Fixed-Length Payload Encoding for Low-Jitter Controller Area Network Communication

Fixed-Length Payload Encoding for Low-Jitter Controller Area Network Communication

Gianluca Cena, Ivan Cibrario Bertolotti, Tingting Hu, and Adriano Valenzano

***Abstract*—The controller area network (CAN) bit stuffing mechanism, albeit essential to ensure proper receiver clock synchronization, introduces a significant, payload-dependent jitter on message response times, which may worsen the timing accuracy of a networked control system. Accordingly, several approaches to overcome this issue have been discussed in literature. This paper presents a novel software payload encoding scheme, which is able to guarantee that no stuff bits will ever be added to the data field by the CAN controller during transmission and, hence, lessens jitters considerably. Particular care has been put in its practical implementation and its subsequent evaluation to show how the simplicity and inherent high performance of the scheme make it suitable even for low-cost, embedded architectures.**

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

I. Introduction

SINCE it was introduced about 20 years ago, controller area networks (CANs) [1] have steadily gained popularity and are nowadays adopted in a variety of embedded, networked control systems. One peculiar facet of the CAN protocol is its *bit-stuffing* (BS) mechanism, which is an efficient way to ensure that a sufficient amount of edges appear in the signal sent over the bus, and thus, guarantee a proper receiver clock synchronization. However, due to BS, the actual length of a message sent over the bus depends not only on the *size* of its payload but also on its *content*. As a consequence, the exact duration of frame transmissions cannot be known in advance, leading to a certain degree of jitter on response times. Also for this reason, recent automotive protocols such as FlexRay [2] rely on fixed-length encoding instead.

Communication jitter may be, at least in some demanding cases, a limiting factor in the design of a networked control system, both in general [3]–[7], and for CAN in particular [8]–[13]. Even in the case of simple systems that rely on the master–slave approach, the nonconstant duration of CAN messages leads to annoying jitters on actuation. TTCAN controllers [14] overcome this drawback but in this case applications ought to be redesigned in order to comply with the time-triggered paradigm.

Although new-generation isochronous protocols—e.g., FlexRay and some real-time Ethernet solutions—are the best option for a new design of a high-performance dependable control system [15], CAN is still a viable option in many other cases [16]. In fact, CAN technology is undoubtedly really stable, well known, and widespread. For these reasons, most microcontroller families from all major vendors include at least one member that embeds, at no extra cost, a CAN controller. As a consequence, when design constraints are not so tight and the focus is instead on reducing system cost, complexity, and time-to-market, many designers still prefer to stick to CAN.

Recently, some researchers started considering the use of conventional CAN also for applications with tight timing constraints, as witnessed by some proposals made for software-based synchronization techniques in CAN. However, a high degree of determinism can be obtained in a distributed system only if the duration of frame transmissions over the network is fixed and known in advance (unless a time-triggered approach is followed). As discussed above, this is not the case in CAN. The common rationale behind all solutions to this issue that have been proposed in the past [8], [9], [17]–[20] is to encode the payload of the frames in software, so that the hardware insertion of stuff bits in this part of the frame by the CAN controller is either reduced or prevented completely. Using the same principle, and building on a preliminary version presented in [21], this paper discusses a new fixed-length encoding scheme, called *8B9B*. It is characterized by an encoding efficiency comparable to, or better than, most other proposals, and it is able to completely prevent bit stuffing in the CAN data field. Even more importantly, since *8B9B* independently encodes every single byte in the original payload, it is amenable to a portable, very compact, and fast implementation, realized by means of a high-level programming language. Regarding these aspects, it outperforms the other techniques described in the literature.

This paper is structured as follows. In Section II, the problem of jitters in CAN caused by BS is briefly described and the *8B9B* encoding technique is introduced, whereas in Section III some details are provided about the development of a portable *8B9B* codec and its optimization. Then, in Section IV, the corresponding experimental results on two dissimilar microcontrollers are presented. Section V contains a performance evaluation of the codec, including a comparison with related work, and Section VI draws some concluding remarks.

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 Turn, Italy, and also with the Dipartimento di Automatica e Informatica, Politecnico di Torino, I-10129 Turin, Italy.

Fig. 1. CAN frame format, with an 11-bit identifier.

II. Reducing Stuff Bits in CAN

The physical layer of CAN relies on the *nonreturn to zero* encoding scheme with bit stuffing. Every time the CAN controller in the transmitting node detects that five consecutive bits with the same value have been sent, it inserts a *stuff bit* at the opposite level. For example, if the original sequence is `01111101 00000001`..., the sequence of bits sent on the bus will be `011111`$\underline{\texttt{0}}$`01 00000`$\underline{\texttt{1}}$`001`... (from now on, underlining is used to denote stuff bits). Bit stuffing applies only to those fields in a CAN frame from the start of frame bit (SOF) up to the cyclic redundancy check (CRC) included. The remaining fields, from the CRC delimiter up to the end of the frame (EOF), are of fixed form and are not affected (see Fig. 1).

The rationale behind BS lies in its higher efficiency when compared to fixed-length encoding schemes, e.g., Manchester and 4B5B (adopted in Ethernet). The worst case for the encoding efficiency of BS occurs when the bitstream to be sent consists of, e.g., 5 b at `0`, followed by a repeated pattern made up of 4 b at `1` followed by 4 b at `0`, that is, `00000`$\underline{\texttt{1}}$`1111`$\underline{\texttt{0}}$`0000`$\underline{\texttt{1}}$... [9] (in the following, sequences of 5 b at the same value, either `0` or `1`, are referred to as *primer sequences*). This means that the maximum number $N_{\max}$ of stuff bits that, in theory, can be added to a frame in CAN, is equal to $\lfloor (34+8s-1)/4 \rfloor = 8+2s$, where s is the size in bytes of the payload. In practice, $N_{\max}$ is always one or two bits less than the above theoretical value, since part of the header has a fixed value (e.g., SOF and res bits) while DLC is closely related to the payload.

As pointed out in [9], in real CAN networks the mean number of stuff bits actually inserted in messages is much lower than $N_{\max}$. While this is surely a benefit for the average frame transmission times, which are typically quite close to their minimum value $C_{\min} = (44+8s)t_{\text{bit}}$, where t_{bit} is the bit time, BS causes jitters on the expected reception times. The actual transmission time C for a message lies in the range $C_{\min} \leq C \leq C_{\min} + J_{\text{C}}$, where J_{C} is the worst-case "bit-level" jitter the message may suffer because of BS, that is, $J_{\text{C}} = N_{\max} t_{\text{bit}}$.

When CAN is used to support event-driven communications, as in the automotive domain, in normal operating conditions J_{C} is usually much lower than the "frame-level" jitter J_{R} that may affect response times, which is mostly due to the non-preemptive priority-based CAN access scheme (bitwise arbitration). At the frame level, the worst-case response time R for any message can be evaluated through schedulability analysis [22] and can last many frame times, whereas in the best case (when the bus is idle) the frame is sent immediately and received completely after a time equal to C. Seemingly, the contribution of BS to the overall jitter could be neglected in these cases. However, a change in the effective duration C of any frame affects both the blocking time for higher priority messages and the interference on the lower priority ones. Therefore, bit-level jitters do indeed affect frame-level jitters as well. In order to prevent such a dependency, schedulability analysis techniques always assume that the maximum number of stuff bits is added to every message.

When CAN is used to implement tightly synchronized distributed systems [8], jitters caused by BS can be even more annoying. In these cases, suitable techniques are usually adopted in order to avoid frame-level jitters. A simple approach is using very-high-priority frames for time-critical messages: when the highest priority CAN identifier is used for this aim, J_{R} decreases to just one frame time at worst. In the case time-division multiple access (TDMA) is exploited, as in TTCAN [14], messages are assigned disjoint slots, hence making arbitration unnecessary. As a simpler alternative, bus contentions can be avoided by using CAN according to a *master–slave* approach. In this kind of system, countermeasures are usually required in order to reduce bit-level jitters as well, since message reception times should not depend on the specific values carried in the payload. Additional hardware and software jitters introduced by the CAN controller and protocol stack [23] are not taken into account explicitly in the following, since they do not depend on the protocol and cannot be tackled by means of suitable encoding techniques.

A. Previous Solutions

As several literature papers point out, jitters in CAN, which are due to BS, can be reduced by modifying the payload on the transmitter side so as to remove primer sequences. This process must be revertible on the receiver side. For instance, the XOR technique in [9] carries out the exclusive OR (EXOR) between the original payload and a specific pattern consisting of an alternating bit sequence. Although XOR reduces the likelihood that stuff bits are inserted in the data field—especially when real data are taken into account—there is no guarantee that they are completely prevented.

This approach was enhanced in [8], [18], where EXOR was applied selectively to either the payload as a whole (*SXB*) or on single bytes separately (*SXB*). These techniques were formerly denoted Nolte B and C, respectively. Despite achieving reduced jitter (especially *SXB*), encoding efficiency decreases because of the need to include information about the EXOR process (whether or not, and where to apply it) while encoding time increases. Moreover, some stuff bits may still be added to the data field nevertheless.

Other techniques were subsequently introduced, which are able to prevent the insertion of stuff bits in the data field completely. As an example, the *SBS* approach [19] performs a software bit stuffing in advance on the original payload, so that one software stuff bit is added every time 4 consecutive bits are found at the same value. Processing time increases consequently with respect to *SXP* and *SXB*. In a similar way, *EEM* is a fixed-length 8-to-11 modulation scheme [20], which encodes every byte in the original payload as an 11-bit pattern and proved to be faster than *SBS*.

To the best of our knowledge, all of the existing approaches only tackle the data field, and hence, they are unable to prevent stuff bits in other parts of the frame.

B. 8B9B Encoding

The *8B9B* encoding scheme is straightforward. Basically, every single byte of the original payload is translated separately to a suitable pattern made up of 9 b. The encoded bit stream is then obtained by concatenating all these patterns in the original

order. Clearly, this does not apply to frames with an empty payload (i.e., when DLC is 0). Encoded 9-b patterns must satisfy two properties, given here.

1) None of the patterns is allowed to include primer sequences. For instance, the pattern 010000011 is unsuitable for our purposes.
2) Primer sequences must never appear in the encoded bit stream, not even across pattern boundaries. In order to meet this additional constraint, all patterns that include 3 (or more) b at the same level, at either the beginning or the end, have to be discarded as well. For instance, the pattern 010101000 is not suitable because, when followed by the (legitimate) pattern 001010101, it would give rise to the sequence 010101000 001010101, which includes a primer sequence.

A simple algorithm was developed to find exhaustively all the 9-b patterns that satisfy both of the above constraints. The resulting set of candidates includes 258 different elements. This confirms that every single-byte value (from 0 to 255) can be encoded by means of a unique 9-b pattern. To this aim, 256 patterns were reserved. Two additional patterns exist in the set, which are not used to encode any data byte value. They were labeled J (001000010) and K (110111101) and could be adopted as escape sequences, but this aspect falls beyond the scope of this paper.

Basically, the above ordered set of patterns represents a *forward lookup table* (FLT) for the direct replacement of bytes in the original payload with corresponding 9-b patterns. Let X be the original data byte, Y the corresponding *8B9B*-encoded value, and $F(\cdot)$ the encoding function carried out through the FLT. The forward translation process can be synthetically expressed as $Y = F(X)$.

Because of the properties the patterns which were selected satisfy, if Y is a valid pattern also $\sim Y$ is necessarily valid, where "$\sim$" represents the *complement* operator (bitwise NOT), which inverts every single bit in the pattern it is applied to. Since candidate patterns were obtained in increasing numerical order, a basic property of the binary representation of numbers (that is, $2^9 - 1 - Y = \sim Y$) implies that, overall, the FLT is "specular." In formulas, we have

$$Y = F(X) \Leftrightarrow \sim Y = F(\sim X). \tag{1}$$

By exploiting this property, only the first half of the FLT is required to be stored in real implementations (see Table I).

TABLE I
8B9B FORWARD LOOKUP TABLE (ENCODING PROCESS)

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

C. Break Bit and Padding Field

The reasoning above takes into account the data field alone. Unfortunately, the preceding parts of the frame may contribute to the creation of a primer sequence together with the beginning of the data field. The DLC field is not in complete control of the user, since it specifies the size in bytes of the payload—represented on 4 b as a binary value. For this reason, unlike the payload, it cannot be encoded. For example, when the DLC value is 8 (1000) and the first encoded pattern is 001010101, the resulting (partial) bit sequence is . . . 1000 001010101 . . ., which includes a primer sequence.

A simple remedy is inserting a single bit, called *break bit* (BB), in the very first position of the data field. The value of BB is opposite to the last DLC bit, i.e., it is either 1 when the DLC value is even or 0 on the contrary. Thanks to BB, bit strings with two or more bits at the same value cannot appear immediately before the first 9-b pattern in the encoded payload, for any value of DLC between 1 and 8, included. Only when DLC $= 15$, the BB could be preceded by one stuff bit at its same level, appended to the DLC by the CAN controller. In any case, the occurrence of a primer sequence affecting the data field is completely precluded.

In theory, BB is mandatory only when the DLC is 3 (0011), 7 (0111), or 8 (1000). The reason why it is required also in the first case is that a stuff bit at 1 could be possibly inserted just after the 00 bit pair, hence giving rise to the sequence 00111. In practice, two choices are available for the codec: either BB is included only when strictly required or it can be maintained for any frame size. In this paper we opted for the latter choice, since it does not worsen efficiency appreciably and makes the codec simpler and faster.

A second aspect to be considered is that the bit sequence obtained by the *8B9B* translation process is generally not aligned to a byte boundary. As a consequence, the last byte in the data field may not be filled up completely by actual data (i.e., the encoded payload). In this case, a variable-size *padding* field (PAD) is foreseen to align the encoded bit stream to the next 8-b boundary. The transmitting node sets PAD to a suitable value that does not cause the insertion of any stuff bit, e.g., an alternating bit pattern (0101 . . .).

D. Other Fields of the Frame

Only the data field is covered by *8B9B*. There are, however, other parts of the CAN frame that are subject to BS too, namely the frame header and CRC.

The header in CAN frames is basically made up of the message identifier, DLC and few additional bits whose value is fixed (e.g., SOF and res). Stuff bits are typically not a problem here,

TABLE II
8B9B-ENCODED DATA FIELD VERSUS ORIGINAL PAYLOAD

Original payload size (byte)	Data field in the *8B9B*-encoded frame: size (byte)	DLC val.	BB val.	BB size (b)	9-bit pattern sequence (b)	PAD size (b)
0	0	0000	—	0	0	0
1	2	0010	1	1	9	6
2	3	0011	0	1	18	5
3	4	0100	1	1	27	4
4	5	0101	0	1	36	3
5	6	0110	1	1	45	2
6	7	0111	0	1	54	1
7	8	1000	1	1	63	0
8	—	—	—	—	—	—

since in real-world applications both the message identifier and the payload size for any given message stream are usually not allowed to change over time. As a consequence, the number of stuff bits that are added by the CAN controller to the frame header is fixed and known in advance by the system designer. A number of design hints on this subject are reported in [17].

The CRC field follows immediately the data field and is the last part of the frame to which BS applies. Unlike the preceding fields, for which countermeasures exist to prevent jitters, there is no simple remedy to avoid the insertion of stuff bits in the CRC, because it is computed in hardware by the CAN controller at runtime. No more than four stuff bits can be added in the CRC when using *8B9B*. The worst case consists of the sequence ...000 00$\underline{1}$1111$\underline{0}$0000$\underline{1}$1111$\underline{0}$0... In fact, the first three bits can be found at the end of the data field (at most 2 b at the same value are allowed at both ends of 9-b patterns, but they can be equal to the single padding bit at the end of the data field when $\mathrm{DLC} = 7$), whereas the following 15 b correspond to a plausible CRC. This is enough for causing one primer sequence and three 4-b subsequences, which, because of the domino effect, give rise to four stuff bits.

E. Encoding Example

Let us consider a very simple one-byte datum at the value `0xf0`. In conventional CAN, this value would fit directly into a single-byte data field ($\mathrm{DLC} = 1$), and would result in the transmitted (partial) sequence ...0001 1111$\underline{0}$0000$\underline{1}$... (only the DLC and data fields are shown). As can be seen, two stuff bits have been inserted.

Table II shows that the DLC value in the corresponding *8B9B*-encoded frame is 2 (`0010`). As the DLC is even, BB is set to `1`, which is placed as the starting bit of the data field. Then, every byte in the payload (only one in this case) is translated. According to (1), byte `0xf0` is first complemented (it starts with a bit at `1`), which yields `0x0f`; a lookup operation on the FLT in Table I returns the 9-b pattern `001010110`, which, when complemented, yields `110101001`. Finally, a padding is appended to the end of the *8B9B*-encoded payload, which consists of 6 b set to the alternating bit pattern mentioned before. Overall, the transmitted (partial) sequence is ...0010 1 110101001 010101..., where the DLC, BB, encoded payload (one pattern) and PAD are shown. As expected, no additional stuff bit is inserted into the data field when the frame is transmitted.

Incidentally, *8B9B* is also able to improve the error detection capabilities of CAN slightly. In fact, no invalid pattern should be found in the received payload in normal operating conditions: their presence is evidence that a transmission error has occurred. This constitutes an additional error detection mechanism and increases reliability further at the expense of an extra decoding overhead.

III. IMPLEMENTATION AND OPTIMIZATION

The encoding and decoding algorithms discussed in the previous section have been implemented in the ISO C language [24] and then cross-compiled, using different toolchains, for two dissimilar architectures. Those architectures have been chosen because they are at the extremes of the range of microcontrollers currently adopted for CAN-based applications.

Namely, the NXP LPC2468 microcontroller [25] (called `arm7` in the following) is a low-end component based on the ARM7TDMI-S processor core designed in 2001 and running at 72 MHz. Instead, the NXP LPC1768 microcontroller [26] (`cm3` in the following) is based on the contemporary ARM Cortex-M3 processor core running at 100 MHz.

The base version of the C code has then been optimized in several ways, better discussed in the following, by working at both the toolchain and source code levels. Neither handwritten nor hand-optimized machine code has been developed, because one of the implementation goals was to show that it is possible to achieve good optimization results across different processor architectures—by working exclusively at the C language level and without necessarily having a thorough knowledge of the processor machine language level.

For both platforms, the toolchain consists of open-source components only, namely: `binutils` (assembler, linker, librarian), the `gcc` compiler collection, and the `newlib` runtime library. Platform initialization files and linker scripts have been taken from the FreeRTOS [27] operating system.

A. Determinism and Performance Optimization

The first optimization goal was to achieve full execution time *determinism*, that is, make the execution time of the encoding/decoding algorithms independent from message contents. After all, if full determinism were impossible to achieve, the original source of jitter (CAN bit stuffing) would be simply replaced by another one (data processing jitter due to the encoding/decoding layer, called J_P in the following). On the other hand, a (hopefully linear) dependency on the payload size s is both expected and acceptable.

For this purpose, the code has been reworked in two different ways. First, all conditional *statements* have been replaced by conditional *expressions*, in order to make their execution time independent from predicate truth. For instance, the execution time of a conditional statement in the form `if (f < 0) f = ~f` depends on the sign of `f`, because the assignment and the bitwise complement are performed only if `f` is negative.

On the contrary, assuming that `f` is an 8-b variable, the execution time of `m = ((f < 0) ? 0x00 : 0xFF)` is constant, because both sides of the conditional expression have got the same structure. Then, the expression `m^f` can subsequently be used to obtain either `f` itself or the bitwise complement of `f` depending on its sign, again, in constant time.

Moreover, by leveraging a property of the *8B9B* encoding and decoding algorithms, it was possible to write down the code so

that all iteration statements are executed a number of times that only depends on the payload size, rather than contents. All in all, after both optimizations have been carried out, the code contains just one loop for the encoder and one for the decoder, and there is a single execution path within these loops. Both loops are always executed once for each payload byte to be encoded and decoded.

The second goal of the optimization was to improve code *performance* as much as possible to minimize the additional end-to-end response time introduced by the encoding and decoding process. Three kinds of optimization have been identified and implemented.

1) Code/Data Placement: Microcontrollers usually support different kinds of memory such as Flash memory, dynamic RAM (DRAM), and static RAM (SRAM). Each kind of memory has its own peculiar behavior in terms of access speed and determinism. The goal of this optimization step was to force the toolchain to place the code, data, and stack segments of the encoding and decoding routines in the most appropriate place. Even if this is a relatively straightforward decision, most toolchains are unable to take it autonomously. Moreover, as it will be better discussed in Section IV, their default placement rules are far from being optimal.

2) Computed Masks: In the base implementation shown in [21], several auxiliary arrays hold the bit masks used on every iteration of the encoding and decoding loop, to split the 9-bit value obtained from the forward lookup table into two output bytes during encoding, and join them again during decoding. In this optimization, the masks have been computed directly, as a function of the loop index. Since this computation can be carried out in the processor registers, the extra memory accesses to the arrays are avoided. The arrays themselves can be deleted to save memory, too.

3) Load/Store Reduction: The base version of the encoding algorithm, on each iteration, loads one input byte and stores some data into two adjacent output bytes. One of the output bytes overlaps with those of the previous iteration, and hence, each output byte except the first one is accessed twice. With this optimization, a local buffer has been introduced to carry forward the common output byte from one iteration to the next, in order to store it only once. The decoding algorithm has been optimized in the same way, too.

Since these optimizations are independent from each other, each optimization step started from the previous one.

B. Memory Footprint Optimization

The starting point of the memory footprint optimization is the basic symmetry property of the FLT equation (1). With the help of this property, as stated in Section II, it is possible to store only half of the table and reduce its size from 256 to 128 9-b entries. For efficient access, the size of each FLT entry must nevertheless be an integral number of bytes, and hence, each entry actually occupies 16 b in memory. However, the most significant bit of the FLT is always zero in the first half of the table, as shown in Table I, whereas it is always one in the second half. Therefore, it is not necessary to store this bit explicitly and the FLT can be further shrunk down to 128 8-b entries, that is, one quarter of its original size.

The quickest method to revert the *8B9B* encoding scheme on the receiver side is to use a reverse lookup table (RLT) that represents the inverse of the encoding function $F(X)$. Since F is injective, but not surjective, not every 9-bit pattern actually corresponds to a valid encoder output. As a consequence, each RLT entry must store two distinct pieces of information: an indication of whether or not the given value of Y is valid, and the original data byte, that is, the value of X for the given Y, if any. The entry size is therefore 9 b, 1 b for the flag, and 8 b for X, corresponding to 16 b in memory. In turn, the total size of the full RLT is 512 16-b entries.

As for the FLT, the RLT size can be halved by exploiting symmetry, which means that 256 entries are sufficient. Moreover, when doing so, only the seven least significant bits of each X must be stored explicitly, because the most significant bit of X can be inferred from the most significant bit of Y. Overall, the RLT can hence be reduced to 256 8-b entries. Of these, 128 entries correspond to either invalid patterns or the J and K escape codes.

Further memory savings would be possible, by noticing that the numerically lowest valid pattern in the halved FLT is `001000011` (67_{10}) while the highest is `011110110` (246_{10}). This means that only 180 RLT entries must actually be stored in memory, whereas the others can be inferred by range checking. However, this opportunity has not been further considered, because preliminary experiments have shown that the overhead associated with it would be excessive.

IV. Experimental Results

The performance of the code discussed in Section III has been evaluated by encoding and decoding a set of 10^7 uniformly-distributed, pseudorandom messages of varying lengths. On both architectures, the time spent in the encoding and decoding routines, as a function of the payload size s, has been measured by means of a free-running, 32-b counter clocked by the CPU core clock.

In this way, it was possible to collect a cycle-accurate encoding and decoding delay measurement for each message. Working on a large number of pseudorandom messages was convenient to single out any data dependency. External sources of measurement noise were avoided by running the experiments in a very controlled environment, with interrupts disabled and unused peripheral devices powered down.

A. Code/Data Placement

A first interesting insight into system behavior is given by the experiments carried out on the `arm7` architecture with the default code/data placement set by the toolchain. On the evaluation board being used, this implies that code, data, and stacks are stored in an off-chip DRAM. Fig. 2 shows the frequency distribution of the encoder delay in this case, for a payload size of $s = 1$ and $s = 7$ bytes.

The experimental data clearly show that the encoder is unacceptably slow, namely, the amount of time needed to encode a payload of 7 bytes is 15 μs on average. At a bit rate of 500 kbps, this corresponds to about $7.5t_{\mathrm{bit}}$. Moreover, the delay is also affected by a significant amount of jitter, which increases as

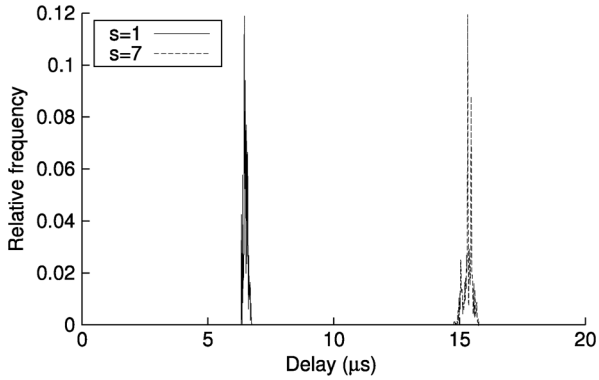

Fig. 2. Frequency distribution, DRAM-resident `arm7` encoder delay.

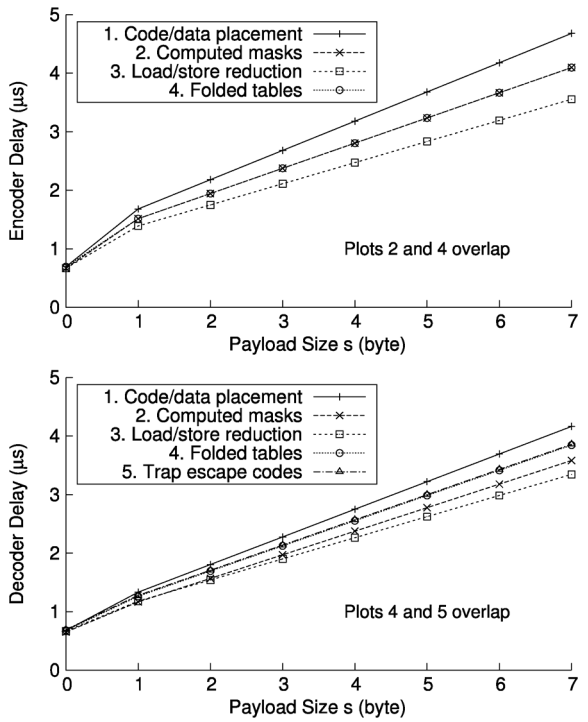

Fig. 3. Encoder/decoder delay, `arm7` architecture.

s grows, and is of the order of 1 μs in the worst case. The slowness can be justified by considering that external memory is accessed by means of a 16-b data bus, to reduce costs. On the other hand, jitter is due to DRAM refresh cycles, which stall any instruction or data access operation issued by the CPU while they are in progress. For this architecture, both issues were solved by moving the critical code, data and stack into the on-chip SRAM. After this was done, all of the subsequent measurements exhibited *no jitter*.

It must also be remarked that this behavior is not specific to the `arm7` architecture, but is just a special case of a more widespread phenomenon. For instance, the `cm3` architecture exhibited a very similar behavior (not shown in the paper for conciseness) when the forward and reverse lookup tables were stored in Flash memory.

In this case, the jitter was due to a component—known as Flash accelerator—whose purpose is to mitigate the relatively large access time of Flash memory (up to five CPU clock cycles), by anticipating future Flash memory accesses. Although

TABLE III
ENCODER/DECODER DELAY MODEL, `arm7` ARCHITECTURE

Code version	$\rho = 0$ for all measurements					
	Encoder (μs)			Decoder (μs)		
	b	l	k	b	l	k
1. Code/data placement	0.69	0.49	0.50	0.68	0.18	0.47
2. Computed masks	0.67	0.42	0.43	0.65	0.11	0.40
3. Load/store reduction	0.67	0.36	**0.36**	0.67	0.15	**0.36**
4. Folded tables	0.69	0.39	0.43	0.68	0.15	0.43
5. Trap escape codes	—	—	—	0.69	0.15	0.43

the prediction algorithm is quite effective with instruction fetch, it does not work equally well for forward and reverse table lookup, and, hence, it introduces a variability in the table access time. Like in the previous case, the solution was to force the compiler to allocate the forward and reverse lookup tables in on-chip SRAM instead of Flash.

B. Delay Model

Fig. 3 (plots 1–3) shows the outcome of the optimizations discussed in Section III-A on the `arm7` architecture. Unlike Fig. 2, it shows the delay as a function of s. The sample distribution is not shown because, after appropriate code/data placement, the sample variance was zero in all experiments. In other words, during the experiments J_{P} was consistently below the timer resolution, that is, one CPU clock cycle.

Intuitively, the delay needed to encode or decode a message, where s is the payload size in bytes, can be modeled by $\hat{t}(s)$ as

$$\hat{t}(s) = \begin{cases} b, & \text{if } s = 0 \\ b + l + ks, & \text{if } s > 0 \end{cases} \tag{2}$$

where b is the base delay incurred, regardless of the payload length, because of the function prologue and epilogue, l is the additional processing time needed to set up the encoding or decoding loop when $s > 0$, and k is the time needed to process each byte of the message.

The model parameters have been derived from the experimental data as follows: b has been measured directly, k has been obtained by a linear least-squares fit of the experimental data for $s > 0$, and l has been obtained by difference between the extrapolation of the linear fit to the case $s = 0$ and b. To evaluate the accuracy of the fit, instead of the commonly used norm of the residuals, the maximum absolute value of the residuals ρ has been used, because the worst-case prediction error is of interest in this case. It is defined as

$$\rho = \max_{s=1}^{7} \left| \hat{t}(s) - t(s) \right| \tag{3}$$

where $t(s)$ is the measured delay and $\hat{t}(s)$ is the predicted delay for a given s. $t(0)$ was not considered in (3), because the case $s = 0$ is handled in a special way by the algorithms.

Table III (rows 1–4) lists the model parameters for the software versions presented in Section III-A, derived from the corresponding experiments. The first interesting result is that the prediction error ρ was zero in all cases, that is, the behavior of the `arm7` implementation always followed the linear model (2) perfectly.

On the other hand, the same experiments performed on the more modern `cm3` architecture gave rather different results, even with the same toolchain configuration and options. They are shown in Fig. 4 for what concerns the decoder delay, but the

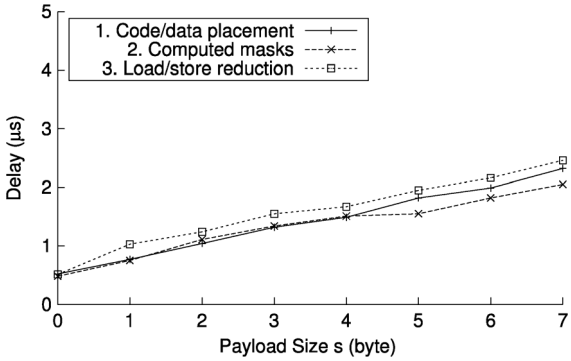

Fig. 4. Decoder delay, cm3 architecture, loop unrolling enabled.

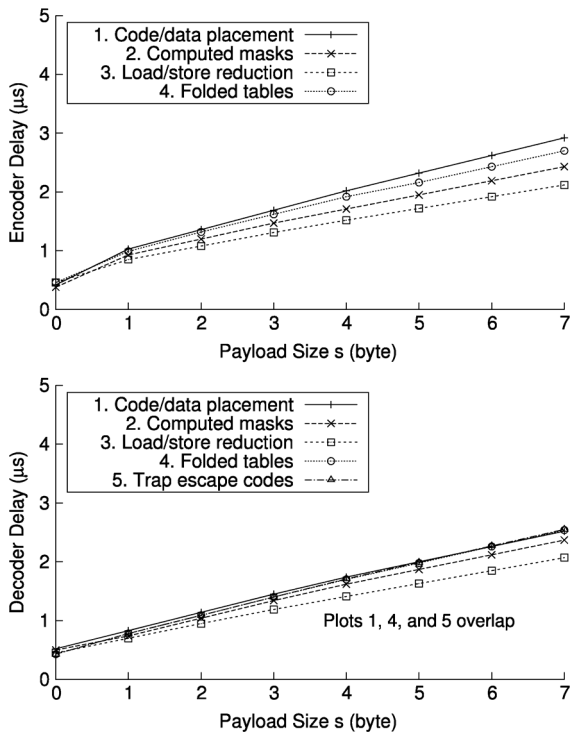

Fig. 5. Encoder/decoder delay, cm3 architecture, loop unrolling disabled.

encoder's behavior is very similar. Although the sample variance was still zero in all cases—meaning that the architecture nevertheless behaved in a fully deterministic way—the clean, linear relationship between s and the encoding/decoding time of the arm7 architecture was lost.

After some further experimentation, the reason of the peculiar behavior was identified in a compiler optimization, known as *loop unrolling*. In this particular scenario, the compiler unrolled the encoding and decoding loops by a factor of two, and this gave rise to the up/down pattern in the delays, depending on whether s is odd or even. When this optimization was turned off linearity was almost completely restored, as shown in Fig. 5 (plots 1–3), at a small performance cost.

Interestingly enough, loop unrolling never took place on the arm7 architecture (Fig. 3). This difference is most probably due to the different compiler versions in use and to the architectural dissimilarities between arm7 and cm3.

The linear model (2) was then fitted onto the new set of data, obtaining the results shown in Table IV (rows 1–3). Since ρ is not zero, perfect linearity was not achieved yet. However, the values of ρ also show that the worst-case prediction error of the linear delay model is less than or equal to 42 ns. This value is likely to be acceptable in most applications, both in absolute terms (less than 5 clock cycles on the cm3 platform) and when compared with other sources of delay misprediction in the system as a whole.

TABLE IV
ENCODER/DECODER DELAY MODEL, cm3 ARCHITECTURE

Code v.	Encoder				Decoder			
	b (μs)	l (μs)	k (μs)	ρ (ns)	b (μs)	l (μs)	k (μs)	ρ (ns)
1.	0.42	0.31	0.32	26	0.52	0.06	0.28	34
2.	0.38	0.32	0.25	21	0.50	0.00	0.27	34
3.	0.46	0.20	**0.21**	21	0.45	0.04	**0.23**	20
4.	0.46	0.29	0.28	**42**	0.43	0.08	0.29	21
5.	—	—	—	—	0.43	0.07	0.30	26

TABLE V
DETAILED ENCODER/DECODER FOOTPRINT FOR ALL CODE VERSIONS

Architecture and code version	Encoder/Decoder (byte) code (total)	code (loop)	table	stack
arm7				
1. Code/data placement	228/160	80/76	512/1024	32/28
2. Computed masks	212/144	76/72	512/1024	28/24
3. Load/store reduction	200/148	68/68	512/1024	28/24
4. Folded tables	234/164	88/88	128/256	32/28
5. Trap escape codes	—/168	—/88	—/256	—/28
cm3				
1. Code/data placement	164/116	64/60	512/1024	28/32
2. Computed masks	144/100	48/58	512/1024	20/28
3. Load/store reduction	136/96	46/48	512/1024	24/20
4. Folded tables	160/112	66/66	128/256	28/20
5. Trap escape codes	—/116	—/66	—/256	—/20

It is important to remark again that a delay prediction error ρ does *not* imply that there is any uncertainty or jitter in the delay itself. On the contrary, it only means that the use of a linear delay prediction from Table IV instead of the actual measured data plotted in Fig. 5 may introduce a *systematic* prediction error less than or equal to ρ.

Regarding raw performance, it is important to remark the efficacy of the source-level code optimizations described so far. As shown in Tables III and IV, when combined together, they brought the encoding time from 0.50 to 0.36 μs (-28%) per byte on the arm7 architecture. The improvement on the cm3 architecture was even better, from 0.32 to 0.21 μs (-34%) per byte. The optimizations were less effective, but still remarkable, on the decoder: -23% on arm7 and -18% on cm3. In absolute terms, the encoder and decoder loops were both reduced to 26 clock cycles per byte on arm7 and to 21 and 23 cycles, respectively, on cm3.

C. Memory Footprint/Performance Trade-Off

Table V (rows 1–3) lists the footprint of all code versions presented so far. Unlike the optimizations presented in Section III-A, the further updates to the code discussed in Section III-B are not "one-way," because they entail a tradeoff between footprint reduction and loss of performance/determinism.

For what concerns encoder and decoder table folding, measured performance results are shown in Figs. 3 and 5 (plot 4). The corresponding linear delay model parameters can be found in the fourth row of Tables III and IV. In both cases, the Boolean

negation operations needed to fold the table were implemented by means of the C exclusive OR operator with appropriate bit masks, to avoid introducing any data dependency in the delay.

Moreover, a last version of the decoding function has been developed, which checks for invalid patterns and escape codes—discussed in Section II—in the encoded byte stream. Although the raw performance of this version ought to be inferior to the previous ones by intuition, it turns out that—if the additional checks are carefully coded—the actual overhead is as small as 0.01 μs (1 clock cycle) per byte on `cm3`. The overhead on `arm7` is zero, because the compiler is able to embed the additional checks in the existing instruction stream.

As can be seen in Table V, the footprint reduction due to table folding more than offsets the extra code size.

V. Performance Evaluation

8B9B is compared here with the other proposals described in Section II-A, with respect to encoding efficiency, jitter reduction capability, and implementation complexity.

A. Encoding Efficiency

The data field in CAN frames—and, typically, the message payload as well—are encoded on an integral number of bytes. Hence, the encoding efficiency can be defined as the ratio between the size s of the original payload and the DLC value in the encoded frame. Only XOR does not cause any overhead. On the contrary, part of the data field is unavoidably wasted in all the other approaches. *SXP*, *SXB*, *SBS*, and *EEM* have been conceived originally for the use with the *shared-clock* (S-C) synchronization protocol, which means that some additional information (slave ID) has to be included in the frame. In order to carry out a fair comparison we assumed that only the original payload is encoded in the data field.

As said in Section II, because of the translation process of the payload into 9-b patterns, the final *8B9B*-encoded sequence is exactly one byte larger than the original payload. In the case of SXP, only one bit is required in order to specify whether or not the EXOR is applied to the payload; in practice, this means that one byte is wasted. Instead, in *SXB* a bit map has to be included, which specifies on a per-byte basis if EXOR was applied. BS should be prevented in this part of the frame also: if the payload is 5 bytes or less, one byte is sufficient to store the bit map, which can be encoded at the beginning of the data field as $\mathtt{S}_1\mathtt{M}_1\mathtt{M}_2\mathtt{S}_2\mathtt{M}_3\mathtt{M}_4\mathtt{M}_5\mathtt{S}_3$, where bits $\mathtt{S}_1$, $\mathtt{S}_2$, and $\mathtt{S}_3$ have the same function as the break bit (they are at the opposite level as the preceding bit), whereas each bit $\mathtt{M}_y$ specifies whether or not EXOR was applied to the byte in position y of the payload. If the payload is larger than 5 bytes, the bit map should take two bytes. This means, in *SXB*, 6 bytes at most can fit in the payload (7 bytes are also allowed, if no countermeasures are taken to prevent stuff bits in the bit map).

Concerning *SBS*, one software stuff bit may be required every three original bits at worst. This case corresponds, e.g., to the sequence

$$\boxed{0}000\boxed{1}111\boxed{0}000\boxed{1}\ldots$$

for the data field, where boxes denote software stuff bits (the first one takes into account a possible contribution due to the frame header). A padding is required in *SBS* at the end of the encoded bit sequence to make its length fixed. The size of the data field has to be selected so that the worst case depicted above can be tackled as well. Finally, in *EEM*, every byte of the payload is encoded on 11 b as $\mathtt{B}_1\mathtt{S}_1\mathtt{B}_2\mathtt{B}_3\mathtt{B}_4\mathtt{S}_2\mathtt{B}_5\mathtt{B}_6\mathtt{B}_7\mathtt{S}_3\mathtt{B}_8$, where $\mathtt{S}_1$, $\mathtt{S}_2$, and $\mathtt{S}_3$ have the same purpose as BB whereas $\mathtt{B}_k$ is the bit in position k of the considered byte.

TABLE VI
Encoding Efficiency

Payload size s (byte)	Data field size DLC (byte)					
	XOR	*SXP*	*SXB*	*SBS*	*EEM*	*8B9B*
0	0	0	0	0	0	0
1	1	2	2	2	2	2
2	2	3	3	3	3	3
3	3	4	4	**5**	**5**	4
4	4	5	5	**6**	**6**	5
5	5	6	6	**7**	**7**	6
6	6	7	**8**	—	—	7
7	7	8	—	—	—	8
8	8	—	—	—	—	—

In Table VI the DLC value is shown vs. the original payload size s. As can be seen, the resulting data field in *8B9B* is never longer than the other approaches, except XOR. Moreover, it is the only solution that is able to encode also 7-byte payloads, besides *SXP* (and XOR). If the contribution of stuff bits, which are possibly added to the data field by the CAN controller, is taken into account as well, *8B9B*, *SBS*, and *EEM* manage to improve, on the average, their encoding efficiency with respect to EXOR-based approaches. For XOR in particular, the mean transmission time may increase by 1 to 3 bit times [28].

B. Jitter Reduction Capability

The overall communication jitter is made up of two parts, which are not completely uncorrelated: the former is caused by BS whereas the latter depends on the codec. Variations in the encoding and decoding times, in fact, lead to additional processing jitters. Likely, the simpler and shorter the code is, the lower J_P is. Generally speaking, the first contribution on modern CPUs is likely to be larger than the latter.

Concerning jitter due to BS, none of the existing approaches deals with the CRC. Thus, insertion of stuff bits is never prevented completely, which means that jitter can only be lowered but not removed altogether. The best way to describe jitter is through the statistical distribution of frame transmission times C over the bus. Two related meaningful quantities are the standard deviation σ_C and worst-case jitter $J_C = C_\mathrm{max} - C_\mathrm{min}$ for any given message stream.

Although an upper bound $\widehat{J}_C$ on jitter can be computed analytically, its measured value J_C, as well as σ_C, depend heavily on experimental conditions and, in particular, on the actual traffic (payload size and set of values it may assume). Several theoretical bounds and statistical indices—including J_C and σ_C—have been computed in [28] for different, realistic *traffic models* (each of which defines basically the generation law of the messages belonging to a set of streams).

The XOR scheme is by far the simplest and fastest approach, but its performance can be suboptimal—in terms of jitter reduction capability—when the traffic shows a high degree of randomness. As shown in [18], XOR hardly provides any advantage in the case the payload contains uniformly distributed random

values. The same happens when messages carry analog signals that do not vary abruptly [28]. On the contrary, XOR adoption can be advantageous if long sequences of bits at the same value are likely to be found in the payload, as in the case of packed digital signals. In this case, a decrease of both J_C and σ_C in excess of 40% could be expected when the payload includes more than 4 bytes [28]. However, $\widehat{J_C}$ is the same as in plain CAN, that is, $\widehat{J_C} = (4 + 2s)t_{\mathrm{bit}}$.

The selective XOR scheme *SXB* is able to show better behavior than XOR under generic traffic conditions. According to [18], improvements as high as about 40% (for large messages) can be achieved on J_C and σ_C for random traffic. For the sake of truth, stuff bits may still be added to the data field in this case: since one stuff bit may appear at worst on each boundary between bytes, $\widehat{J_C} = (4 + s)t_{\mathrm{bit}}$. Moreover, the codec is more complex, which means that J_{P} is likely to increase consequently. The simpler SXP scheme is not any better than XOR, both in theory (same $\widehat{J_C}$) and in practice [18].

SBS, *EEM*, and *8B9B* are all able to ensure that no stuff bit at all is added to the data field. This implies that both J_C and σ_C are noticeably lower than EXOR-based approaches. Furthermore, the expected residual jitter is, in theory, almost the same in these three cases. In fact, the only possible difference among them depends on bit stuffing carried out on the CRC field. Because of the way the CRC is computed, it is reasonable to assume that the number of stuff bits that are added to the messages in each stream is, on the average, the same, irrespective of payload size, generation law, and encoding.

A thorough analysis of *8B9B* [28] confirms that, as expected, both the overall jitter J_C due to bit stuffing and its standard deviation σ_C do not depend on the specific characteristics of the signals carried in the message stream. In particular, in the experiments J_C never exceeded the theoretical upper bound $\widehat{J_C} = 4t_{\mathrm{bit}}$ while σ_C was $0.73t_{\mathrm{bit}}$ in the worst case.

By contrast, when using XOR, J_C was as high as $13t_{\mathrm{bit}}$ when 8-byte random data are taken into account, whereas σ_C increased to $1.49t_{\mathrm{bit}}$. The improvements achieved by *8B9B* are even higher when analog signals are considered: in this case, for the same payload size, J_C and σ_C in XOR were equal to $18t_{\mathrm{bit}}$ and $2.65t_{\mathrm{bit}}$, respectively.

Evaluating J_{P} is not simple, in that it heavily depends on both the specific platform and code optimizations. For this reason, a direct comparison of the results obtained in Section IV with the figures available in literature (in the order of several tens of microseconds and even bigger) would not be fair. In principle, the *8B9B* codec shall feature a very low J_{P} because it is conceptually simpler than selective XOR, *SBS*, and *EEM*. As a consequence, the optimizations outlined in Section III-A can readily be applied to it without effort.

This is not necessarily true for the other algorithms. For instance, since the insertion of software stuff bits in *SBS* inherently depends on an iterative, bit-by-bit inspection of the payload content, it is conceptually difficult to devise an implementation in which neither the number of iterations nor the operations done in each iteration are affected by the payload content itself.

In practice, as discussed in Section IV, we found that it is indeed possible to reduce the J_{P} of *8B9B* below the measurable threshold (i.e., less than one CPU clock cycle) on two dissimilar processor architectures. For all the above reasons, *8B9B* is able to offer on the whole better jitter reduction capability than the other solutions.

C. Implementation Complexity

This parameter has to do with the codec footprint and (average) execution time. XOR is by far the simplest and fastest solution, with the smallest footprint. Both *SXP* and *SXB* encoders have to check the presence of primer sequences in the payload. Decoding is instead much faster.

SBS operates on a bit-by-bit basis on both the transmitter and receiver's sides, in order to determine when a software stuff bit has to be inserted or removed, respectively. On the other hand, both *EEM* and *8B9B* operate on a byte-by-byte basis, and hence, they are potentially faster. Because of its intrinsic simplicity, *8B9B* is likely able to ensure the shortest execution time, excluding XOR, together with a small footprint.

VI. Conclusion

This paper presented *8B9B*, an encoding scheme designed to prevent bit stuffing within the data field of CAN messages, a property especially useful in tightly synchronized systems. Although several other methods with the same purpose have been proposed in the recent past, we believe that *8B9B* can outperform them when the typical requirements of small embedded systems are taken into account. Namely, it provides a balanced blend of determinism, encoding efficiency, codec speed, and footprint.

The encoding scheme is completely transparent to both the sending and receiving applications, as well as other mechanisms aimed at reducing or removing frame-level jitter. Since no assumptions are made on how the payload is used and formatted by the upper protocol layers, the proposed approach is completely general-purpose. Moreover, it is possible to apply *8B9B* only to jitter-sensitive messages, because encoded messages can coexist with plain ones. In this way, full backward compatibility is achieved. As is typical in CAN, the distinction between the two kinds of message can easily be done through the message identifier. The only limitation is that the payload to be encoded cannot be larger than 7 bytes.

In order to show that the *8B9B* scheme is suitable for practical use, it has been implemented on two popular families of microcontrollers and then thoroughly evaluated. The results show that the code is very fast, exhibits a small memory footprint and does not introduce any processing jitter. Overall, the proposed mechanism can therefore be profitably adopted in existing projects and solutions, even if the underlying hardware platform has got a limited processing power.

References

[1] *ISO 11898-1—Road Vehicles—Controller Area Network (CAN)—Part 1: Data Link Layer and Physical Signalling*, , International Organization for Standardization, 2003, ISO.

[2] FlexRay Communications System Protocol Specification. ver. 3.0.1, FlexRay Consortium, Oct. 2010.

[3] P. Martí, J. Yépez, M. Velasco, R. Villà, and J. Fuertes, "Managing quality-of-control in network-based control systems by controller and message scheduling co-design," *IEEE Trans. Ind. Electron.*, vol. 51, no. 6, pp. 1159–1167, Dec. 2004.

[4] 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.

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

[6] J. Aweya, "Technique for differential timing transfer over packet networks," *IEEE Trans. Ind. Inf.*, vol. 9, no. 1, pp. 325–336, Feb. 2013.

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

[8] M. Nahas, M. Short, and M. J. Pont, "The impact of bit stuffing on the real-time performance of a distributed control system," in *Proc. 10th Int. CAN Conf.*, 2005, pp. 10.1–10.7.

[9] T. Nolte, H. Hansson, C. Norström, and S. Punnekkat, "Using bit-stuffing distributions in CAN analysis," in *Proc. IEEE/IEE Real-Time Embedded Syst. Workshop*, 2001, pp. 1–6.

[10] 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.

[11] G. Rodriguez-Navas, S. Roca, and J. Proenza, "Orthogonal, faulttolerant, and high-precision clock synchronization for the Controller Area Network," *IEEE Trans. Ind. Inf.*, vol. 4, no. 2, pp. 92–101, May 2008.

[12] M. J. Pont, *Patterns for Time-triggered Embedded Systems: Building Reliable Applications With the 8051 Family of Microcontrollers*. Reading, MA, USA: Addison-Wesley, 2001.

[13] 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. Inf.*, vol. 6, no. 4, pp. 503–520, Nov. 2010.

[14] *ISO 11898-4—Road Vehicles—Controller Area Network (CAN)—Part 4: Time-Triggered Communication*, , International Organization for Standardization, 2004, ISO.

[15] P. Gaj, J. Jasperneite, and M. Felser, "Computer communication within industrial distributed environment-a survey," *IEEE Trans. Ind. Inf.*, vol. 9, no. 1, pp. 182–189, Feb. 2013.

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

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

[18] M. Nahas and M. Pont, "Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study," in *Proc. 2nd UK Embedded Forum*, 2005, pp. 4–17.

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

[20] M. Nahas, "Applying eight-to-eleven modulation to reduce message-length variations in distributed embedded systems using the Controller Area Network (CAN) protocol," *Can. J. Electr. Electron. Eng.*, vol. 2, no. 7, pp. 282–293, 2011.

[21] G. Cena, I. Cibrario Bertolotti, and A. Valenzano, "An efficient fixed-length encoding scheme for CAN," in *Proc. 9th IEEE Int. Workshop Factory Commun. Syst.*, 2012, pp. 265–274.

[22] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, 2007.

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

[24] *International Standard ISO/IEC 9899, Programming Languages—C*, , ISO/IEC, Dec. 1999, 2nd ed..

[25] *"LPC24XX User Manual, UM10237 Rev. 2,"* NXP B.V., Dec. 2008.

[26] *"LPC17XX User Manual, UM10360 Rev. 2,"* NXP B.V., Aug. 2010.

[27] R. Barry, *Using the FreeRTOS Real Time Kernel—Standard Edition*, 1st ed. Raleigh, NC, USA: Lulu, 2010.

[28] G. Cena, I. C. Bertolotti, T. Hu, and A. Valenzano, "Performance comparison of mechanisms to reduce bit stuffing jitters in Controller Area Networks," in *Proc. 17th IEEE Conf. Emerging Technol. Factory Autom.*, Sep. 2012, pp. 1–8.

Gianluca Cena (SM'09) received the Laurea degree in electronic engineering and 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, 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 authored and coauthored about 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 an 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 Turin, 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 coauthored 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 M.S. degree in computer engineering from Politecnico di Torino, Turin, Italy, in 2010, where she is working toward the Ph.D. degree in information and system 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 a 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, Turin, Italy, in 1980.

He is Director of Research with the National Research Council of Italy (CNR). He is currently with Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Turin, 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 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. He is also Vice-President of the Piedmont Chapter of the Italian National Association for Automation.