Authors' Accepted Manuscript (AAM): this is the accepted version of the paper. Citation information: DOI: 10.1109/TII.2014.2365153, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, VOL. 11, NO. 1, FEBRUARY 2015

A Mechanism to Prevent Stuff Bits in CAN for Achieving Jitterless Communication

Gianluca Cena, *Senior Member, IEEE,* Ivan Cibrario Bertolotti, *Member, IEEE,* Tingting Hu, *Member, IEEE,* and Adriano Valenzano, *Senior Member, IEEE*

***Abstract*—The bit stuffing mechanism adopted in Controller Area Networks leads to unwanted jitter on frame reception times, which worsens timing accuracy even if countermeasures are adopted to avoid contentions on the bus. Several solutions have been proposed so far for dealing with stuff bits in the payload of messages, but they are not effective for the CRC.**

In this paper a mechanism is presented that prevents the occurrence of stuff bits in the whole frame completely. It makes the duration of frame transmissions fixed and, hence, it achieves very accurate reception times. An optimized codec has been implemented to demonstrate that this approach is feasible and can be profitably adopted in low-cost networked embedded systems with demanding timing constraints.

***Index Terms*—Controller area network (CAN), industrial control, real-time distributed systems.**

I. Introduction

BESIDES the automotive scenario, Controller Area Networks (CAN) [1] are today increasingly used in networked embedded systems as well [2]. The main reason is that CAN permits a significant reduction of implementation costs. Actually, many modern microcontrollers embed one or more CAN controllers natively. Moreover, unlike Ethernet-based solutions, connections in CAN rely on inexpensive cables, and neither intermediate equipment (switches) nor specific components (magnetics) on end devices are required.

Distributed real-time control applications sometimes demand for the ability of enforcing actions in different places of the controlled physical system at exact time instants [3]–[7]. Although time-triggered solutions are certainly the best option [8], a (much) simpler event-driven approach could often suffice. In this case, actions are triggered on devices following the correct reception of suitable commands from the network and, in particular, upon the interrupt signal raised by the communication controller. This involves the ability to control both the transmission instant and frame duration precisely.

When timing constraints are tight (for instance, if jitters are required to stay well below $100\,\mu$s), suitable countermeasures have necessarily to be adopted in CAN to avoid that frame exchanges suffer from delays due to bus contentions (i.e., when a node loses arbitration). A simple solution is operating the system according to a master-slave approach—at least, in the operational phase, when determinism is a key requirement. Such a kind of solution manages to offer good performance, high flexibility, and low complexity at the same time. Moreover, in recent years a significant amount of research went into reducing interrupt latency jitter within real-time operating systems to less than one hundred clock cycles, even on off-the-shelf devices [9].

Unfortunately, even if all the other sources of jitter are removed or mitigated, the *bit stuffing* mechanism (BS) the CAN physical layer relies on still causes variations in the duration of message exchanges, which in theory can be up to 24 bit times in the case of standard format frames [10]—actually, the worst case is $22\,\mathrm{b}$, since some bits in the frame header have a fixed value [11]. They make reception times suffer from unwanted jitters and affect timing accuracy negatively.

Several approaches were defined to overcome this limitation of CAN. In particular, XOR-based solutions [10], [12], [13] operate by scrambling the content of the data field in software before the frame is sent, trying to reduce the number of stuff bits. More recent solutions, such as SBS [14], EEM [15], 8B9B [16], and VHCC [17] are able to prevent the occurrence of stuff bits in the data field completely by using suitable encoding schemes. We will refer to the latter class of approaches as *Zero Stuff-bit Data* (ZSD). In particular, 8B9B was shown to provide the best encoding efficiency in its class [16]. A very efficient implementation of the codec exists, which has a really small footprint and proves the effectiveness of the approach even on low-end microcontrollers. In every case, a decoder is required on the receiver side to get the original payload back.

The main drawback of all the above approaches is that, the occurrence of stuff bits is prevented only in the payload of the message. Two significant parts of the frame are left out, namely the header and the cyclic redundancy check (CRC). The frame header (that contains information about the message identifier and size) does not constitute a real problem. In fact, typical applications rely on a static priority assignment and the payload size for a given message identifier is not allowed to change over time. Hence, the number of stuff bits that are inserted in the header is fixed, known in advance, and does not lead to communication jitters. The CRC field, instead, is more problematic. Its value is computed in hardware by the CAN controller at run time. Apparently, the system designer has no means to prevent (or just reduce) the number of stuff bits inserted in this part of the frame. As pointed out in [16],

G. Cena, I. Cibrario Bertolotti, and A. Valenzano are with the National Research Council of Italy, Institute of Electronics, Computer and Telecommunication Engineering (CNR-IEIIT), I-10129 Turin, Italy.

T. Hu is with the National Research Council of Italy, Institute of Electronics, Computer and Telecommunication Engineering (CNR-IEIIT), I-10129 Turin, Italy, and also with Politecnico di Torino, Department of Control and Computer Engineering (DAUIN), I-10129 Turin, Italy.

[1]Patent pending.

this implies that a residual jitter—which can be as high as 4 bit times—seems to be unavoidable, even with ZSD approaches. This is clearly unacceptable when a timing accuracy in the order of one μs or less is needed.

In this paper a new mechanism is introduced, known as *Zero Stuff-bit CRC* (ZSC)[1], that enhances ZSD encoding schemes (like 8B9B) by preventing the occurrence of stuff bits in the CRC. The resulting *Zero Stuff-bit* (ZS) code, which combines ZSD and ZSC, is able to avoid the insertion of stuff bits in every variable part of the CAN frame, hence it leads to truly fixed transmission times over the bus. A prototype codec has been also developed. When coupled with a suitable mechanism able to prevent bus contentions (e.g., a master-slave approach), this makes CAN communication timings completely deterministic, without having to resort to time-triggered paradigms.

The paper is structured as follows: in Section II the problem of jitters related to bit stuffing in CAN is briefly recalled, and a possible solution is sketched, whereas in Section III the ZSC mechanism is introduced and its correctness is proved. More details about the practical implementation of ZSC, as well as its execution time and footprint are given in Section IV, while Section V draws some concluding remarks.

II. Preventing bit stuffing in CAN

Every time a sequence is found over the bus that consists of 5 adjacent bits at the same level (referred to in the following as *primer sequence*), the transmitting CAN controller automatically inserts one *stuff bit* at the opposite value, hence creating an edge in the bit stream. This enables receivers to synchronize their bit timings accurately, and ensures that the signal read from the bus is decoded properly, irrespective of the actual data included in the frame.

It is worth pointing out that, from a practical point of view, preventing stuff bits in the CRC field by means of the ZSC mechanism is mostly useless, unless countermeasures have been taken to remove all the sources of BS jitter from the other parts of the frame, too. In particular, ZSD encoding schemes can be adopted to this aim to cope with the data field. They satisfy the following two properties:

1) the encoded data field never includes primer sequences;
2) primer sequences are not found at the boundary between the frame header and the encoded data field.

To make jitter prevention easier in the CRC, a third property may be optionally taken into account, which requires that no more than a given number k_e of bits at the same value be found at the end of the encoded data field ($1 \leq k_e \leq 4$). The class of related encoding schemes will be denoted ZSD_{k_e}. For these classes, the property $k_1 < k_2 \Rightarrow \mathrm{ZSD}_{k_1} \subseteq \mathrm{ZSD}_{k_2}$ holds. As having more than 4 adjacent bits at the same value causes the insertion of a stuff bit, ZSD_4 actually coincides with ZSD.

In the following, ZSD_2 schemes will be considered explicitly. However, most results we obtained apply, with slight changes, to other approaches as well.

A. CAN frame format

As shown in the lower part of Fig. 1, every frame in CAN is made up of different sections (in the following, the term *section* will be used to refer to either a single field or a specific part of the frame comprising several adjacent fields):

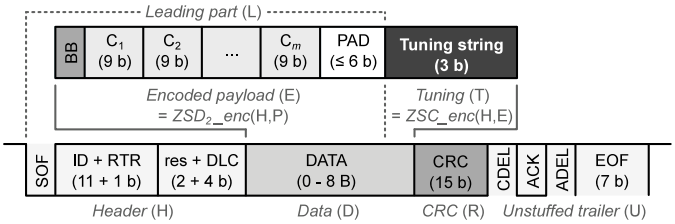

Fig. 1. CAN frame format (11-bit id.) with 8B9B and ZSC encoding.

- *Header* (H), consisting of the *start of frame* bit (SOF), *identifier* (ID), *remote transmission request* bit (RTR), two *reserved* bits (r1 and r0) and *data length code* (DLC).
- *Data* field (D), which is not interpreted by the CAN controller and includes information to be exchanged among devices over the network. It is built from the *original payload* (P), which is defined at the application level, and is completely under control of the application itself. All mechanisms that prevent/lessen the effects of bit stuffing in CAN operate by encoding P in a suitable way before storing the result into D.
- *CRC* field (R), evaluated as the remainder of a polynomial division modulo 2 on all the preceding fields.
- *Unstuffed trailer* (U), which includes *CRC delimiter*, *ACK slot*, *ACK delimiter* and *end of frame* field (EOF).

Bit stuffing affects only the part of the frame from the SOF bit up to the end of the CRC field, whereas U has a fixed form. For this reason, it is not relevant at all in our analysis and will not be considered in the following any longer.

B. Definitions

The nominal size in bits of a generic section Y is denoted n_{Y}: for instance, $n_{\mathrm{H}} = 19$ (using standard 11-bit identifiers), $n_{\mathrm{R}} = 15$, $n_{\mathrm{U}} = 10$, while $n_{\mathrm{D}} = 8 \cdot \mathrm{DLC}$ (where $0 \leq \mathrm{DLC} \leq 8$). Calligraphic letters refer to the sets of legal values (bit sequences) that sections can assume (before bit stuffing is applied). For convenience, section names are also used to denote a specific, legal value of that section. For instance, $\mathcal{Y}$ denotes the set of all legal values that section Y can assume and $\mathrm{Y} \in \mathcal{Y}$ represents one of them. In general, $|\mathcal{Y}| \leq 2^{n_{\mathrm{Y}}}$ and the strict inequality holds when some values of Y are illegal.

Let "$\wr$" denote the concatenation operator between sections. For example, section $\mathrm{X} \wr \mathrm{Y}$ is made up of all the bits in X (in the original order) followed by all the bits in Y (in the original order). When sizes are concerned, $n_{\mathrm{X} \wr \mathrm{Y}} = n_{\mathrm{X}} + n_{\mathrm{Y}}$. By extension, the same operator will also be used to concatenate individual bits and bit sequences.

As proved in the following for specific conditions, part of the bits in D can be used to prevent the insertion of stuff bits in the CRC field. Such bits, located at the very end of the data field, are referred to as the *tuning* field (T) and are leveraged by the ZSC mechanism to "shape" the bit pattern in R properly. Quite obviously, this requires that D is not empty, that is, $\mathrm{DLC} > 0$. To this extent, it is worth remarking that neither ZSD nor ZSC apply to messages with no payload (i.e.,

data frames with $\mathrm{DLC} = 0$ and remote frames). However, in these specific cases the pattern of bits corresponding to the frame is fixed and known in advance (provided that the header does not change at runtime). As a consequence, the number of stuff bits inserted by the CAN controller (if any) is also constant, which does not lead to any transmission jitter.

The part of D allotted to the user information—i.e., with the exclusion of the tuning field—is referred to as the *encoded payload* (E) and carries (in our case encoded through a ZSD scheme) the original payload P. This implies that $\mathrm{D} = \mathrm{E} \wr \mathrm{T}$, which means that the number of bits in D that are actually available for E is $n_\mathrm{E} = 8 \cdot \mathrm{DLC} - n_\mathrm{T}$. Together, header and encoded payload build up the *leading part* of the frame (L), defined as $\mathrm{L} = \mathrm{H} \wr \mathrm{E}$.

C. CRC computation

The CRC computation is carried out in hardware by the CAN controller according to a predefined scheme. For this reason it is just impossible to rely on approaches similar to 8B9B (or any other proposed in the literature) in order to prevent the occurrence of stuff bits in the CRC field.

The portion of the frame that is covered by the CRC is referred to as *message* (M) and is given by

$$\mathrm{M} = \mathrm{H} \wr \mathrm{D} = \mathrm{H} \wr \mathrm{E} \wr \mathrm{T} = \mathrm{L} \wr \mathrm{T} \ , \tag{1}$$

while the whole CAN *frame* (F) can be expressed as $\mathrm{F} = \mathrm{M} \wr \mathrm{R} \wr \mathrm{U}$. Of course, we are interested only in the portion of the frame that is affected by bit stuffing, namely, $\mathrm{M} \wr \mathrm{R}$.

Concerning CRC computation, every frame section can be seen as a *polynomial*. The same symbol is used in the following to denote both a section and the corresponding polynomial, that is, polynomial $Y(x)$ corresponds to section Y. The degree of $Y(x)$ is $n_\mathrm{Y} - 1$ while its coefficients y_i ($0 \leq i \leq n_\mathrm{Y} - 1$) are the same as the bits in Y.

For instance, the CRC field can be seen as a bit sequence

$$\mathrm{R} = r_{n_\mathrm{R}-1} \wr r_{n_\mathrm{R}-2} \wr \ldots \wr r_1 \wr r_0 \ , \tag{2}$$

while the corresponding polynomial is

$$R(x) = \sum_{i=0}^{n_\mathrm{R}-1} r_i \cdot x^i \ . \tag{3}$$

From (1) the polynomial associated to M is

$$M(x) = L(x) \cdot x^{n_\mathrm{T}} + T(x) \ . \tag{4}$$

The CRC is defined as the remainder of the division between $M(x) \cdot x^{n_\mathrm{R}}$ and the generator polynomial $G(x)$:

$$M(x) \cdot x^{n_\mathrm{R}} = Q(x) \cdot G(x) + R(x) \ , \tag{5}$$

or, using the mod operator (in modulo 2 arithmetic)

$$R(x) = M(x) \cdot x^{n_\mathrm{R}} \bmod G(x) \ . \tag{6}$$

From a practical viewpoint, the numerator is obtained by appending 15 bits at the "0" value to the end of M. Let $c(\cdot)$ denote the function that computes the CRC in CAN. Then,

$$\mathrm{R} = c(\mathrm{M}) = c(\mathrm{L} \wr \mathrm{T}) \ . \tag{7}$$

D. 8B9B encoding

Although ZSC can be profitably paired to any ZSD encoding scheme, in the following attention is focused specifically on 8B9B (or, more generally, on the class of codes it belongs to). In these cases, a codebook $\mathcal{C}$ is used for translating every single byte P_i of the original payload ($\mathrm{P} = \mathrm{P}_1 \wr \ldots \wr \mathrm{P}_m$, where $m \geq 1$ is the size in bytes of P) to a distinct $9\,\mathrm{b}$ pattern $\mathrm{C}_i \in \mathcal{C}$. Other such codebooks exist, that feature different properties. For instance, the one used in VHCC [17] satisfies a nesting property, which can be exploited to encode sub-byte values.

Such translation process can be modeled as a function $f(\cdot)$ so that $\mathrm{C}_i = f(\mathrm{P}_i)$. From a practical point of view the codebook $\mathcal{C}$ in 8B9B corresponds to a forward lookup table (FLT), which includes 256 entries (memory optimizations are possible thanks to its symmetry [16]). On the receive side, a reverse lookup table (RTL) is required to get back each single byte of the original payload, that is, $\mathrm{P}_i = f^{-1}(\mathrm{C}_i)$.

Patterns C_i obtained from translations are then concatenated, in the original order, and become part of E. Let us denote with $\mathcal{G}_{n_\mathrm{C},k_l,k_b,k_t}$ the set of all the binary strings of length n_C, which have at most k_l leading bits at the same value, at most k_b consecutive inner bits at the same value, and at most k_t trailing bits at the same value. If $\mathcal{C} \subseteq \mathcal{G}_{9,2,4,2}$ (as in the case of the 8B9B family of encoding schemes) then codewords do not include more than 4 consecutive inner bits at the same level and no more than 2 bits at the same level at both ends. It can be easily shown that the same property holds for the bit sequence obtained by concatenating codewords C_i (as long as this sequence is not empty), that is, $\mathrm{C}_1 \wr \mathrm{C}_2 \wr \ldots \wr \mathrm{C}_m \in \mathcal{G}_{9 \cdot m,2,4,2}$ (m also corresponds to the number of $9\,\mathrm{b}$ patterns in E).

Depending on the value of DLC (that does not vary for any given message stream), a *break bit* (BB) may be included by 8B9B at the beginning of the data field, which is set at the opposite value than the last bit of the DLC field. Moreover, as shown in the upper part of Fig. 1, n_D is a multiple of $8\,\mathrm{b}$ whereas each codeword is $9\,\mathrm{b}$ long. Hence, generally D is not completely filled by $\mathrm{BB} \wr \mathrm{C}_1 \wr \mathrm{C}_2 \wr \ldots \wr \mathrm{C}_m$ and T. The remaining space, which lies between the last codeword C_m and the tuning field T, is referred to as *padding* (PAD). Its value, which is effectively unused, has to be chosen so that it does not cause the insertion of any stuff bit. To this extent, an alternating bit pattern is sufficient [16].

The resulting bit sequence for the encoded payload, whose general structure is $\mathrm{E} = \mathrm{BB} \wr \mathrm{C}_1 \wr \mathrm{C}_2 \wr \ldots \wr \mathrm{C}_m \wr \mathrm{PAD}$, satisfies the *first* ZSD property. As shown in [16], BB prevents propagation of stuff bits from H to E when necessary. Hence, primer sequences cannot appear at the boundary between these two sections, and the *second* ZSD property holds as well. If the filling pattern is selected so that the first bit of PAD is at the opposite value than the last bit of C_m, no more than two bits at the same value can ever be found at the end of L, even in the limit cases when $n_\mathrm{PAD} = 1$ or 0. Therefore, also the *third* property is true for $k_e = 2$. This implies that 8B9B fulfills all the requirements for ZSD_2 codes. Consequently, such an approach (and the like) can be effectively adopted for practical implementations of ZSC.

The condition that a specific encoded payload E, following a given header H, satisfies the ZSD_{k_e} properties, is generically expressed as $\mathrm{E} \in \mathcal{E}_{\mathrm{H}}^{(\mathrm{ZSD}_{k_e})}$ (only the last bit of H is actually relevant in the case of 8B9B). The resulting leading part L will be said to belong to $\mathcal{L}^{(\mathrm{ZSD}_{k_e})}$, defined as the set of all the legal values section L of the frame may assume when the payload is encoded with a ZSD_{k_e} scheme.

E. Counting stuff bits

In order to detect the occurrence of a primer sequence, also previously inserted stuff bits have to be taken into account. This may lead to a domino effect, which in the worst case results in the insertion of one stuff bit every time 4 bits at the same value are found in the original frame [10]. As a consequence, bit stuffing in any section of the frame is affected by all preceding fields. This means that, in theory, in order to evaluate the exact number of stuff bits that are added to the CRC, sections H, E, and T have to be considered.

Let $s(\mathrm{Y})$ be a function that returns the exact number of stuff bits that are added to the bit sequence Y when considered alone—that is, when the contribution of any field that (possibly) precedes Y is not taken into account. Moreover, let $s_{\mathrm{X}}(\mathrm{Y})$ be a function that returns the actual number of stuff bits that are specifically added to section Y when preceded by the bit sequence X, that is:

$$s_{\mathrm{X}}(\mathrm{Y}) \triangleq s(\mathrm{X} \wr \mathrm{Y}) - s(\mathrm{X}) \ . \tag{8}$$

It is worth noting that $s(\mathrm{X} \wr \mathrm{Y})$ is not necessarily the same as $s(\mathrm{X}) + s(\mathrm{Y})$, because the final bits in X may contribute to the creation of a primer sequence across the boundary with Y. The difference (if any) between $s(\mathrm{Y})$ and $s_{\mathrm{X}}(\mathrm{Y})$ is due to X, which represents the part of the frame that precedes Y.

$s(\mathrm{F})$ represents the overall number of stuff bits that are added to F by the CAN controller (in hardware):

$$s(\mathrm{F}) = s(\mathrm{H}) + s_{\mathrm{H}}(\mathrm{E}) + s_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) \ . \tag{9}$$

Obviously, no stuff bits are added to the unstuffed trailer.

The header is the first field in the frame. Under the (reasonable) assumption that H does not change at run-time for a given message stream, the number of stuff bits added to this section, denoted s_{H}, is fixed ($s_{\mathrm{H}} \geq 0$). As a consequence, no jitter on transmission times is due to H. By means of a careful selection of message identifiers (but not for every possible size of the payload) s_{H} can often be reduced to 0.

If a ZSD scheme is adopted, no stuff bits at all can be inserted in the encoded payload, irrespective of the header:

$$s_{\mathrm{H}}(\mathrm{E}) = s(\mathrm{E}) = 0, \ \ \forall \mathrm{H} \in \mathcal{H}, \forall \mathrm{E} \in \mathcal{E}_{\mathrm{H}}^{(\mathrm{ZSD})} \ . \tag{10}$$

Therefore, the overall jitter on frame reception due to BS depends only on stuff bits added to the fields that follow the encoded payload, that is, the tuning and CRC fields:

$$s(\mathrm{F}) = s_{\mathrm{H}} + s_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}), \ \ \forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})} \ . \tag{11}$$

So that the algorithm used to select the value of T in ZSC can work properly, bit stuffing in T and R must be decoupled from the preceding fields in the frame (i.e., L). While a second break bit, located right before T, may suffice—irrespective of the actual encoding of E—exploiting the peculiar properties of ZSD_2 codes is a better solution. Moreover, not every value of T is legal in order not to add any stuff bits.

Let $\mathcal{T}$ be a subset of the values T may assume ($n_{\mathrm{T}} > 1$), which satisfies the following properties:

$$\mathcal{T} \subseteq \begin{cases} \{01, 10\}, & n_{\mathrm{T}} = 2 \\ \mathcal{G}_{n_{\mathrm{T}},2,4,4}, & n_{\mathrm{T}} > 2 \ . \end{cases} \tag{12}$$

The first property means that, in the case T includes only two bits, they are not allowed to be at the same value. Instead, in the second property it is indicated that T must not include primer sequences and no more than two leading bits can be found at the same level.

It can be proved that, if $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)}$ and $\mathrm{T} \in \mathcal{T}$, then the number of stuff bits added to the tuning and CRC fields equals $s(\mathrm{T} \wr \mathrm{R})$, that is:

$$s_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) = s(\mathrm{T} \wr \mathrm{R}) \ . \tag{13}$$

In fact, because of (12) and the properties of ZSD_2 codes, no more than 4 bits at the same level can be found at the boundary between L and T. Therefore, primer sequences are prevented there. This implies that the value $s_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R})$ depends only on the values of T and R and not on the (preceding) leading part L of the frame. Under the above hypotheses, $s(\mathrm{F})$ can be obtained as

$$s(\mathrm{F}) = s_{\mathrm{H}} + s(\mathrm{T} \wr \mathrm{R}) \ . \tag{14}$$

Basically, (13) implies that stuff bits are completely prevented in the part of the frame that follows L, provided that E is ZSD_2 encoded, T is (suitably) selected in $\mathcal{T}$, and no primer sequence is found in $\mathrm{T} \wr \mathrm{R}$. According to (7), R depends on L too. Hence, section L still affects (indirectly) the value of $s(\mathrm{T} \wr \mathrm{R})$ in (13) and hence $s(\mathrm{F})$ in (14).

III. Preventing bit stuffing in the CRC

The CRC cannot be controlled directly, as it is calculated in hardware. Conversely, size and value of the tuning field can be selected in software. As conjectured above, T could be computed at runtime—before the data field is copied into the CAN controller—so as to prevent primer sequences from occurring in the CRC.

The ZSC feasibility depends on the existence of a function $z(\cdot)$ that, starting from L, determines a value for T able to prevent stuff bits in the CRC (and T as well). That is to say:

$$\forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})}, \exists \mathrm{T} \in \mathcal{T} \ \mid \ \mathrm{T} = z(\mathrm{L}) \ \wedge \ s_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) = 0 \ . \tag{15}$$

The cardinality of $\mathcal{T}$ (and, consequently, $\mathcal{E}$ and $\mathcal{L}$) is directly affected by n_{T}. However, this is not highlighted explicitly in the notation in order to keep it simple. It can be proved that, if (15) holds for a given value of n_{T}, then it certainly holds for any higher values (since $\mathcal{L}$ consequently shrinks).

Before trying to conceive an efficient algorithm for $z(\cdot)$, the minimum size of T that makes the above approach feasible (if any) has to be evaluated. In the following, it will be shown that 3 bits are sufficient to this purpose. In theory, this could be (trivially) demonstrated by considering all possible combinations of bits for the header, encoded payload, and

tuning fields. In particular, for any size n_{T} of T and for any value $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})}$, a set $\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle}$ is determined as

$$\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle} \triangleq \{\mathrm{T} \mid s(\mathrm{L} \wr \mathrm{T} \wr c(\mathrm{L} \wr \mathrm{T})) = s_{\mathrm{H}}\} \ . \tag{16}$$

Each set $\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle}$ may include zero or more elements. If the condition

$$\exists n_{\mathrm{T}} \mid \mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle} \neq \emptyset, \ \forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})} \tag{17}$$

holds, then, generally speaking, the tuning field can be used to prevent stuff bits in the CRC completely. In this case, the minimum value of n_{T} that satisfies (17), denoted $\min(n_{\mathrm{T}})$, is the optimal choice and would demonstrate the claim.

Unfortunately, this approach cannot be adopted in practice. This is because, set $\mathcal{L}^{(\mathrm{ZSD})}$ is too large to carry out an exhaustive search. In fact, although $\left|\mathcal{L}^{(\mathrm{ZSD})}\right| < |\mathcal{M}|/2^{n_{\mathrm{T}}}$, $\mathcal{M}$ includes more than 2^{11+64} patterns for standard identifiers and even more for extended ones. In the following, some techniques are described aimed at reducing dramatically the state space and, consequently, the search complexity.

A. Partial remainders

The ZSC mechanism relies on the linearity of CRC codes. In particular, the value of the R field in the frame sent over the bus can be decoupled into two independent contributions, which depend on the L and T portions of message M, respectively. The following partial remainders can be defined by carrying out independent divisions for the sections that make up M:

$$\begin{aligned} R_{\mathrm{L}}(x) &\triangleq (L(x) \cdot x^{n_{\mathrm{T}}+n_{\mathrm{R}}}) \mod G(x) \\ R_{\mathrm{T}}(x) &\triangleq (T(x) \cdot x^{n_{\mathrm{R}}}) \mod G(x) \ . \end{aligned} \tag{18}$$

It is possible to prove that the R field can be evaluated from the partial remainders related to the L and T sections, that is:

$$\mathrm{R} = c(\mathrm{L} \wr \mathrm{T}_0) \oplus c(\mathrm{T}) \ , \tag{19}$$

where "$\oplus$" denotes bitwise EXOR and T_0 consists of a sequence of n_{T} bits at the 0 value.

To this extent, it should be noted that the following properties hold for $R_{\mathrm{L}}(x)$ and $R_{\mathrm{T}}(x)$:

$$\begin{aligned} L(x) \cdot x^{n_{\mathrm{T}}+n_{\mathrm{R}}} &= Q_{\mathrm{L}}(x) \cdot G(x) + R_{\mathrm{L}}(x) \\ T(x) \cdot x^{n_{\mathrm{R}}} &= Q_{\mathrm{T}}(x) \cdot G(x) + R_{\mathrm{T}}(x) \ . \end{aligned} \tag{20}$$

By summing these contributions, from (4) we find

$$\begin{aligned} &M(x) \cdot x^{n_{\mathrm{R}}} = \\ &\quad [Q_{\mathrm{L}}(x) + Q_{\mathrm{T}}(x)] \cdot G(x) + R_{\mathrm{L}}(x) + R_{\mathrm{T}}(x) \ . \end{aligned} \tag{21}$$

As the degree of polynomials $R_{\mathrm{L}}(x)$ and $R_{\mathrm{T}}(x)$ is certainly lower than $G(x)$, the same property holds for their sum as well. This means that $R_{\mathrm{L}}(x) + R_{\mathrm{T}}(x)$ in (21) equals $R(x)$ in (5).

From (18), working on bit sequences and because of (6) and (7), $\mathrm{R}_{\mathrm{L}} = c(\mathrm{L} \wr \mathrm{T}_0)$ and $\mathrm{R}_{\mathrm{T}} = c(\mathrm{T})$. Equation (19) is proved by remembering that additions among polynomials in modulo 2 arithmetic correspond to bitwise EXOR operations on the corresponding bit patterns.

TABLE I
CONTRIBUTION OF THE TUNING FIELD TO THE CRC

j	T_j	$c(\mathrm{T}_j)$ (binary)	$c(\mathrm{T}_j)$ (hex)
1	001	100 0101 1001 1001	0x4599
2	010	100 1110 1010 1011	0x4eab
3	011	000 1011 0011 0010	0x0b32
4	100	101 1000 1100 1111	0x58cf
5	101	001 1101 0101 0110	0x1d56
6	110	001 0110 0110 0100	0x1664

B. Zero Stuff-bit CRC mechanism

In this section a *sufficient condition* is determined for the feasibility of the ZSC mechanism. In particular, the minimum size of the T field, $\min(n_{\mathrm{T}})$, which ensures that no stuff bit will ever be inserted in $\mathrm{T} \wr \mathrm{R}$ when $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)}$, is 3 bits.

In order to demonstrate the above claim, exhaustive space state analysis can be carried out on a proper subset of states, small enough to make the analysis computable in a finite (and short) time. The reasoning below is repeated for increasing values of n_{T} from 2 on.

Instead of considering separately every possible value for L, only its contribution R_{L} to the CRC is taken into account. All possible values $\mathrm{R}_k \in \mathcal{R}$ have to be considered for R_{L}, where $\mathcal{R}$ is the set of values R can assume.

Let $\mathcal{L}_{\mathrm{R}_k}$ be a set of L values defined as

$$\mathcal{L}_{\mathrm{R}_k} \triangleq \left\{\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)} \mid c(\mathrm{L} \wr \mathrm{T}_0) = \mathrm{R}_k\right\} \ . \tag{22}$$

Sets $\mathcal{L}_{\mathrm{R}_k}$ are disjoint, i.e., $\mathrm{R}_x \neq \mathrm{R}_y \Rightarrow \mathcal{L}_{\mathrm{R}_x} \cap \mathcal{L}_{\mathrm{R}_y} = \emptyset$, and their union equals the universe, $\bigcup_{\mathrm{R}_k \in \mathcal{R}} \mathcal{L}_{\mathrm{R}_k} = \mathcal{L}^{(\mathrm{ZSD}_2)}$.

For every pattern R_k, all possible values $\mathrm{T}_j \in \mathcal{T}$ of the tuning field have been considered. Because of (19),

$$\begin{aligned} c(\mathrm{L} \wr \mathrm{T}_j) &= c(\mathrm{L} \wr \mathrm{T}_0) \oplus c(\mathrm{T}_j) \\ &= \mathrm{R}_k \oplus c(\mathrm{T}_j), \quad \forall \mathrm{L} \in \mathcal{L}_{\mathrm{R}_k} \ . \end{aligned} \tag{23}$$

In order to meet the assumptions of (13), only patterns T_j that satisfy (12) have to be taken into account. This is clearly a limiting assumption. Since the CAN controller, when transmitting a frame, actually knows L, discarding in advance the values of T not included in $\mathcal{T}$ is not strictly necessary. Nevertheless, we are looking for a sufficient condition that ensures complete jitter prevention, and operating in this way is necessary to make exhaustive search feasible.

In Table I, the value of $c(\mathrm{T}_j)$ is shown for every pattern T_j, in the specific case when $n_{\mathrm{T}} = 3$. The case $n_{\mathrm{T}} = 2$ can be dealt with easily, by using just the first two values of $c(\mathrm{T}_j)$ from the table—due to a known property of CRC calculation that leading zeros do not change the final result.

For every $\langle \mathrm{R}_k, \mathrm{T}_j \rangle$ pair, R is computed from (23) and the presence of stuff bits in the tuning and CRC fields is checked by evaluating $s_{\mathrm{L}}(\mathrm{T}_j \wr \mathrm{R})$. Because of (13), only $\mathrm{T}_j \wr \mathrm{R}$ is relevant to this extent.

The space of $\langle \mathrm{R}_k, \mathrm{T}_j \rangle$ pairs was explored exhaustively so as to determine whether or not the following property holds:

$$\forall \mathrm{R}_k \in \mathcal{R}, \exists \mathrm{T}_j \in \mathcal{T} \mid s(\mathrm{T}_j \wr (\mathrm{R}_k \oplus c(\mathrm{T}_j))) = 0 \ . \tag{24}$$

The size of the space to be explored does not exceed $2^{n_{\mathrm{T}}+n_{\mathrm{R}}}$. When the tuning field includes 3 bits, this leads to

$6 \cdot 2^{15}$ states, which can be easily dealt with in a very short time on any modern computer.

We found that, if $n_{\mathrm{T}} = 2$, some messages exist in which one or more stuff bits are still added to $\mathrm{T} \wr \mathrm{R}$, whatever the value chosen for T. Due to the way this result was obtained, also the case $n_{\mathrm{T}} = 1$ (that, indeed, needs to be dealt with in a slightly different manner) is ruled out for sure. On the contrary, when $n_{\mathrm{T}} = 3$, then at least one T_j always exists such that, irrespective of the specific content of H and E, no stuff bits at all are added to $\mathrm{T}_j \wr \mathrm{R}$. The above reasoning fully demonstrates the initial claim about the sufficient condition and, as a consequence, ZSC feasibility.

As a conclusion, it should be noted that:

1) the number of stuff bits in H is fixed and known;
2) ZSD schemes prevent stuff bits in E completely;
3) ZSC, when combined with ZSD, prevents stuff bits completely in $\mathrm{E} \wr \mathrm{T} \wr \mathrm{R}$.

Consequently, the overall jitter in CAN due to BS can be reduced to 0 by a suitable ZS algorithm that encodes P in D.

IV. Implementation and Experimental Results

The practical implementation of ZSC had two main goals. The first intent was to verify by experiment that the CAN frames generated by a combination of a suitable ZSD_2 encoder plus ZSC incur no bit stuffing at the data-link level, as a way to double-check the theoretical proof given in Section III.

Secondly, effort has been put in optimizing the prototype codec so as to reduce both the amount of jitter it injects into the communication path, and its processing time. Results show that the proposed approach is feasible also in practice, and its adoption neither disrupts CAN communication performance, nor impacts the memory requirement of the system in a significant way.

As discussed in the following, performance evaluation experiments carried out on large sets of data revealed that the total communication jitter ($1.08\,\mu$s in the worst case, but it could be further improved) is 40 times lower than in plain CAN (22 bit times, corresponding to $44\,\mu$s when the CAN bit rate is $500\,\mathrm{kb/s}$). Actually, this is entirely due to data processing jitter in the codec and residual jitter at the interface between the CPU and the CAN controller [18].

A. Implementation

As discussed before, operation of the ZS encoder, which includes both ZSD_2 and ZSC, is described by the algorithm shown in Fig. 2. In practice, all experiments have been performed by means of the testbed shown in Fig. 3, which faithfully reproduces the whole software stack needed for ZS CAN communication. The 8B9B codec [16], slightly modified as described in the following, has been chosen as ZSD_2. The main testbed components (white blocks in Fig. 3) are the following:

- A *test harness* schedules the experiment, and then collects and stores test results for later analysis.
- A *test data generator* produces the application-level test data to be used for the experiments.

1) $ZS\_enc(\text{in H}, \text{in P}) \rightarrow \text{out D}$
2) $\quad \mathrm{E} := ZSD_2\_enc(\mathrm{H}, \mathrm{P})$;
3) $\quad \mathrm{T} := ZSC\_enc(\mathrm{H}, \mathrm{E})$;
4) $\quad \mathrm{D} := \mathrm{E} \wr \mathrm{T}$;
5) $\quad$ return D;
6) $ZSD_2\_enc(\text{in H}, \mathrm{P}) \rightarrow \text{out E}$
7) $\quad \mathrm{E} := \mathrm{BB}$; // break bit depends on DLC (i.e., H)
8) $\quad$ for $i = 1$ to m // size in bytes of the payload
9) $\quad\quad \mathrm{E} := \mathrm{E} \wr f(\mathrm{P}_i)$; // translate one byte with FLT
10) $\quad \mathrm{E} := \mathrm{E} \wr \mathrm{PAD}$; // padding depends on E
11) $\quad$ return E;
12) $ZSC\_enc(\text{in H}, \mathrm{E}) \rightarrow \text{out T}$
13) $\quad \mathrm{R_L} := c(\mathrm{H} \wr \mathrm{E} \wr 000_2)$;
14) $\quad$ for each $\mathrm{T}_j \in \{001_2, 010_2, 011_2, 100_2, 101_2, 110_2\}$
15) $\quad\quad \mathrm{R}_j := \mathrm{R_L} \oplus c(\mathrm{T}_j)$;
16) $\quad\quad$ if $s(\mathrm{T}_j \wr \mathrm{R}_j) = 0$ then $\mathrm{T} := \mathrm{T}_j$;
17) $\quad$ return T; // the last T value is taken as $z(\mathrm{H} \wr \mathrm{E})$

Fig. 2. Operation of the ZS encoder.

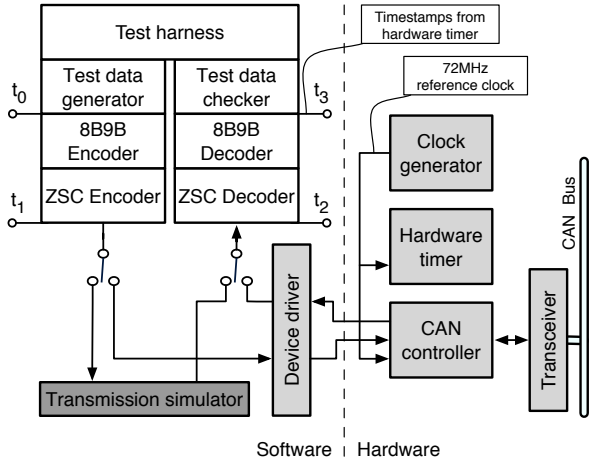

Fig. 3. Experimental testbed.

- The *8B9B* and *ZSC encoders* encode the test data and build the data field of the CAN frames to be transmitted conforming to the algorithm shown in Fig. 2.
- The *8B9B* and *ZSC decoders* recover the test data from the received CAN frames. The ZSC decoder is actually very thin and must just discard the tuning field before feeding the incoming data into the ZSD_2 decoder.
- The *test data checker* verifies that the encoding/decoding process did not modify the test data in any way.

The most important difference between the general algorithm shown in Fig. 2 and its practical implementation is that, for the sake of efficiency, the boundary between partial and complete CRC calculation (corresponding to variables R_L and R_j in the algorithm) has been moved to the position between the last byte of D and its predecessor, instead of being placed between E and T_j. This optimization does not affect results because (19) is actually valid regardless of where the boundary is. By analogy with (19), when L is split at a byte boundary as $\mathrm{L} = \mathrm{L}_1 \wr \mathrm{L}_2$, it is

$$R = c(\mathrm{L}_1 \wr 00000000_2) \oplus c(\mathrm{L}_2 \wr \mathrm{T}) \ , \qquad (25)$$

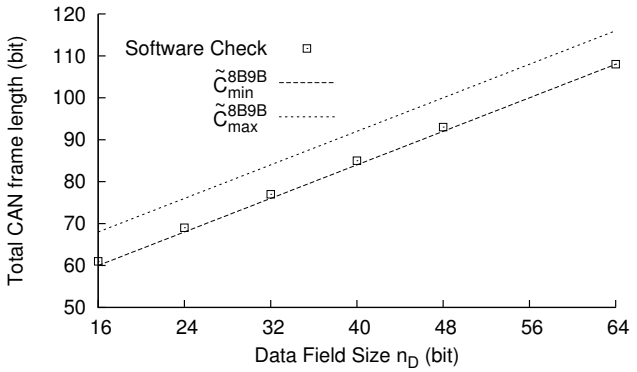

Fig. 4. Total CAN frame length, as a function of n_{D}, measured in the software-based correctness check (10^9 samples for each experiment).

and the width of $\mathrm{L}_2 \wr \mathrm{T}$ is one byte. To facilitate the CRC calculation, a predefined CRC lookup table with $2^8 = 256$ two-byte entries, which stores $c(x)$ for any one-byte value x, is used in the implementation.

B. Software- and Hardware-Based Correctness Checks

When the testbed is configured for *software-based* checks, the ZS modules exchange CAN frames through a *transmission simulator* [19] (dark grey block in Fig. 3). It implements CRC calculation, bit-stuffing and, in general, all the other parts of the CAN data-link protocol specification needed to compute the final length of a CAN frame as it is transmitted on wire. Those lengths are collected by the test harness.

The software-based checks have been carried out with the test data generator configured to produce 10^9 uniformly-distributed, random data items for each length from 1 to 6 bytes, corresponding to a data field size n_{D} from 16 to 64 bits after encoding. In fact, due to the 8B9B algorithm properties, n_{D} is generally one byte longer than the original data item length. The original payload length of 6 bytes is an exception, because it produces a sequence of 54 8B9B-encoded bits. If that sequence were stored in a 56-bit data field, there would not be enough room for the tuning field, because $\min(n_{\mathrm{T}}) = 3$. As a consequence, extra room should be added to make tuning possible. Since n_{D} must be a multiple of 8 due to CAN frame format constraints, it must be $n_{\mathrm{D}} = 64$ in this case.

Results of the software-based correctness check are shown in Fig. 4 (□ diagram). The total CAN frame length measured by the transmission simulator during the experiments is plotted as a function of n_{D}. The sample variance is not shown, because it was zero in all experiments. In other words, the total CAN frame length was constant for any given n_{D}, regardless of the data being transmitted, thus confirming that the ZS encoder works as intended.

As a further reference, two quantities $\widetilde{C}^{8\mathrm{B9B}}_{\mathrm{min}}$ and $\widetilde{C}^{8\mathrm{B9B}}_{\mathrm{max}}$ have also been plotted in Fig. 4. They are defined as the theoretical lower and upper bounds of the CAN frame length taking into account BS, when 8B9B is used alone. As 8B9B can completely prevent stuff bits in the data field, the difference $\widetilde{C}^{8\mathrm{B9B}}_{\mathrm{max}} - \widetilde{C}^{8\mathrm{B9B}}_{\mathrm{min}}$ represents the worst-case transmission time variation that may be introduced by the frame header and CRC fields. In the best case, no stuff bits are added, and hence, for CAN messages with standard 11-bit identifiers, it is

$$\widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{min}} = 19 + n_{\mathrm{D}} + 15 + 10 = 44 + n_{\mathrm{D}} \ , \qquad (26)$$

where the value of $44\,\mathrm{b}$ represents the combined size of the frame header ($19\,\mathrm{b}$), the CRC ($15\,\mathrm{b}$), and the trailer ($10\,\mathrm{b}$).

On the other hand, the worst-case sequence concerning bit stuffing is made up of $5\,\mathrm{b}$ at `0`, followed by an alternating pattern made up of $4\,\mathrm{b}$ at `1` and $4\,\mathrm{b}$ at `0`. Therefore, as shown in [11], the maximum number $k_{\max}(p, l)$ of stuff bits that can be inserted in a message fragment of length l bits—when it is preceded by at most p consecutive bits ($0 \le p \le 4$) at the same value as the leading bit of the fragment—is

$$k_{\max}(p, l) = \left\lfloor \frac{p + l - 1}{4} \right\rfloor \ . \qquad (27)$$

As a consequence, $\widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{max}}$ can be calculated as

$$\begin{aligned} \widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{max}} &= \widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{min}} + k_{\max}(0, 19) + k_{\max}(2, 15) \\ &= 44 + n_{\mathrm{D}} + 4 + 4 = 52 + n_{\mathrm{D}} \ . \end{aligned} \qquad (28)$$

In the previous formula, $k_{\max}(0, 19)$ represents the worst-case number of stuff bits added to the header (since the SOF bit at `0` is always preceded by a bit at `1`) and $k_{\max}(2, 15)$ gives the worst-case number of stuff bits added to the CRC (because, due to the 8B9B properties, the CRC can be preceded at worst by 2 consecutive bits at the same value as the leading bit of the CRC itself).

By comparing (26) and (28), the worst-case message length variation $\widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{max}} - \widetilde{C}^{\,8\mathrm{B9B}}_{\mathrm{min}}$ that may lead to jitter in transmission is therefore at most $8\,\mathrm{b}$ ($4\,\mathrm{b}$ in the header and another $4\,\mathrm{b}$ in the CRC), when considering standard 11-bit identifiers.

The experimental results shown in Fig. 4 reveal that this jitter has been completely removed by the ZS encoder. The only difference of 1 bit between the actual frame length and $\widetilde{C}^{8\mathrm{B9B}}_{\mathrm{min}}$ for $n_{\mathrm{D}} = 16, \ldots, 48$ is due to the *fixed* amount of bit stuffing (not jitter) introduced in the message header for the given combinations of the message identifier chosen for the experiments ($2\mathrm{AA}_{16}$) and the DLC field. For the same reason, the 1 bit difference is not there when $n_{\mathrm{D}} = 64$.

In order to further investigate the correctness of the codec and evaluate its performance, the testbed shown in Fig. 3 can also be configured to perform *hardware-based* checks, by enabling the light gray components. With respect to the software-based configuration, two remarkable differences exist:

1) The transmission simulator is replaced by a *device driver* for a hardware CAN controller.
2) A *hardware timer* with a resolution of $13.9\,\mathrm{ns}$ (corresponding to a $72\,\mathrm{MHz}$ clock frequency) is used to timestamp each test message at points $t_0, \ldots, t_3$ as shown in the figure. t_1 and t_2 are taken just before (after) transmitting (retrieving) a CAN message to (from) the CAN controller, while t_0 and t_3 are placed before encoding starts and after decoding ends.

For hardware-based checks, the whole system has been implemented on an LPC 2468 [20], [21] single-chip microcontroller and the CAN controller has been configured to run at a line speed of $500\,\mathrm{kb/s}$ in self-reception mode.

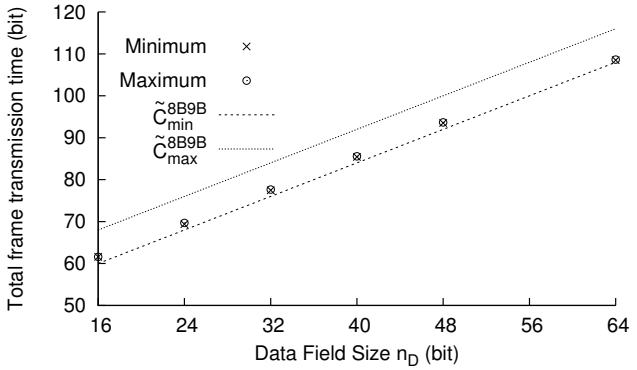

Fig. 5. Min./max. CAN frame transfer time $t_2 - t_1$, as a function of n_{D}, from the hardware-based correctness check (10^7 samples for each experiment).

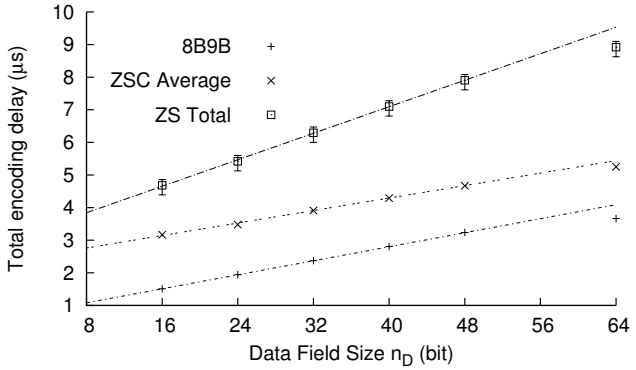

Fig. 6. Encoding delay and jitter as a function of n_{D} (10^7 samples for each experiment).

TABLE II
HARDWARE-BASED CAN FRAME TRANSFER TIME AND RESIDUAL JITTER (10^7 SAMPLES FOR EACH VALUE OF n_{D}, CAN LINE RATE OF 500 kb/s)

	Frame transmission time ($t_2 - t_1$)			
n_{D}	Average (μs)	Min (μs)	Max (μs)	Jitter (μs)
16	123.16	123.16	123.16	0.00
24	139.09	138.75	139.36	0.61
32	155.14	154.94	155.25	0.31
40	171.10	170.83	171.14	0.31
48	187.12	187.03	187.33	0.30
64	217.12	216.97	217.28	0.31

TABLE III
ENCODING DELAY (10^7 SAMPLES FOR EACH VALUE OF n_{D})

n_{D}	8B9B Delay (μs)	ZSC Delay Average (μs)	ZS Delay ($t_1 - t_0$) Average (μs)	Min (μs)	Max (μs)	Jitter (μs)
16	1.51	3.17	4.68	4.39	4.86	0.47
24	1.94	3.48	5.42	5.13	5.60	0.47
32	2.38	3.91	6.29	6.00	6.47	0.47
40	2.81	4.29	7.10	6.81	7.28	0.47
48	3.24	4.67	7.91	7.61	8.08	0.47
64	3.67	5.25	8.92	8.63	9.10	0.47

The timestamp points just introduced are suitable for a variety of measurements, concerning both correctness and performance. In the following, the discussion will be focused only on correctness, whereas performance-related measurements will be discussed in Section IV-C.

When correctness is considered, the difference $t_2 - t_1$ is very important, because it represents the actual end-to-end CAN frame transfer time. The results are shown in Fig. 5 and Table II. In the figure, the minimum and maximum value of $t_2 - t_1$ measured for 10^7 random test data items corresponding to a given value of n_{D} are plotted. They have been normalized to bit periods, for easier comparison with the results of software-based correctness check in Fig. 4. The table contains more detailed information about the residual jitter, in numeric form.

Experimental results further confirm the correctness of the ZSC proposal. As it can be seen by comparing Figs. 4 and 5, the actual end-to-end CAN frame transfer time is always the same as the theoretical value. At the same time—as listed in Table II—the residual jitter is well below one bit time (that corresponds to $2\,\mu$s in the experiments being discussed). Hence, it cannot be caused by the bit stuffing mechanism, which would introduce an amount of jitter that is a multiple of the bit time itself. Rather, this value is totally compatible with what was shown to be the residual jitter at the interface between the CPU and CAN controller on the same kind of hardware [18] and using the same controller interface. It is worth remarking that, as mentioned in the same paper, this jitter is neither bit-rate dependent nor n_{D} dependent. It could probably be further reduced by upgrading the hardware.

C. Performance Evaluation

Performance-oriented measurements were focused on the ZS encoding and decoding time, as they are interesting from the application viewpoint. They correspond to $t_1 - t_0$ and $t_3 - t_2$ in Fig. 3, respectively. Moreover, the contribution to them made by the 8B9B and ZSC modules was evaluated as well.

For what concerns decoding, as discussed before, the ZSC decoding layer is very thin and it simply discards the tuning field before feeding the incoming data to the 8B9B decoder. Consequently, the performance of the ZS decoder is very close to the 8B9B decoder alone, which has already been thoroughly evaluated in [16]. Experimental data also shows that the extra delay introduced by ZSC decoding is always fixed and negligible (just a couple of CPU clock cycles). Overall, the ZS decoding time $t_3 - t_2$ is about $3.42\,\mu$s at the maximum DLC on the platform being considered in this paper.

One crucial point of the former evaluation [16] was to show that the 8B9B decoder works in *constant* time for a given DLC, and hence, it does not introduce any communication jitter. As mentioned above, the ZSC decoding module does not introduce any other source of jitter either. Hence, the execution of ZS decoding module as a whole is jitterless.

Fig. 6 and Table III show the experimental results concerning $t_1 - t_0$, that is, the total ZS encoding time. For better evaluation, $t_1 - t_0$ has also been broken down into the basic 8B9B encoding time plus the additional time needed for ZSC encoding. Since the 8B9B encoder is completely jitterless [16], all the jitter encountered in overall ZS encoding is due to ZSC. For clarity, the error bar is not shown for ZSC encoding delay.

As it can be seen from Fig. 6, both components of the

encoding delay are linear with respect to n_{D} for $n_{\mathrm{D}} \leq 48\,\mathrm{b}$. As a consequence, the ZS encoding time also preserves linearity in those cases, as shown in the figure. The difference in the slope between 8B9B and ZSC encoding can be easily justified by considering that in the first case, it depends on the 8B9B encoding loop, whereas in the second case it depends on the $\mathrm{R_L}$ calculation loop, which is simpler. In addition, as it can be seen from the figure, ZSC encoding is always slower than 8B9B encoding because ZSC also includes a fixed and n_{D}-independent calculation of the tuning field (T).

It is worth mentioning that, when $n_{\mathrm{D}} \leq 48\,\mathrm{b}$, the data field after encoding is 1 byte longer than the original payload size. Instead, when $n_{\mathrm{D}} = 64$, it is 2 bytes longer since one byte is used to give enough room for tuning in the ZS encoding as explained in Section IV-B. However, the number of iterations of the byte-by-byte 8B9B encoding loop just depends on the size of the original payload. That explains why 8B9B encoding time in ZS encoder is *faster* than linear when $n_{\mathrm{D}} = 64$.

Similarly, when $n_{\mathrm{D}} = 64$, ZSC encoding is also *faster* than linear, however, for a different reason. In the actual implementation, before calculating $\mathrm{R_L}$ and T, generally some checks on n_{D} need to be done. The compiler optimizer is able to omit this kind of check when $n_{\mathrm{D}} = 64$.

Since ZS is made up of 8B9B and ZSC, these observations also explain why ZS is *faster* than linear in the case $n_{\mathrm{D}} = 64$.

The experimental data shown in Table III also indicates that the jitter encountered in software encoding is always less than one bit time. More importantly, it is not n_{D} dependent. In fact, further investigation revealed that this jitter comes from step 16) of the algorithm shown in Fig. 2. This is because bit stuffing check is only required for $\mathrm{T}_j \wr \mathrm{R}_j$ instead of the whole frame as already explained in Section II-E and it is done in two steps in the real implementation, by first checking the existence of primer sequence(s) of all 1s and then all 0s.

However, the compiler optimized the code in a way that if a primer sequence of 1s is found, it will silently skip the check for 0s. Most likely, the programmer might not notice this optimization. In any case, this introduces uncertainty into software execution time. With little effort, this source of jitter can be dramatically reduced to 2 clock cycles (corresponding to $28\,\mathrm{ns}$) with a delay penalty of $1.49\,\mu\mathrm{s}$ at most.

Without this extra jitter reduction, the ZS encoding time is about $8.92\,\mu\mathrm{s}$ at the maximum DLC. This means that the worst-case delay introduced into the communication by ZS is around $12.34\,\mu\mathrm{s}$, among which $3.42\,\mu\mathrm{s}$ (28%) is for ZS decoding, $3.67\,\mu\mathrm{s}$ (30%) is for 8B9B encoding and $5.25\,\mu\mathrm{s}$ (42%) is for ZSC encoding. This is quite acceptable, as it is well below the minimum frame transmission time $t_2 - t_1$, which is about $123.16\,\mu\mathrm{s}$ as shown in Table II. What's more, the ZS encoding/decoding time could probably be reduced by adopting either a better CPU or a higher clock frequency.

D. Memory requirement

Memory requirement is an important design constraint in low-end embedded systems as generally they are short of on-chip memory. Resorting to external memory will probably increase the cost and complexity for development whereas the real-time performance may be impaired somehow [16].

TABLE IV
MEMORY REQUIREMENT

Segment	8B9B (B)	ZSC (B)	ZS (B)
text	398	898	1296
read-only data	384	512	896
bss	n/a	12	12
stack	60	8	68

Table IV summarizes the memory footprint of the ZS codec and its two components, namely 8B9B and ZSC. For what concerns the ZSC modules, the most significant contributions to footprint are given by $898\,\mathrm{B}$ of code in the *text* segment, mainly for ZSC encoding, plus $512\,\mathrm{B}$ of *read-only data* used for the CRC lookup table mentioned in Section IV-A. On the other hand, the requirement of ZSC in the *bss* (uninitialized data) and *stack* segments amounts to only $20\,\mathrm{B}$ in total, and hence, is minimal.

Generally speaking, both the text and the read-only data segments are stored in (non-volatile) FLASH memory, while uninitialized data and stack take space in RAM memory. As a result, the ZS modules overall require $2192\,\mathrm{B}$ of FLASH memory and $80\,\mathrm{B}$ of RAM. Considering that $512\,\mathrm{KB}$ of FLASH memory and $96\,\mathrm{KB}$ of RAM are available on the microcontroller used in the experiments, this is quite acceptable. It should also be noted that the extra space needed for ZSC mostly concerns FLASH memory, which is generally more abundant than RAM on this kind of systems.

V. CONCLUSIONS

In this paper a mechanism has been presented, known as ZSC, that allows to prevent stuff bits in the CRC of CAN frames. When coupled with a ZSD encoding scheme, like 8B9B, this prevents the insertion of stuff bits all over the frame except for the header. As mentioned in Section II-E, stuff bits in the header do not lead to any jitter and, at the end, ZSC is indeed able to achieve truly jitterless communication (provided that CAN arbitration is not exploited). The advantage of this approach, when compared to other interesting solutions such as the one in [22], is that no modification is required to the hardware. On the contrary, conventional CAN controllers can be used, by placing over them a tiny software layer that carries out the ZSD and ZSC functions.

A ZS codec has been developed as a part of our activities, that features small footprint and low processing times. Experiments carried out on large sets of data show that the residual jitter can be at least 40 times lower than in plain CAN. As a consequence, the proposed solution is proved to be able to support razor-sharp actuation in CAN systems even when implemented in software.

It is worth remarking that ZSC operates mostly on the transmitting side. Receivers must mainly be able to decode the leading part of the frame by means of the ZSD codec, because ZSC decoding just discards the tuning field before the data field is passed to the ZSD decoder. This means, that performance penalties are only found on transmitters. At the limit, simple nodes that only have to receive jitter-free messages, are allowed not to implement ZSC at all. In

the case of master-slave systems, this permits lowering the implementation cost of slaves noticeably.

Besides a drastic reduction of transmission jitters, ZS encoding decreases noticeably the residual error probability, too. This is because bit stuffing may interfere with CRC error detection capability [23]. In [24] the effect of ZSD on data integrity was evaluated, and was found able to offer very encouraging results (undetected errors lowered by about two orders of magnitude). As part of our future work, we plan to assess to which extent the inclusion of ZSC, which prevents stuff bits completely, can bring further improvements in this direction. It is easy to see that the adoption of inexpensive ZS codecs in the automotive industry may affect tangibly the lifetime of legacy CAN controllers in such a scenario.

REFERENCES

[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] D. Gessner, M. Barranco, and J. Proenza, "Design and verification of a media redundancy management driver for a CAN star topology," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 237–245, Feb. 2013.

[3] R. A. Gupta and M.-Y. Chow, "Networked control system: Overview and research trends," *IEEE Trans. Ind. Electron.*, vol. 57, no. 7, pp. 2527–2535, Jul. 2010.

[4] L. Zhang, H. Gao, and O. Kaynak, "Network-induced constraints in networked control systems—a survey," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 403–416, Feb. 2013.

[5] Y. Xia, J. Yan, P. Shi, and M. Fu, "Stability analysis of discrete-time systems with quantized feedback and measurements," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 313–324, Feb. 2013.

[6] H. Zeng, M. Di Natale, P. Giusto, and A. Sangiovanni-Vincentelli, "Using statistical methods to compute the probability distribution of message response time in Controller Area Network," *IEEE Trans. Ind. Electron.*, vol. 6, no. 4, pp. 678–691, Nov. 2010.

[7] P. Martí, A. Camacho, M. Velasco, and M. El Mongi Ben Gaid, "Runtime allocation of optional control jobs to a set of CAN-based networked control systems," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 503–520, Nov. 2010.

[8] M. Hu, J. Luo, Y. Wang, M. Lukasiewycz, and Z. Zeng, "Holistic scheduling of real-time applications in time-triggered in-vehicle networks," *IEEE Trans. Ind. Informat.*, vol. 10, no. 3, pp. 1817–1828, Aug. 2014.

[9] W. Hofer, D. Danner, R. Muller, F. Scheler, W. Schroder-Preikschat, and D. Lohmann, "Sloth on time: Efficient hardware-based scheduling for time-triggered RTOS," in *Proc. 33rd IEEE Real-Time Systems Symp. (RTSS)*, Dec 2012, pp. 237–247.

[10] T. Nolte, H. Hansson, and C. Norström, "Minimizing CAN response-time jitter by message manipulation," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.*, Sep. 2002, pp. 197–206.

[11] G. Cena, I. Cibrario Bertolotti, T. Hu, and 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*, Sep. 2012, pp. 1–8.

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

[13] M. Nahas and M. Pont, "Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study," in *Proc. Second UK Embedded Forum*, Oct. 2005, pp. 4–17.

[14] M. Nahas, M. J. Pont, and M. Short, "Reducing message-length variations in resource-constrained embedded systems implemented using the CAN protocol," *J. of Systems Architecture*, vol. 55, no. 5–6, pp. 344–354, May 2009.

[15] M. Nahas, "Applying eight-to-eleven modulation to reduce message-length variations in distributed embedded systems using the Controller Area Network (CAN) protocol," *Canadian J. on Electrical and Electronics Engineering*, vol. 2, no. 7, pp. 282–293, Jul. 2011.

[16] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "Fixed-length payload encoding for low-jitter Controller Area Network communication," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2155–2164, Nov. 2013.

[17] ——, "On a family of run length limited, block decodable codes to prevent payload-induced jitter in Controller Area Networks," *Computer Standards & Interfaces*, vol. 35, no. 5, pp. 536–548, Sep. 2013.

[18] ——, "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*, Sep. 2012, pp. 1–8.

[19] ——, "Software-based assessment of the synchronization and error handling behavior of a real CAN controller," in *Proc. 18th IEEE Conf. on Emerging Technologies and Factory Automation*, Sep. 2013, pp. 1–9.

[20] *LPC24XX User manual, UM10237 rev. 2*, NXP B.V., Dec. 2008.

[21] *LPC2468 Product data sheet, rev. 4*, NXP B.V., Oct. 2008.

[22] I. Sheikh, M. Hanif, and M. Short, "Improving information throughput and transmission predictability in Controller Area Networks," in *Proc. IEEE International Symp. on Industrial Electronics*, Jul. 2010, pp. 1736–1741.

[23] J. Charzinski, "Performance of the error detection mechanisms in CAN," in *Proc. 1st International CAN Conference (iCC 1994)*, Sep. 1994, pp. 20–29.

[24] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "Effect of jitter-reducing encoders on CAN error detection mechanisms," in *Proc. 10th IEEE International Workshop on Factory Communication Systems (WFCS)*, May 2014, pp. 1–10.

Gianluca Cena (SM'09) received the Laurea degree in electronic engineering and the Ph.D. degree in information and system engineering from the Politecnico di Torino, Turin, Italy, in 1991 and 1996, respectively.

In 1995, he became an Assistant Professor with the Department of Computer Engineering, Politecnico di Torino. Since 2005 he has been Director of Research with the Institute of Electronics, Computer and Telecommunication Engineering of the National Research Council of Italy (CNR–IEIIT), where he is engaged in research activities concerning industrial communications and real-time networks. In these areas, he has coauthored more than 100 technical papers.

Prof. Cena served as Program Co-Chairman for the 2006 and 2008 editions of the IEEE Workshop on Factory Communication Systems and has been Associate Editor of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS since 2009.

Ivan Cibrario Bertolotti (M'06) received the Laurea degree (*summa cum laude*) in computer science from the University of Torino, Turin, Italy, in 1996.

Since then, he has been a Researcher with the National Research Council of Italy (CNR). Currently, he is with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Turin, Italy. He has taught several courses on real-time operating systems at Politecnico di Torino, Turin, Italy, has co-authored a book on the same topics, and serves as a technical referee for primary international conferences and journals. His current research interests include real-time operating system design and implementation, industrial communication systems and protocols, and formal methods for vulnerability and dependability analysis of distributed systems.

Tingting Hu (M'11) received the MS degree in computer engineering from Politecnico di Torino, Turin, Italy, in 2010, where she is working toward the Ph.D. degree in control and computer engineering.

Since then, she has been a Research Fellow with the National Research Council of Italy (CNR). Currently, she is with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Turin, Italy. Her primary research interests concern design and implementation of real-time operating systems and communication protocols. She serves as technical referee for several primary conferences in her research area.

Adriano Valenzano (SM'09) received the Laurea degree in electronic engineering from Politecnico di Torino, Torino, Italy, in 1980.

He is Director of Research with the National Research Council of Italy (CNR). He is currently with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Torino, Italy, where he is responsible for research concerning distributed computer systems, local area networks, and communication protocols. He has coauthored approximately 200 refereed journal and conference papers in the area of computer engineering.

Dr. Valenzano is the recipient of the 2013 IEEE IES and ABB Lifetime Contribution to Factory Automation Award. He also received, as a coauthor, the Best Paper Award presented at the Fifth and Eighth IEEE Workshops on Factory Communication Systems (WFCS 2004 and WFCS 2010). He has served as a technical referee for several international journals and conferences, also taking part in the program committees of international events of primary importance. Since 2007, he has been serving as an Associate Editor for the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.