the signal read from the bus correctly. They are removed by the controller while the frame is being received. As said above, only the frame stuffed prefix (S in Figure 1) is involved in BS. For this reason, the trailer is irrelevant to our purposes and will not be considered in the following.
Different approaches have to be adopted to deal with stuff bits in different sections of the frame, as explained below.

Frame header

As shown in [5] a careful selection of message identifiers can prevent stuff bits in the frame header completely. However, it is worth noting that, in the vast majority of real-world applications, the header is fixed for any given message stream. As a consequence, at worst it introduces a fixed number of stuff bits (up to 4 in base frames), which do not cause any jitter.
Concerning data integrity, in order to prevent the issues highlighted in [2], strictly speaking there must be no stuff bits in the header. Nonetheless, reducing the number of stuff bits in the header as much as possible, and preventing them in the other parts of the frame, reduces the probability of such an occurrence.

Data field

The data field in CAN frames is under complete control of the user. For this reason, specific encoding schemes can be purposely adopted for the payload in order to avoid the insertion of stuff bits in D.
Former approaches [5] operated by scrambling P before copying it into D. For instance, P can be XOR-ed with a fixed pattern made up of an alternating bit sequence. As shown in [6], when process data encode patterns with specific generation laws, doing so permits to reduce the average number of stuff bits.
Conversely, more recent solutions like 8B9B [7] are able to prevent the insertion of stuff bits in D completely, by exploiting block codes where each byte of the payload is encoded separately.

8B9B belongs to the class of the Zero Stuff-bit Data (ZSD) encoding schemes, and in particular to ZSD2, where no more than 2 bits at the same level can be found at the end of the encoded payload. Codewords are 9 bit long and permit to encode every possible value expressed on one byte. Each codeword satisfies two properties:

P1. No more than 4 consecutive bits at the same value can be found everywhere;

P2. No more than 2 consecutive bits at the same value can be found at both ends.

It is easy to show that properties *P1* and *P2* also hold for every bit sequence obtained by concatenating two or more codewords. Hence, the conditions that trigger stuff bit insertion in the encoded payload are prevented in advance.

The ZS encoding scheme described below relies on the Variable-length, High-performance Code for CAN (VHCC) [8]. Basically, VHCC resembles 8B9B closely (that is, it is ZSD_2), but relies on a different codebook, which satisfies a nesting property that enables variable-length data encoding.

As proven in [8], both VHCC and 8B9B offer the best communication efficiency that can be obtained in practical implementations. In addition, VHCC permits to reuse the same lookup tables in the codec in order to perform sub-byte encoding and decoding. Doing so enables applications to exploit every available bit in D to transfer user information.

CRC field

Dealing with stuff bits in the CRC is not easy, as this field is managed completely inside CAN controllers. In the following, CRC calculation according to the CAN rules is modeled as a function c(·)

$$R = c(M)$$

where the message M coincides with the part of the frame covered by the CRC (from the SOF bit up to the end of D)

$$M = H \setminus D$$

Up to 4 stuff bits may be added to R (depending on M), which cause jitter and worsen reliability. The value of R can be "shaped" by using a small portion of D, denoted tuning string (T). As proven in [9], reserving 4 bits for T (at the end of D) is sufficient so that the computation of R can be always steered to a value that does not include any stuff bits.

In the case the payload is encoded using ZSD2, 3 bits will suffice. This mechanism is known as Zero Stuff-bit CRC[1] (ZSC).

Since part of D is reserved to store T, the remaining portion, available to encode user information and referred to as effective data field (E), is slightly smaller. Overall, D can be seen as

$$D = E \setminus T$$

Zero stuff-bit encoding schemes

In the following, a brief description is provided about the encoding schemes that are used in ZS to prevent stuff bits in the data and CRC fields. Concerning the header, please refer directly to [5]. All the techniques described there are compatible with the ZS scheme and can be used in concert with it.

VHCC

The behavior of VHCC is quite simple. Let P_i be the i-th byte of the original payload P ($1 \le i \le m$, where m is the original payload size). Frames with no payload ($m = 0$) will not be considered, as the related sequence of bits sent on the bus is fixed. Additional sub-byte user information, encoded on h bits ($1 \le h \le 7$), can be possibly present in P_{m+1}, so that

$$P = P_1 \setminus P_2 \setminus \ldots \setminus P_i \setminus \ldots \setminus P_m \setminus P_{m+1}$$

First, every whole byte P_i ($1 \le i \le m$) is encoded separately into a codeword C_i using a forward lookup table (FLT), modeled as a function $f(\cdot)$

$$C_i = f(P_i)$$

Such codewords are then concatenated in the same order as P

$$C_{<1\ldots m>} = C_1 \setminus C_2 \setminus \ldots \setminus C_i \setminus \ldots \setminus C_m$$

Since the size of $C_{<1\ldots m>}$ is typically smaller than E, part of the last byte in D (between C_m and T) becomes a slack space (K). Let k be its size (in bits).

In 8B9B, all the bits that followed C_m and preceded R remained unused and were filled with a fixed padding sequence (PAD).

[1] Patent pending

Instead, in the current advanced ZS code, they can be used to encode P_{m+1}. In particular, the slack bears an additional (smaller) codeword C_{m+1}, which is derived from a reduced codebook as

$$C_{m+1} = f_k(P_{m+1})$$

where $f_k(\cdot)$ shares the same FLT and implementation as $f(\cdot)$. From a general point of view, the maximum number of bits allowed for additional user data is one less than the slack size, i.e., max(h) = k-1. For instance, up to 4 bits of P_{m+1} can be encoded when 5 bits are allotted to C_{m+1}.

If K includes only one bit (k = 1), then no additional data can be included (h = 0) and C_{m+1} has to be set to the opposite value of the least significant bit in Cm. If $h < k$-1 then Pm+1 has to be left padded to k-1 bits with zeroes. In the case P_{m+1} is not present, any value can be used in its place, since the padding produced by $f_k(\cdot)$ always prevents BS. Overall, the concatenation of codewords (C) is

$$C = C_{<1\ldots m>} \setminus C_{m+1}$$

In order to prevent the occurrence of a stuff bit on the boundary between H and D, a break bit (B) is possibly added as the first bit of D, evaluated by complementing the least significant bit of DLC. B is only required when DLC equals 3, 7, or 8, whereas it is omitted otherwise. The *encoded payload* is obtained by concatenating B and all the codewords

$$E = B \setminus C$$

Since the payload encoded this way fits exactly the entire effective data field, the same symbol E will be used in the following to denote both.

Generally speaking, every ZSD2 encoding scheme can be modeled as a function e(·) that, starting from the payload P and the header H, returns the effective data field

$$E = e(H, P)$$

ZSC

ZSC exploits the tuning field, located at the end of D, in order to prevent the insertion of stuff bits in R completely. As proven in [9], and because of the linearity of CRC codes, it is always possible to select a value for T in the set {0012, 0102, 0112, 1002, 1012, 1102}, which includes only six Ti values, so that the above condition holds for any given header and payload.

Calculation of T is not completely trivial, and can be modeled as a function z(·)

$$T = z(H, E)$$

A possible way to do so is as follows: first, the part of frame that precedes T, called the leading part (L) of the message

$$L = H \setminus E$$

is considered, and its contribution to the overall CRC evaluated

$$R_L = c(L \setminus 000_2)$$

Then, the contribution to R due to T is evaluated and XOR-ed with RL for any possible value of T_i mentioned above, in order to find the related CRC value Ri.

Let M_i be the message obtained by setting T = T_i

$$Mi = L \setminus Ti$$

It can be easily proven that

$$Ri = c(Mi) = RL \text{ O+ } c(Ti), \quad i = 1\ldots6$$

where "O+ " represents the exclusive OR operator.

Let $g(\cdot)$ be a Boolean function that is true if (and only if) a given bit sequence includes more than 4 consecutive bits at the same value (i.e., if it would lead to stuff bits). Every value Ti such that

$$g(T_i \setminus R_i) = \text{false}$$

can be used as the outcome of z(·). From [9] we know that there is at least one. In the case there is more than one acceptable value, the choice of which to select is arbitrary. Performing an exhaustive search and selecting the one with the highest value of i helps reducing the codec jitter in software implementations.

Putting all together

The whole ZS procedure can be described as follows. First, the encoded payload is evaluated as E = e(H, P). Then, the tuning string is obtained as T = z(H, E). Finally, the outcomes of VHCC and ZSC (E and T, respectively) are put together: the value to be used for the data field is evaluated as D = E \ T. The way H and D are fed to the CAN controller is completely irrelevant to our purposes. When operating this way, no stuff bits will ever be added to the data and CRC fields by the CAN controller, as proven in [9].

Although ZS has very good encoding efficiency, it necessarily introduces overheads, which reduce the amount of information that can be conveyed in CAN frames.

In Table 1, the maximum amount of user information that fit into a single frame is reported for every possible frame size (DLC).
It has been calculated starting from the size of D (listed in the two leftmost columns of the table), subtracting the space reserved to T (3 bits) and B (if needed), and then considering the VHCC encoding overhead of 1 bit for each codeword.
The result (that is, the size of P) is specified in the rightmost column of the table as *m.h*, where *m* represents the number of whole bytes (P_1 to P_m) whereas *h* is the number of additional bits (in P_{m+1}). As can be seen, up to 6 bytes plus 5 bits can be carried in a CAN frame with maximal size (DLC = 8).

Table 1: Maximum payload size in ZS

D	DLC	E			P (*m.h*)
		B	$C_{<1...m>}$	C_{m+1}	
size [B]	val.	val.	size [b]	size [b]	size [B.b]
0	0000	-	0	0	0.0
1	0001	-	0	5	0.4
2	0010	-	9	4	1.3
3	0011	0	18	2	2.1
4	0100	-	27	2	3.1
5	0101	-	36	1	4.0
6	0110	-	45	0	5.0
7	0111	0	45	7	5.6
8	1000	1	54	6	6.5

Implementation and performance

The primary goal of ZS is to make the duration of frame transfers in CAN deterministic, hence removing any transmission jitter due to BS. To this extent, it has to be remarked that hardware implementations of ZS (e.g., based on FPGAs) are able to reduce this jitter to zero.
Moreover, a very interesting side effect of stuff bit removal is that data integrity improves noticeably, because the interaction between BS and CRC calculation no longer takes place.
As shown in [10], the residual error probability decreases by about two orders of magnitude when a ZSD codec like 8B9B is used alone. The addition of the ZSC mechanism in ZS is likely to improve reliability further, because stuff bits are removed from the CRC field as well. Unlike nested CRCs, ZS has the advantage of improving both timing accuracy and data integrity at the same time. Research activities on these aspects are in progress.
It is worth noting that not necessarily all messages exchanged on the CAN bus must undergo ZS encoding. On the contrary, it can be enabled selectively (depending on the message identifier) uniquely for those data that require high reliability and low jitter. Only nodes that exchange ZS-encoded messages are required to implement the related codec. Moreover, ZSC is necessary only on the transmitting side: receivers only have to strip the tuning string away. In this way, system implementation complexity and cost decrease, and seamless coexistence with existing CAN devices and applications is ensured.

ZS codec

The ZS method can be implemented efficiently through software codecs. As discussed in the previous sections, from the conceptual point of view, ZS encoding is performed in two steps:

1. Payload encoding, according to VHCC;
2. Generation of T, according to ZSC.

Given that the same approach has been adopted in its practical implementation, we can consider that ZS encoding time is the sum of the times spent in the two steps listed above.
In the decoding process, T is simply discarded before forwarding E to VHCC decoding. Given that discarding T can be done in negligible time on any modern microcontroller, we can safely assume that ZS decoding time as a whole is equal to VHCC decoding time. The same reasoning also applies to memory footprint.
Considering ZS *encoding* first, experimental results about VHCC encoding performance and memory footprint on an NXP LPC1768 microcontroller [11] running at 100MHz, are available in [8] and are shown in the top rows of Tables 2 and 3, respectively. For what concerns performance, the table lists the worst-case encoding delay, that is, the one corresponding to the maximum size of D (and to a size of P equal to 6.5), as well as the corresponding computation jitter.
Regarding footprint, separate figures are given for read-only and read/write memory, a distinction often important in embedded systems.

Moreover, separate values have been given for code and data, the latter consisting of the lookup tables used by the various code modules. Stack space has been accounted for as read/write memory.

The ZSC algorithm has instead been implemented (in combination with the 8B9B payload encoding scheme) on an NXP LPC2468 microcontroller [12] running at 72MHz and then evaluated in [9]. As before, performance and footprint data are shown in the center rows of Tables 2 and 3 even though, in this case, they must be considered as a conservative approximation of ZSC encoding properties on the LPC1768.

This is because, besides the raw difference in clock speed already mentioned above, the two platforms support different instruction sets and are based on different processor cores, too. More specifically, the LPC2468 is a low end component based on an ARM7TDMI S core designed in 2001. Instead, the LPC1768 is based on the contemporary Cortex-M3 processor core with a better instruction/cycle ratio, due to architectural enhancements.

The improvement may be quantified by examining the experimental results reported in [7], where the 8B9B codec was evaluated on both platforms. In that case, performance was on average about 35% better and code footprint was about 30% lower on the LPC1768 with respect to the LPC2468. Clearly, using one architecture or the other could not bring any improvement to table sizes.

Concerning table sizes, it is also worth to remark that the relatively large size of the ZSC table with respect to the others is due to the fact that, in order to generate T properly, the ZSC algorithm needs to calculate L's contribution to the CRC RL. When this operation is performed on a byte-by-byte basis, it requires a lookup table of 28=256 15-bit entries, that is, 512 bytes as indicated in Table 3.

As mentioned above, it is possible to accurately approximate ZS decoder footprint and the corresponding delay by considering only the VHCC decoder contribution [8], shown in the bottom rows of Tables 2 and 3.

Summarizing the results, ZS encoding and decoding as a whole can be performed by introducing less that 12μs of delay in the communication path.

Even more importantly, software processing introduces a negligible amount of jitter, less than 0.5μs. Hence, it does not go against one of the goals of ZS itself, that is, jitter reduction. Indeed, the remaining communication jitter is bit-rate independent and well below one bit time, even at the maximum CAN bit rate of 1Mbit/s.

Furthermore, as pointed out in [9], a low-jitter version of the ZS encoder was also developed, which trades off at most 1.49μs of extra delay to reduce jitter below 30ns.

It is also possible to get an estimate of the actual delay and footprint of the ZS encoder, when it is completely implemented on the LPC1768, by applying the LPC2468LPC1768 improvement factors mentioned above.

The predicted worst-case delay turns out to be 10.15μs, while the predicted footprint is 2193B. However, due to the inherent uncertainty of the method (applying a footprint and performance improvement factor derived from analyzing a certain code module to another), these numbers shall be taken as an estimate and will be the subject of further experimental evaluation in a future work.

Table 2: ZS codec performance

Processing step	Delay [μs]	Jitter [μs]
VHCC encoding (worst case)	3.30	0.00
ZSC encoding	5.25	0.47
VHCC decoding (worst case)	3.44	0.00
Total	**11.99**	**0.47**

Table 3: ZS memory footprint

Component	Read-only [B]	Read/Write [B]
VHCC encoder code	368	48
VHCC encoder table	128	0
ZSC encoder	898	20
ZSC table	512	0
VHCC decoder code	300	40
VHCC decoder table	256	0
Total	**2462**	**108**

Summary

CAN FD brings noticeable improvements over CAN, which include much higher throughput and better data integrity. Unfortunately, its advantages cannot be easily exploited in networks to which nodes based on legacy controllers are attached.

In this paper, an encoding scheme is described, referred to as Zero Stuff-bit (ZS), which permits to prevent the insertion of stuff bits completely in frames sent by conventional CAN controllers. As a consequence, it completely eliminates the related transmission jitter and noticeably reduces the residual error probability.

A portable, highly optimized industrial-grade ZS codec for typical embedded platforms has been developed in our Institute, mainly considering the NXP LPC1768 [11] and LPC2468 [12] microcontrollers as a case study. It has been thoroughly tested in order to prove feasibility, correctness, and performance of our solution [9]. While ZS does not improve throughput, it successfully addresses both jitter and data integrity issues that affect conventional CAN communications. For this reason, we believe that ZS can be adopted as an interim software solution to ease migration from CAN to CAN FD.

Gianluca Cena
CNR-IEIIT
C.so Duca degli Abruzzi, 24
IT-10129 Torino
Tel./Fax +39 011 090 5424/5429
gianluca.cena@polito.it
http://ceng.ieiit.cnr.it/gc

Ivan Cibrario Bertolotti
CNR-IEIIT
C.so Duca degli Abruzzi, 24
IT-10129 Torino
Tel./Fax +39 011 090 5426/5429
ivan.cibrario@polito.it
http://ceng.ieiit.cnr.it/icb

Tingting Hu
CNR-IEIIT
C.so Duca degli Abruzzi, 24
IT-10129 Torino
Tel./Fax +39 011 090 5432/5429
tingting.hu@polito.it
http://ceng.ieiit.cnr.it/people/tingting.hu/

Adriano Valenzano
CNR-IEIIT
C.so Duca degli Abruzzi, 24
10129 Torino - Italy
Tel./ Fax +39 011 090 5410/5429
adriano.valenzano@polito.it
http://ceng.ieiit.cnr.it/av

References

[1] ISO 11898-1: Road vehicles – Controller area net¬work – Part 1: Data link layer and physical signal¬ling (International Organization for Standardization, 2003).

[2] J. Charzinski: Performance of the error detection mechanisms in CAN (in: Proc. iCC 1994, pp. 20–29).

[3] ISO 11898-4 – Road vehicles – Controller area network – Part 4: Time-triggered communication, (International Organization for Standardization, 2004).

[4] ISO/DIS 11898-1: Road vehicles – Controller area net¬work – Part 1: Data link layer and physical signal¬ling (International Organization for Standardization, 2015).

[5] 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).

[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. IEEE ETFA 2012, pp. 1–8).

[7] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano: Fixed-Length Payload Encoding for Low-Jitter Controller Area Network Communication (in: IEEE Trans. Ind. Informat., vol. 9, no. 4, pp. 2155–2164, 2013).

[8] 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 (in: Comput. Stand. Interfaces, vol. 35, no. 5, pp. 536–548, 2013).

[9] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano: A mechanism to prevent stuff bits in CAN for achieving jitterless communication (in: IEEE Trans. Ind. Informat., vol. 11, no. 1, pp. 83–93, Feb. 2015).

[10] G. Cena, I. Cibrario Bertolotti, T. Hu, A. Valenzano: Effect of jitter-reducing encoders on CAN error detection mechanisms (in: Proc. IEEE WFCS 2014, pp. 1–10).

[11] LPC17XX User manual, UM10360 rev. 2, (NXP B.V., Aug. 2010).

[12] LPC24XX User manual, UM10237 rev. 2, (NXP B.V., Dec. 2008).