Preventing bit stuffing in CAN

G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano
CNR-IEIIT (Torino)
Outlook

- **Bit stuffing** is a very simple and efficient coding scheme, but it causes *transmission jitters* and worsens *data integrity*
- The problem about integrity has been tackled *explicitly* in the recently standardized *ISO CAN FD* protocol
- Is it possible to improve the behavior of *existing* controllers (or, generally, of *any* CAN controller operating in *legacy* mode) *easily* and *inexpensively*?
- **Yes!** We just have to remove *all* stuff bits from frames with *critical* requirements
- **Much easier** than one may expect... let’s see how...
Physical layer

• The signal sent on the CAN bus relies on non-return to zero (NRZ) encoding with bit stuffing (BS)
  – Whenever 5 consecutive bits at the same level are detected on the bus the CAN controller in the transmitting node automatically inserts one stuff bit at the opposite value
  – Stuff bits introduce edges in the signal on the bus that permit CAN controllers in receiver nodes to synchronize their DPLLs properly
  – Stuff bits are removed before frames are decoded
Data-link layer

- The **standard** (base) frame format (11-bit ID) is mainly considered in the following
  - The **extended** 2.0B frame format (29-bit ID) is mostly similar and can be dealt with in the **very same** way
  - **Remote** frames do not have the data field, so they are of **fixed** duration
First problem: transmission jitter

Transmitter nodes: message generation times can be very precise

Receiver nodes: fluctuations may appear in reception times

Interrupt

Minimum intertime

Actuation jitter
Details on transmission jitter problem

- **Transmitting** devices send messages at *precise* points in time if arbitration is not exploited (e.g., in *master-slave* schemes)
- **Receiving** devices carry out actuations on the *interrupts* raised at the end of message reception in *event-driven* systems
- Because of bit stuffing the *duration* of each message depends on the *value* of the conveyed data besides their nominal *size*
- This leads to undesired fluctuations (*jitter*) on *actuation* times that *in theory* can be as long as 24 *bit times*
- This phenomenon limits the *precision of timings* in distributed control systems based on CAN
Second problem: data integrity

Original frame

Transmitted frame

2 transmission errors (bit flips)

Seen as stuff bit and removed

Seen as data bit and retained

Received frame

Decoded frame

TX node

CAN bus

RX node
Details on data integrity problem

- CAN frames include a cyclic redundancy check (CRC) which is able to detect 5 erroneous bits located anywhere in the frame.
- The bit stuffing/destuffing mechanism may lead to a decrease of the CRC error detection capability of CAN.
- A pair of erroneous bits located apart enough that cause both insertion and removal of stuff bits may trick the receiver.
- The residual error probability that a corrupted message is considered correct by receivers increases.
- This phenomenon worsen integrity of data exchanged on the CAN bus in distributed control systems.
Two (non-competing) solutions

**H/W Solution**
- *New* frame format: requires *specific* CAN FD controllers
- Much *faster* than CAN
- Much better *data integrity*
- Much *larger payload*
- *Partially compatible* with existing systems and devices

**S/W Solution**
- *Legacy* frame format: can use *existing* CAN controllers
- More *accurate* timings
- Better *data integrity*
- No additional cost
- *Fully compatible* with existing systems and devices
Comparison of features

• (ISO) **CAN FD** protocol
  – One order of magnitude *faster* than CAN (overclocking+oversizing)
  – Does not suffer from the known CAN issue about *data integrity*
  – ...but a legacy CAN node receiving a CAN FD frame reacts very badly!
  – To ensure *compatibility* CAN FD controllers can operate in legacy mode
  – ...but doing so precludes all CAN FD advantages!

• Our solution: **Zero Stuff-bit encoding**
  – *Prevents* stuff bit insertion in frames done by CAN controllers
  – *Communication jitters* reduced by about *two* orders of magnitude
  – *Residual error probability* reduced by about *two* orders of magnitude
  – Can also be implemented using *lightweight* and *fast* S/W codecs
  – *Inexpensive* and *completely compatible* with legacy CAN devices
CAN frame format

- Every CAN frame ($F$) is made up of 4 sections
  \[ F = H \| D \| R \| U \]
  - Operator “\|” denotes concatenation of bit strings
- Bit stuffing only applies to the stuffed part of the frame ($S$)
  \[ S = H \| D \| R \]
- The unstuffed trailer ($U$) is not involved and will be neglected

**Part of frame encoded with bit stuffing**
Zero Stuff-bit (ZS=ZSD+ZSC)

- **Header**: not a problem
  - Since the header is typically *fixed* for any given *message stream* it does not cause any jitter
  - Stuff bits can be often eliminated *completely* from the header through proper *identifier selection* in the configuration phase

- **Data**: dealt with using *Zero Stuff-bit Data* (ZSD) encodings
  - The payload is *encoded* at *runtime* in order to prevent the CAN controller from inserting stuff bits in the data field
  - Several solutions available: 8B9B, VHCC, etc.

- **CRC**: dealt with using the *Zero Stuff-bit CRC* (ZSC) mechanism
  - It is the most difficult achievement because the CRC is calculated by the CAN controller *autonomously* according to predefined rules
CAN frame format (ZS usage)

- ZS exploits a suitable encoding of the data field in order to completely prevent stuff bit insertion in both D and R.

```
<table>
<thead>
<tr>
<th></th>
<th>ZSD</th>
<th>ZSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C1 (9b)</td>
<td>K (&lt;8b)</td>
</tr>
<tr>
<td></td>
<td>C2 (9b)</td>
<td>T (3b)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cm (9b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
header (H)  data (D)  chk. (R)  trailer (U)
```

```
<table>
<thead>
<tr>
<th></th>
<th>ID + RTR (11+1b)</th>
<th>Res (2b)</th>
<th>DLC (4b)</th>
<th>DATA (0–8B)</th>
<th>CRC (15b)</th>
<th>CDEL</th>
<th>ACK</th>
<th>ADEL</th>
<th>EOF (7b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Part of frame encoded with bit stuffing
Dealing with stuff bits in the CRC field

• Every message \( M \) produced by applications can be seen as

\[
M = H \setminus D
\]

• The CAN controller \textit{automatically computes} the CRC and stores it in the checksum field \( R \)

\[
R = c(M)
\]

• \( R \) may include stuff bits but it \textit{cannot} be recoded \textit{directly} as in ZSD encodings → our idea is to act on \( R \) \textit{indirectly}

• 3 bits are \textit{reserved} at the end of the data field \( D \) to encode a tuning string \( T \), while the preceding part constitutes the effective data field \( E \)

\[
D = E \setminus T
\]
Details on Zero Stuff-bit Data (ZSD)

• Every byte of the original *payload* (P) is encoded separately on one 9-bit *codeword* using a *forward lookup table* (FLT)

\[ C_i = f(P_i) \]

• Codewords are *concatenated* in the same order as P

\[ C_{<1...m>} = C_1 \setminus C_2 \setminus ... \setminus C_i \setminus ... \setminus C_m \]

• An initial *break* bit (B) and a final *padding* (PAD) are added in 8B9B to obtain the *encoded effective data field*

\[ E = B \setminus C_{<1...m>} \setminus PAD \]

• The *leading section* (L) of the message does not include any stuff bits

\[ L = H \setminus E \]
Improving encoding efficiency

• The slack \((K)\) in the last byte of \(D\) is wasted in 8B9B
  – Variable-length, High-performance Code for CAN (VHCC) improves over 8B9B by permitting sub-byte encoding

• The payload can include additional user information \((P_{m+1})\) as well, encoded on \(h\) bits (up to one bit less than \(K\))

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

• \(K\) is not padded but used to encode \(P_{m+1}\) by means of specific encoding functions that rely on the same FLT

\[
C_{m+1} = f_k(P_{m+1})
\]

• The entire space available in \(E\) can be exploited

\[
E = B \setminus C_{<1\ldots m>} \setminus C_{m+1}
\]
Details on Zero Stuff-bit CRC (ZSC)

- The *partial contribution* of $L$ to the CRC is computed first

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

- For *every* $T_i$ in $\{001_2, 010_2, 011_2, 100_2, 101_2, 110_2\}$ the *partial contribution* to the CRC is computed and *XOR-ed* with $R_L$

$$R_i = R_L \oplus c(T_i)$$

- The *6 values* found for $R_i$ are checked to determine which one *does not cause* the insertion of any stuff bit in $T_i \setminus R_i$

$$g(T_i \setminus R_i) = false$$

- Every value of $T_i$ that satisfies the above condition can be used as the *tuning string* $T$ for message $M$

$$M = L \setminus T$$
Summary about ZS operation

- Part of frame encoded with bit stuffing
- Part of frame covered by CRC
- Leading section used by ZSC
- Encoded user data

CAN bus
CAN controller
1. ZSD
2. ZSC

Control application

Gianluca Cena
15th int’l CAN Conference (iCC), Vienna, October 27, 2015—Zero Stuff-bit for CAN
...or, in pseudo-code

1. The *payload* is encoded through ZSD: function $e(\bullet)$

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

2. The *tuning string* is calculated through ZSC: function $z(\bullet)$

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

3. The final *data field* is obtained by concatenation

   $$D = E \setminus T$$

4. The values of $H$ and $D$ are *fed* to the CAN controller

We proved *mathematically* that, irrespective of the payload, ZS encoding completely prevents stuff bit insertion in the entire portion of the frame that follows the header
# Overhead (maximum payload size in ZS)

<table>
<thead>
<tr>
<th>D size [B]</th>
<th>DLC val.</th>
<th>DL</th>
<th>E</th>
<th>P (m.h) size [B.b]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-</td>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0</td>
<td>45</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1</td>
<td>54</td>
<td>6</td>
</tr>
</tbody>
</table>
ZS codec

- An industrial-grade ZS codec has been developed for embedded platforms
  - Thoroughly tested for correctness
  - H/W codecs are also possible that do not introduce any delays
- Our SW codec implementation:
  - Runs on the NXP LPC2468 (core ARM7 @ 72 MHz)
  - Footprint: 1566 B code + 896 B r/o data + 108 B r/w data
- **Execution time** (DLC=8):
  -~ 12 µs (complete codec: both encoding and decoding)
  -~ 14 µs (further reduction of S/W jitter through optimization)
Results

• Most results below depend only on the *encoding scheme* — and *not* on the particular *codec implementation*

  • *Communication jitter* (DLC=8):
    – CAN: 24 bit times (48 µs @ 500 Kb/s)
    – ZSD-only: 4 bit times (8 µs @ 500 Kb/s)
    – ZS (H/W): 0 bit times
    – ZS (S/W): 0.47 µs (+ intrinsic jitter due to CAN controller)
    – ZS (Optimized S/W): < 30 ns (+ intrinsic jitter due to CAN controller)

  • *Residual error probability* (DLC=8):
    – CAN: 0.241 · 10⁻⁶ (conditioned to the presence of 2 errors)
    – 8B9B: < 0.0015 · 10⁻⁶ (95% confidence)
    – ZS: under verification *but certainly better* than 8B9B
Conclusions

• The ZS mechanism permits to *eliminate all stuff bits* in CAN
  – No jitters for systems that require *high accuracy* like computerized numerical control (CNC) or precise measuring instruments
  – Better data integrity for systems that require *high dependability* like *automotive* in-vehicle applications
  – An optimized high-performance *lightweight codec* has been developed that can be *directly* included in *embedded* platforms

• Unlike CAN FD, ZS preserves *complete compatibility* with *existing* CAN devices and networks
  – Because it relies on the *legacy* CAN data frame format

**CAN+ZS is not meant to replace CAN FD but to complement it when the *legacy (non-FD) frame format* is used**
Thanks for your attention

Any question?

gianluca.cena@ieiit.cnr.it