Application Report

Configurable Error Generator for Controller Area Network

Joseph Casuga and Hareesh Janakiraman *Applications Engineering - C2000 Real-Time Microcontrollers*

ABSTRACT

Even though it was introduced over three decades ago (primarily for automotive applications), Controller Area Network (CAN) is still going strong and has found its way into other applications such as industrial automation, aerospace and medical electronics. Many of these applications are safety critical and need a framework for emulating and analyzing the different types of error conditions that could occur in the network. Although many inexpensive CAN bus analyzers are currently available, they only provide some rudimentary capability to generate errors. For example, they may provide the capability to generate an error-flag at a specified location, but no more. FPGA Based error generators have been developed in a few academic settings. In this report, two different methods (general-purpose input/output (GPIO) method and LabVIEW™ method) are presented with better configurability in terms of precisely where the error could be introduced.

The GPIO method outlined in this document is applicable to all C2000™ Real-Time Microcontrollers that feature the DCAN module and Driverlib framework in C2000Ware. Currently, this includes the following devices: TMS320F2837xD, TMS320F2837xS, TMS320F2807x, TMS320F28004x, TMS320F2838x and TMS320F28002x.

The code examples were tested on a TMS320F280049 device; however, the examples can be easily adapted to run on any C2000 device that features the DCAN module. The project files and examples described in this document are available as part of C2000Ware.

The LABVIEW files discussed in this document can be downloaded from the following URL: https://www.ti.com/lit/zip/spracq3.

Table of Contents

List of Figures

List of Tables

Trademarks

LabVIEW™ is a trademark of National Instruments Corporation.
C2000™ and Code Compose Studio™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

The objective of this application report is to provide an easy-to-use hardware and software framework to generate and analyze different types errors in a CAN bus. Two different methods are presented:

- GPIO method: A test-case that could be run on any applicable C2000 device. This provides visibility on a GPIO pin, if desired. A working C2000 target board and Code Compose Studio™ (CCS) IDE is all that is required.
- LabVIEW method: This method is useful in case integration into a larger test setup (independent of a C2000 target) is desired. This needs the hardware outlined in Section 3.

For both methods, an oscilloscope with built-in CAN bus triggering/decoding is essential.

All simulated waveforms in this document were captured at the GPIO pin emulating the CAN transmit function. The effect of a CAN receiver node detecting an error and destroying the on-going frame will not be seen since the waveforms do not reflect true CAN bus activity, but only emulated CAN function. For this reason, the complete “CAN waveform” is seen in the oscilloscope captures. CAN frames shown in Table 1-1 were generated with GPIO and LabVIEW methods. These simulated frames were monitored with a CAN bus analyzer. The correct interpretation of the frames by the analyzer was validation that the frames are generated correctly and consistently.

Table 1-1. Generated CAN Frames

Frame Type								
ID	**Remote Request**	**ARBID**	**DLC**	**D0**	**D1**	**D2**	**D3**	**CRC**
Standard	No	0x45B	4	95	1A	23	45	0x5AD8
Standard	Yes	0x45B	4	n/a	n/a	n/a	n/a	0x238C
Standard	Yes	0x45B	0	n/a	n/a	n/a	n/a	0x7B43
Extended	No	0x1914A75B	4	95	1A	23	45	0x4101
Extended	Yes	0x1914A75B	4	n/a	n/a	n/a	n/a	0x4EB3
Extended	Yes	0x1914A75B	0	n/a	n/a	n/a	n/a	0x167C

2 Frame Generation – GPIO/CCS Method

This section explains how to configure error generation using the GPIO/CCS method. This should be used in conjunction with Figure 2-1.

- **Configure message length and bit-rate**
  - Configure message length in bytes (DLC)
  - Configure bit-rate (in bps)

Hardware set up procedure:

CAN_GPIO_MODE

1. Select the GPIO that would generate the CAN bit stream.
2. Connect a jumper wire from the chosen GPIO to the selected CANRX pin.
3. Connection is direct between GPIO and CANRX pins. No transceiver involved.

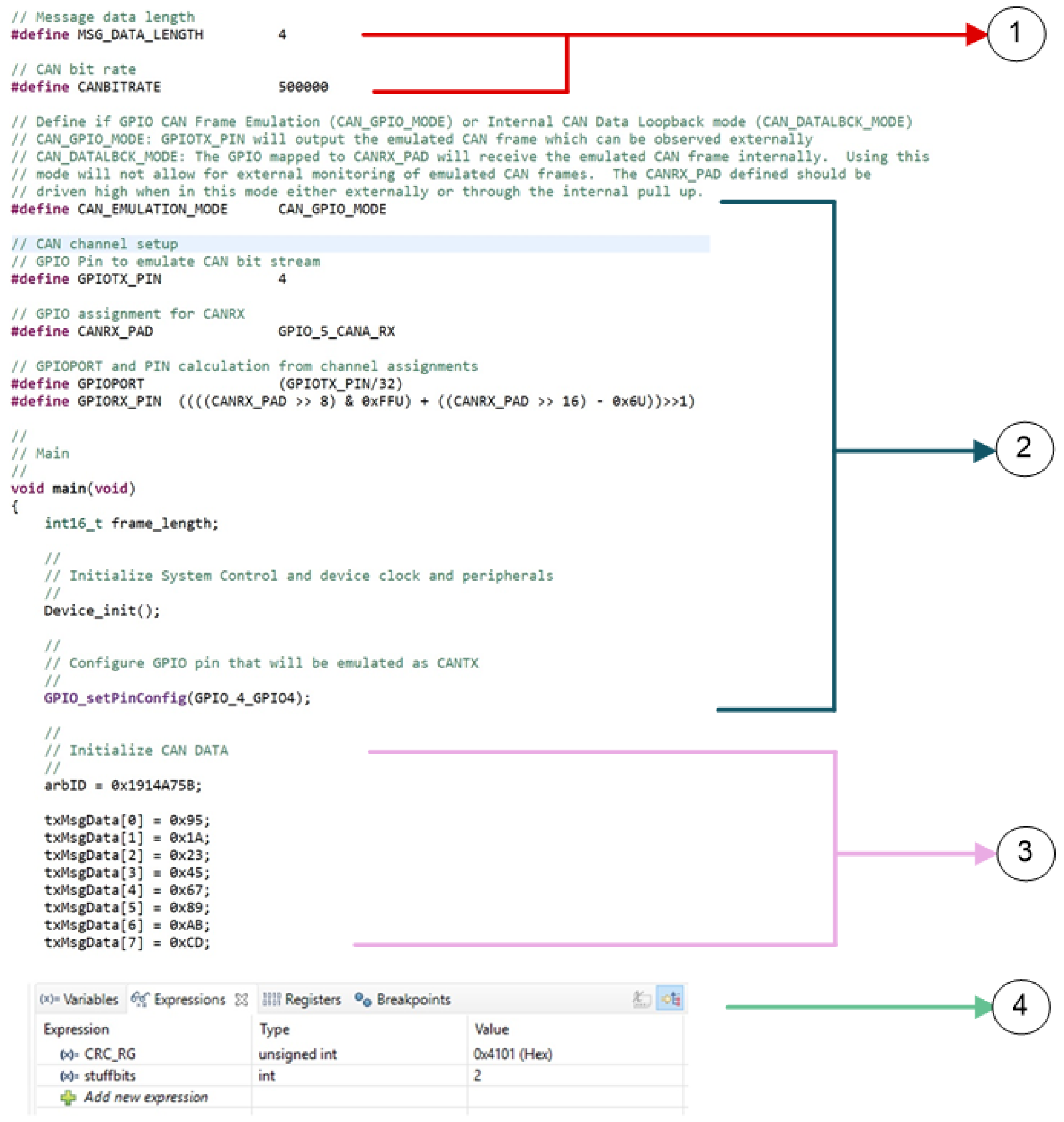
CAN_DATALBCK_MODE

1. No external connections are needed.
2. CANTX data directly goes to the CANRX buffer using internal connection.

- **Configure CAN Data Frame**
  - Enter desired Arbitration ID (Message ID)
  - Enter desired data to be transmitted
  - Choose data frame or remote request

CCS-Use Watch Expression to monitor CAN-frame related variables

- Generated 15-bit CAN CRC using polynomial 0xC599 can be monitored through the *Expressions* window in CCS.
- The number of occurrences for stuffed bits can also be monitored in the *Expressions* window of CCS.



```
// Message data length
#define MSG_DATA_LENGTH        4

// CAN bit rate
#define CANBITRATE             500000

// Define if GPIO CAN Frame Emulation (CAN_GPIO_MODE) or Internal CAN Data Loopback mode (CAN_DATALBCK_MODE)
// CAN_GPIO_MODE: GPIOTX_PIN will output the emulated CAN frame which can be observed externally
// CAN_DATALBCK_MODE: The GPIO mapped to CANRX_PAD will receive the emulated CAN frame internally.  Using this
// mode will not allow for external monitoring of emulated CAN frames.  The CANRX_PAD defined should be
// driven high when in this mode either externally or through the internal pull up.
#define CAN_EMULATION_MODE     CAN_GPIO_MODE

// CAN channel setup
// GPIO Pin to emulate CAN bit stream
#define GPIOTX_PIN             4

// GPIO assignment for CANRX
#define CANRX_PAD              GPIO_5_CANA_RX

// GPIOPORT and PIN calculation from channel assignments
#define GPIOPORT               (GPIOTX_PIN/32)
#define GPIORX_PIN  ((((CANRX_PAD >> 8) & 0xFFU) + ((CANRX_PAD >> 16) - 0x6U))>>1)

//
// Main
//
void main(void)
{
    int16_t frame_length;

    //
    // Initialize System Control and device clock and peripherals
    //
    Device_init();

    //
    // Configure GPIO pin that will be emulated as CANTX
    //
    GPIO_setPinConfig(GPIO_4_GPIO4);

    //
    // Initialize CAN DATA
    //
    arbID = 0x1914A75B;

    txMsgData[0] = 0x95;
    txMsgData[1] = 0x1A;
    txMsgData[2] = 0x23;
    txMsgData[3] = 0x45;
    txMsgData[4] = 0x67;
    txMsgData[5] = 0x89;
    txMsgData[6] = 0xAB;
    txMsgData[7] = 0xCD;
```

Expression	Type	Value
CRC_RG	unsigned int	0x4101 (Hex)
stuffbits	int	2

Figure 2-1. Frame Generation – GPIO/CCS Method

3 Frame Generation – LabVIEW Method

This section illustrates the various steps involved in generating CAN frames using the LabVIEW method, using '*GenerateCANStream.vi*'. It also explains the various indicators and configurable options in the Labview vi.

3.1 Setup Procedure

1. Obtain an arbitrary waveform generator (Agilent AG33250 used in this example; any AWG from A33xxx series should work seamlessly), LabVIEW development Software Package (2016 or later), and a USB-to-GPIB interface that would allow LabVIEW to control the arbitrary waveform generator (AWG).
2. Unzip and save '*GenerateCANStream.vi*' to a local computer running the LabVIEW Development Software.
3. Connect the AWG to the computer using the USB-to-GPIB interface. Connect the output of the AWG to the CANTX side of the CAN transceiver or directly to the CANRX of the CAN module reacting to the generated errors.

3.2 Input Windows

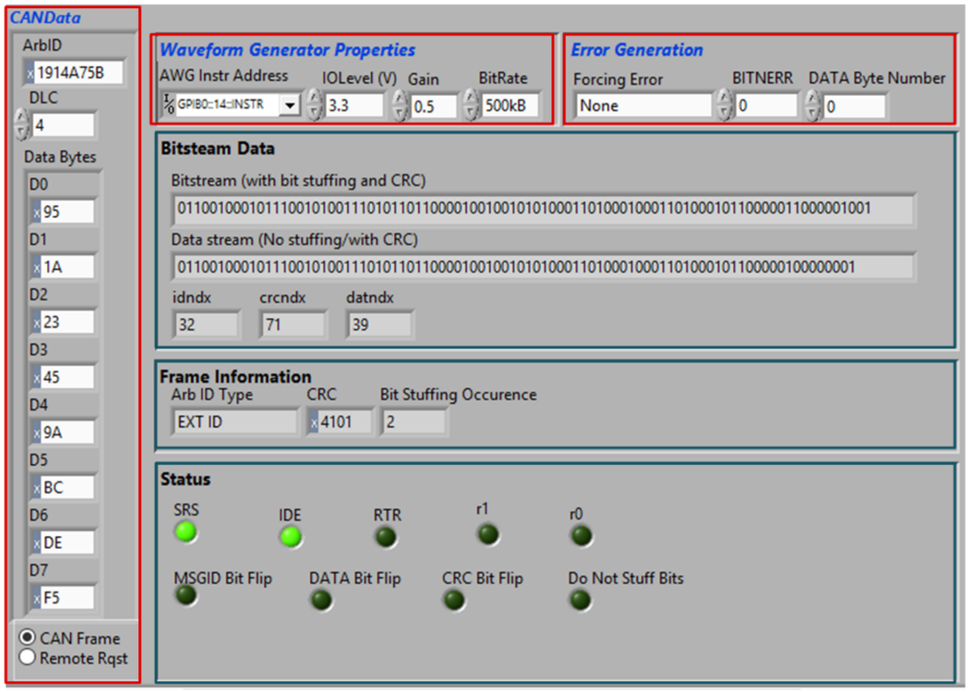
Figure 3-1 shows the various windows in *GenerateCANStream.vi*. Input windows are bordered in red.

- Error Generation:
  - At this point, set the Forcing Error value to 'None'. Configure this area later when generating CAN frames with errors.
- Configure the AWG (Waveform Generator Properties):
  - Set the appropriate GPIB instrument address.
  - Set the I/O level and gain to the desired levels using the I/O Level and Gain. Verify the levels with an oscilloscope to ensure that they are correct prior to connecting to the CANRX pin of the CAN module or to the CANTX pin of the transceiver.
  - Configure the bit-rate.
- Configure CAN Data Frame:
  - Enter desired Arbitration ID (Message ID).
  - Configure message length (in bytes – DLC).
  - Enter desired data to be transmitted.
  - Choose data frame or remote request.
- Press the Run button on the vi Front Panel to start generating CAN frames.

3.3 Output Windows

In the LabVIEW GUI, output windows are bordered in blue.

- Bit stream data:
  - Complete bit stream data is available for both bit-stuffed and non-bit-stuffed string.
  - Indices or markers for different fields are also calculated to supplement the data stream string.
- Frame Information:
  - Identifies whether the message ID is standard or extended.
  - Provides the calculated 15-bit CRC using polynomial 0xC599.
  - Provides the number of instances where bit stuffing occurred.
- Status:
  - Shows the bit values of SRS, IDE, RTR and reserved bits in either normal CAN frame generation or when any of these bits are flipped in error generation mode. Bright green stands for recessive (1) , while dark green is for dominant (0) state.
  - In error generation mode, corresponding fields with intended bit flips are highlighted in bright green color.



CANData
ArbID: 1914A75B
DLC: 4
Data Bytes
D0: 95
D1: 1A
D2: 23
D3: 45
D4: 9A
D5: BC
D6: DE
D7: F5
(●) CAN Frame
( ) Remote Rqst

Waveform Generator Properties
AWG Instr Address: GPIB0::14::INSTR
IOLevel (V): 3.3
Gain: 0.5
BitRate: 500kB

Error Generation
Forcing Error: None
BITNERR: 0
DATA Byte Number: 0

Bitsteam Data
Bitstream (with bit stuffing and CRC): 01100100010111001010011101011011000010010010101000110100010001101000101100000110000001001
Data stream (No stuffing/with CRC): 0110010001011100101001110101101100001001001010100011010001000110100010110000010000000001
idndx: 32
crcndx: 71
datndx: 39

Frame Information
Arb ID Type: EXT ID
CRC: 4101
Bit Stuffing Occurence: 2

Status
SRS 🟢 IDE 🟢 RTR ● r1 ● r0 ●
MSGID Bit Flip ● DATA Bit Flip ● CRC Bit Flip ● Do Not Stuff Bits ●

Figure 3-1. Frame Generation – LabVIEW Method

4 Reference Frames

Figure 4-1 through Figure 4-6 depict the reference frames generated per Table 1-1. All reference frames were generated using a USB-based CAN bus analyzer.

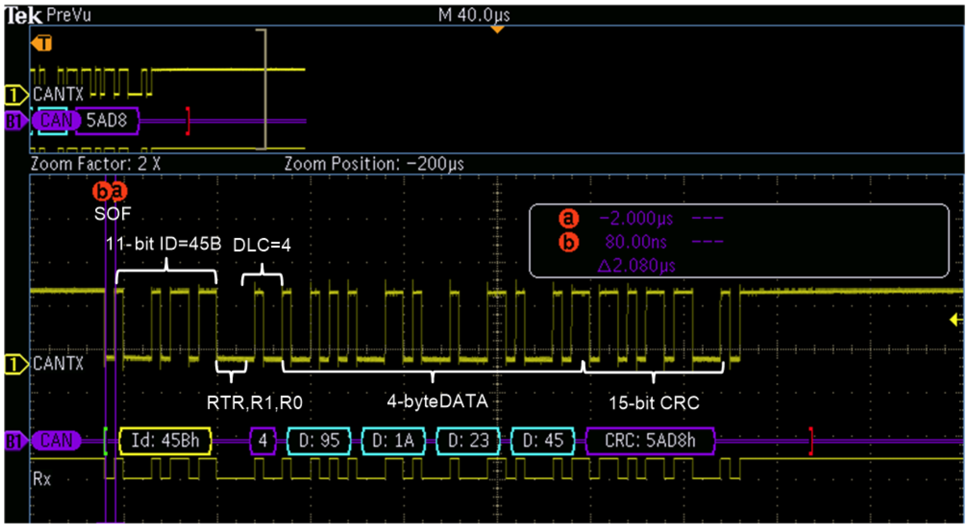


Figure 4-1. Reference Data Frame (Standard MSGID) With No Errors Induced, DLC = 4

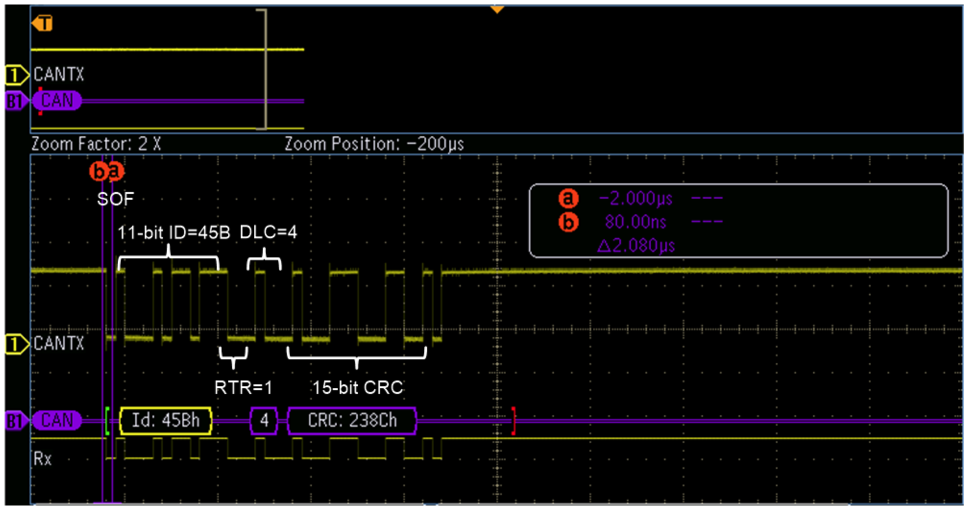


Figure 4-2. Reference Remote Frame (Standard MSGID) With No Errors Induced, DLC = 4

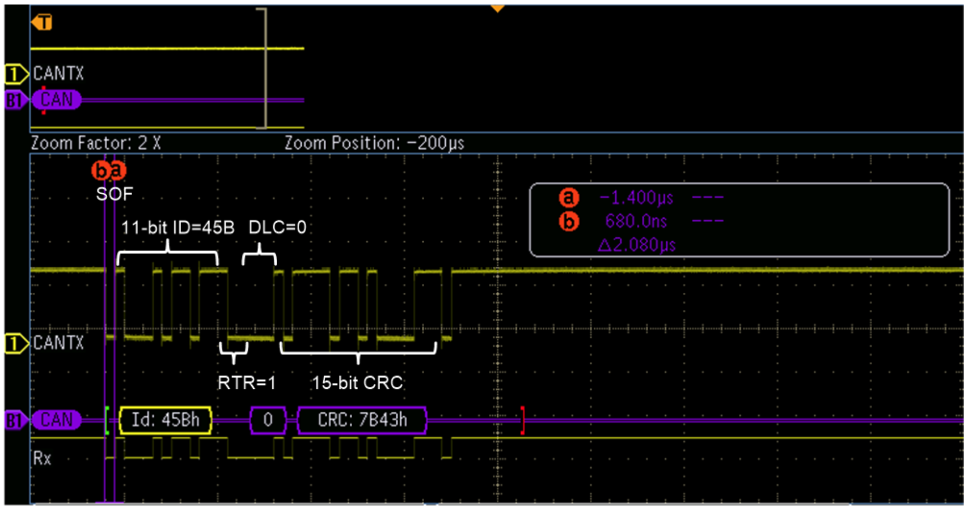


Figure 4-3. Reference Remote Frame (Standard MSGID) With No Errors Induced, DLC=0

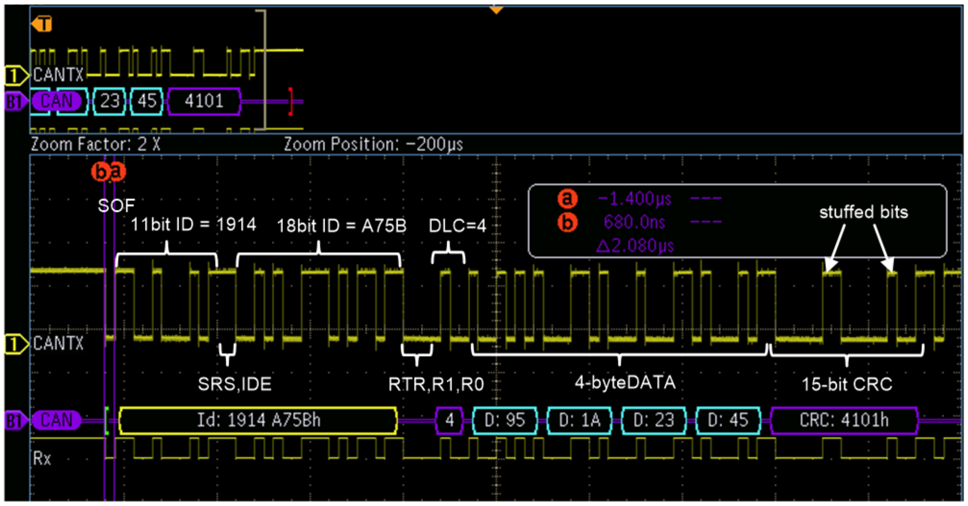


Figure 4-4. Reference Data Frame (Extended MSGID) With No Errors Induced, DLC = 4

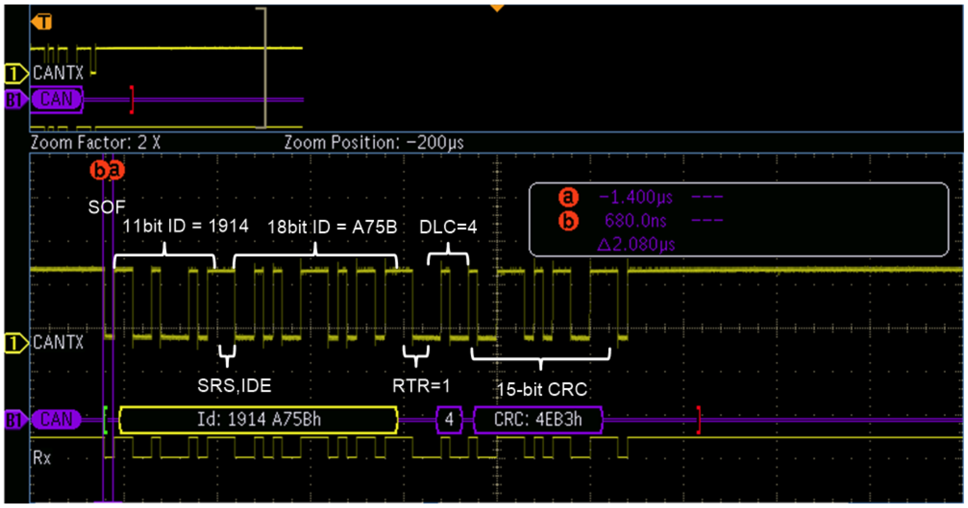


Figure 4-5. Reference Remote Frame (Extended MSGID) With No Errors Induced, DLC = 4

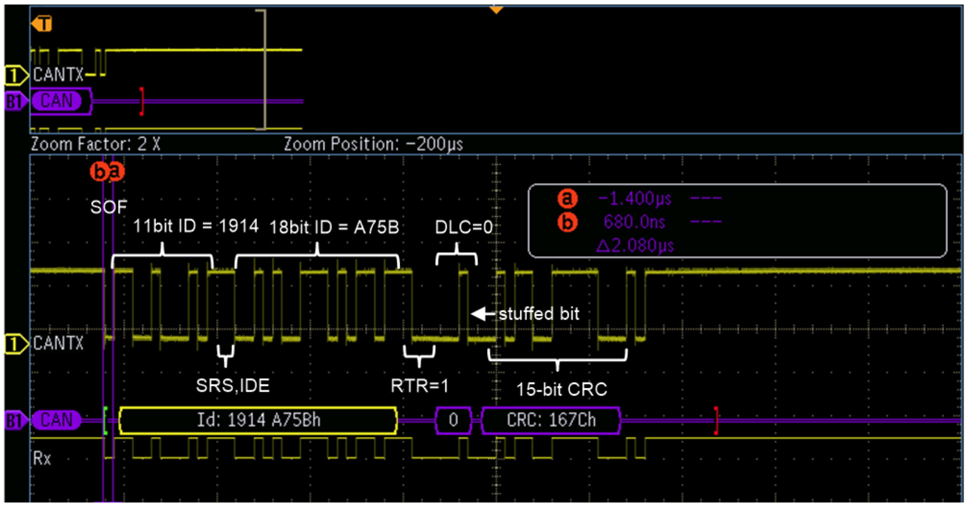


Figure 4-6. Reference Remote Frame (Extended MSGID) With No Errors Induced, DLC = 0

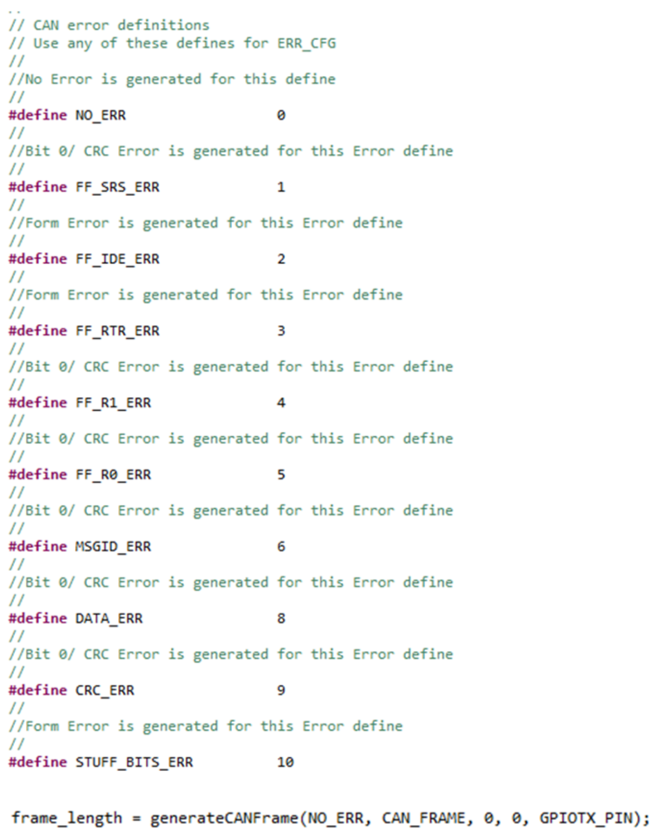
5 Error Generation

This section explains how to generate different type of CAN bus errors.

5.1 GPIO/CCS Method

Figure 5-1 illustrates error generation with the GPIO/CCS method.

- The type of error that can be generated is listed in the *#define* headers of the CCS project.
- To generate a specific error, update the function call *generateCANFrame()* with the error type. By default, NO_ERR is the first argument to the function call, meaning no errors will be generated. The last two arguments of the function *generateCANFrame()* are BITNERR (bit position) and DATABYTENUM (byte number) respectively. The 'bit position' is used for MSGID_ERR, DATA_ERR and CRC_ERR. The 'byte number' is only used with DATA_ERR.
- Meaning of the error definitions:
  – FF_SRS_ERR – SRS bit is flipped
  – FF_IDE_ERR – IDE bit is flipped
  – FF_RTR_ERR – RTR bit is flipped
  – FF_R1_ERR – Reserved bit 'r1' is flipped
  – FF_R0_ERR – Reserved bit 'r0' is flipped
  – MSGID_ERR – 'bit position' in MSGID is flipped
  – DATA_ERR – 'bit position' in data field pointed to by 'byte number' is flipped
  – CRC_ERR – 'bit position' in CRC field is flipped
  – STUFF_BITS_ERR – the function will not generate stuff bits when 5 consecutive bits of the same state occurs.



```
// CAN error definitions
// Use any of these defines for ERR_CFG
//
//No Error is generated for this define
//
#define NO_ERR                  0
//
//Bit 0/ CRC Error is generated for this Error define
//
#define FF_SRS_ERR              1
//
//Form Error is generated for this Error define
//
#define FF_IDE_ERR              2
//
//Form Error is generated for this Error define
//
#define FF_RTR_ERR              3
//
//Bit 0/ CRC Error is generated for this Error define
//
#define FF_R1_ERR               4
//
//Bit 0/ CRC Error is generated for this Error define
//
#define FF_R0_ERR               5
//
//Bit 0/ CRC Error is generated for this Error define
//
#define MSGID_ERR               6
//
//Bit 0/ CRC Error is generated for this Error define
//
#define DATA_ERR                8
//
//Bit 0/ CRC Error is generated for this Error define
//
#define CRC_ERR                 9
//
//Form Error is generated for this Error define
//
#define STUFF_BITS_ERR          10

frame_length = generateCANFrame(NO_ERR, CAN_FRAME, 0, 0, GPIOTX_PIN);
```

Figure 5-1. Error Generation With GPIO/CCS Method

5.2 LabVIEW Method

Figure 5-2 illustrates error generation with the LabVIEW method.

- In the *Error Generation* area, click the *Forcing Error* entry to display the pull-down options. Choose the error type desired.
- *BITnERR* is used for MSGID_ERR, DATA_ERR and CRC_ERR while *DATA Byte Number* is only used with DATA_ERR.
- Error generation pull-down choices are self-explanatory.
- For flipping bits in MSGID, DATA and CRC fields, use the bit value in *BITnERR*.
- The values in *BITnERR* and *DATA Byte Number* fields are used to determine the bit position and the byte number where the bit flip occurs if DATA is chosen in the pulldown menu.

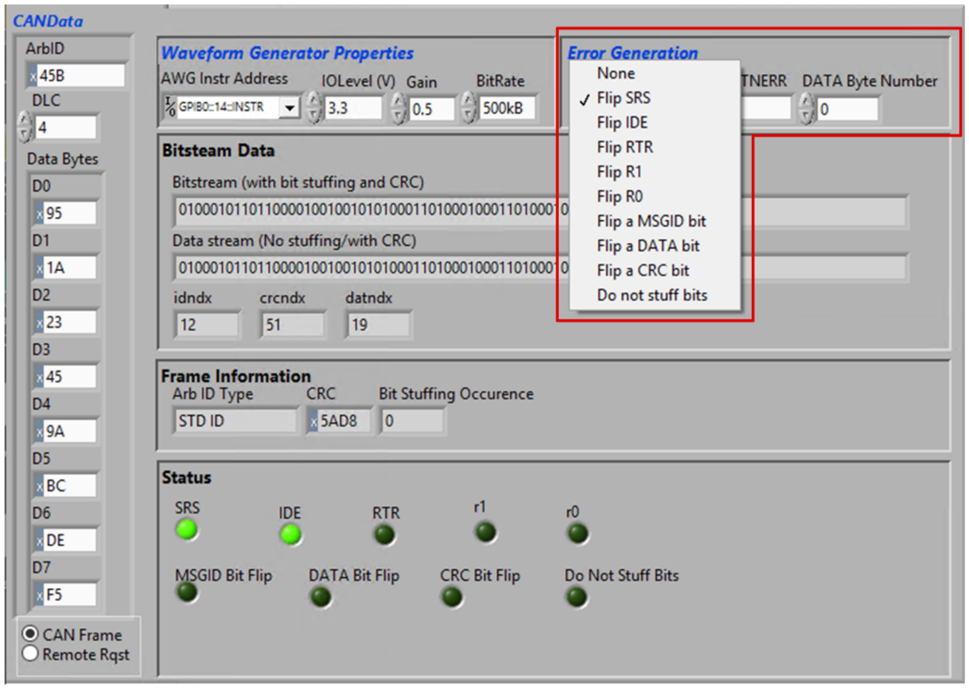


Figure 5-2. Error Generation With LabVIEW Method

6 Emulated Error Frames

This section depicts the emulated error frames. Following illustrations show the various errors generated by the GPIO (or LabVIEW) method while the CAN module in the C2000 device is configured as a receiving node. The last error code (LEC) value in the CANES register is inspected on the receiving module to illustrate how the error-induced CAN frames are interpreted by the CAN module.

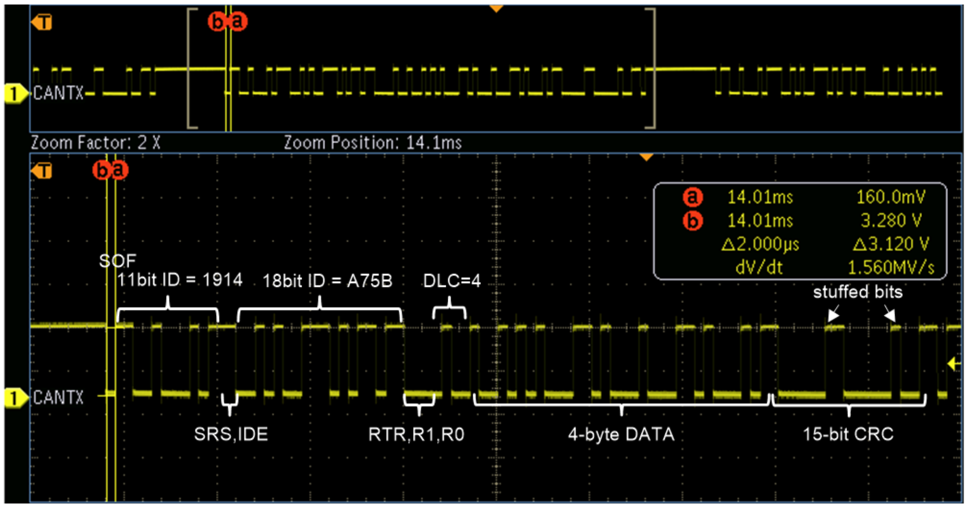


Figure 6-1. Extended ID With No Errors Induced (LEC = 0; No Error)

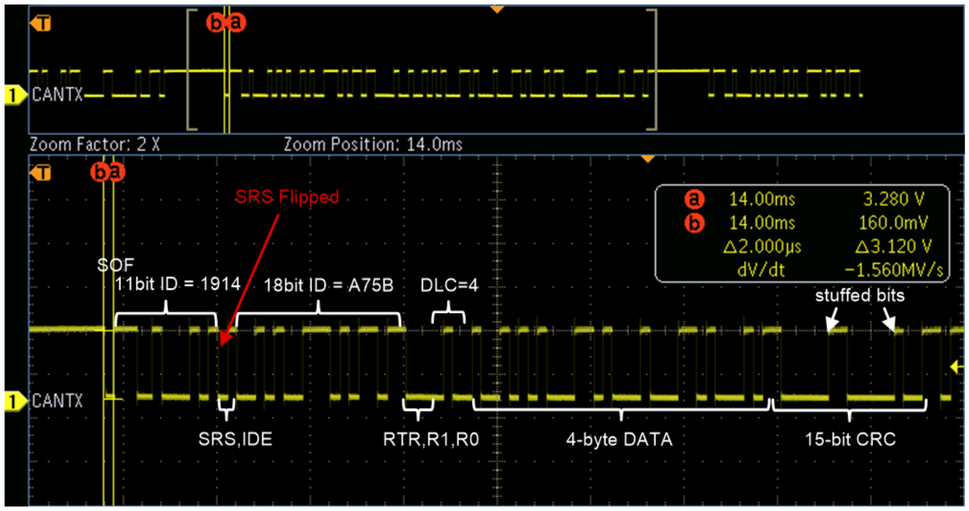


Figure 6-2. Extended ID With SRS Bit Flipped (1 to 0)

Note

LEC = 6; CRC error, not form error is flagged because the receiver accepts 1 or 0 for the reserved bits.

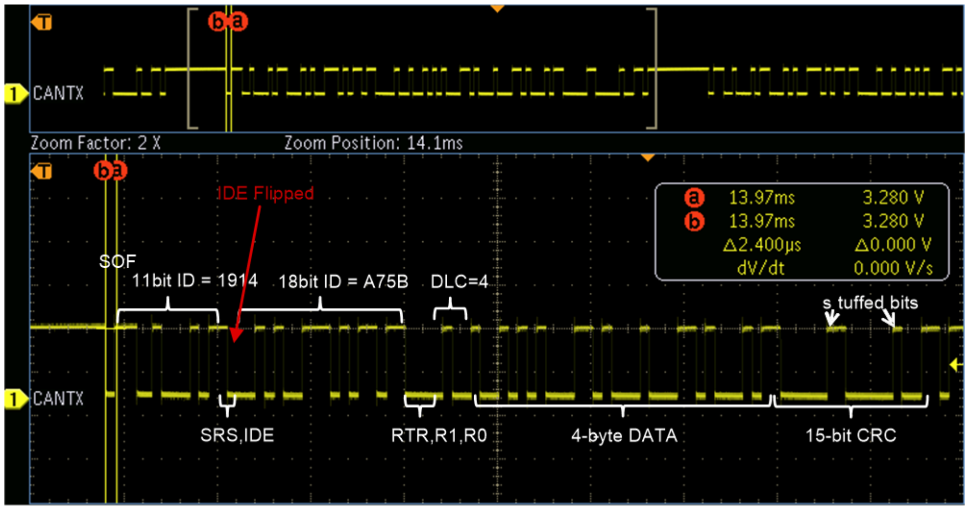


Figure 6-3. Extended ID With IDE Bit Flipped

Note

LEC = 2; Form error, because receiver expects IDE to be 1.

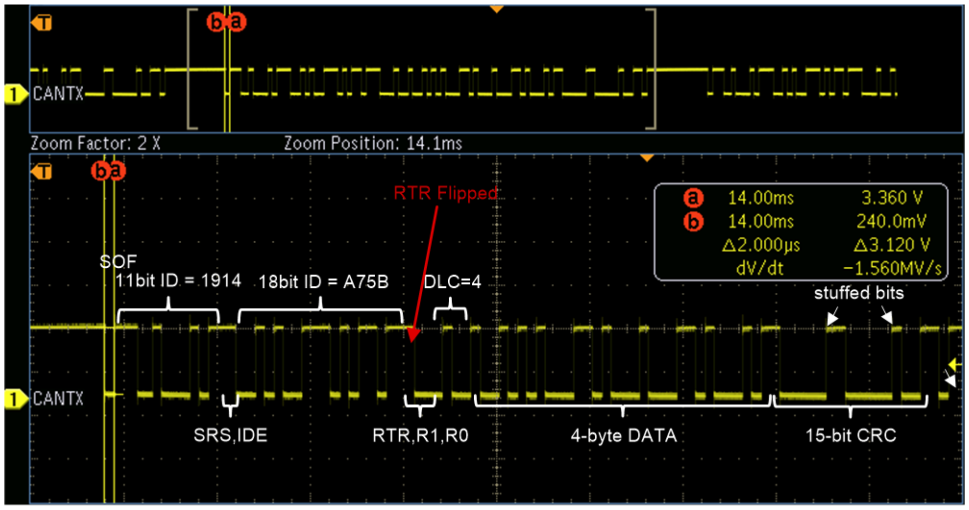


Figure 6-4. Extended ID With RTR Bit Flipped

Note

LEC = 2; Form error, because the receiver was expecting a 'remote frame', but the data bytes corrupted the fixed-form bit fields like CRC delimiter and EOF.

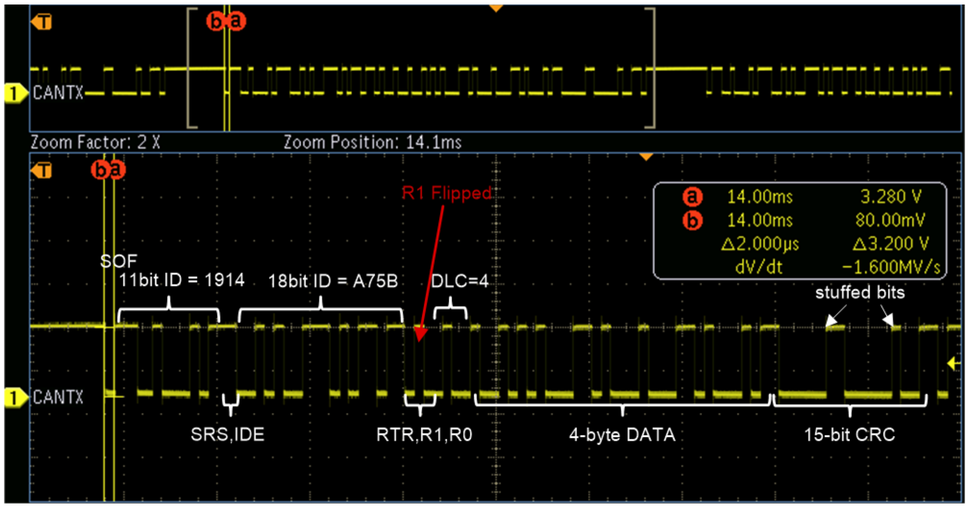


Figure 6-5. Extended ID With r1 Bit Flipped

Note

LEC = 6; CRC error, since the receiver will accept a 0 or a 1 for r1. However, the calculated CRC is not matching.

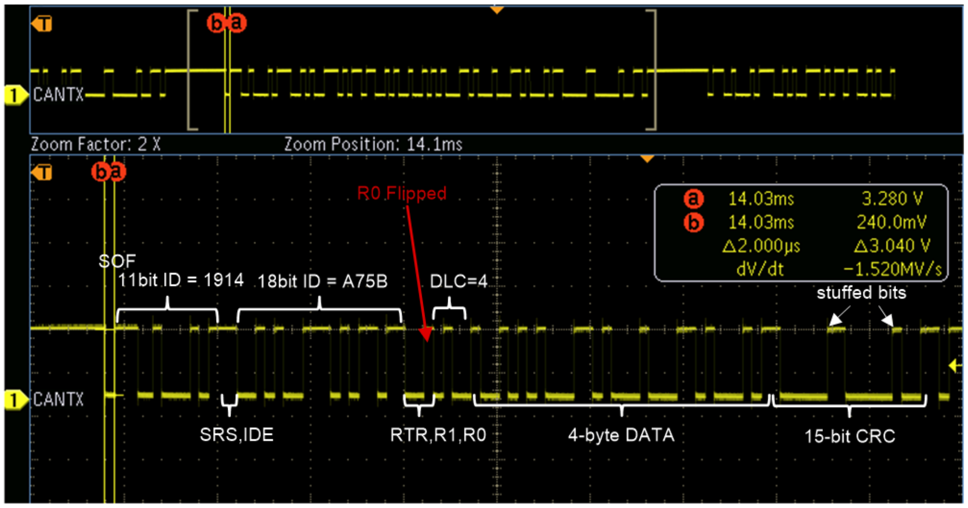


Figure 6-6. Extended ID With r0 Bit Flipped

Note

LEC = 6; CRC error, since the receiver will accept a 0 or a 1 for r0. However, the calculated CRC is not matching.

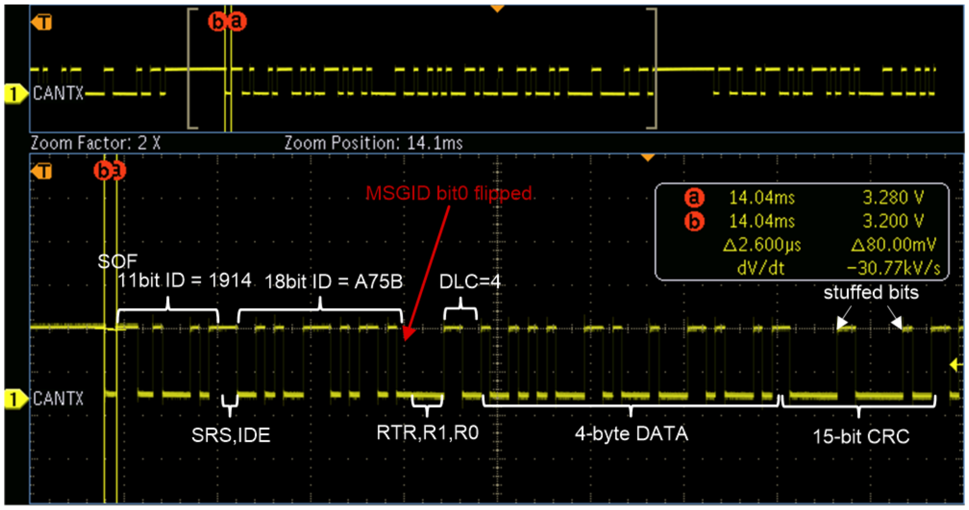


Figure 6-7. Extended ID With Bit0 of MSGID Flipped (LEC = 6; CRC Error)

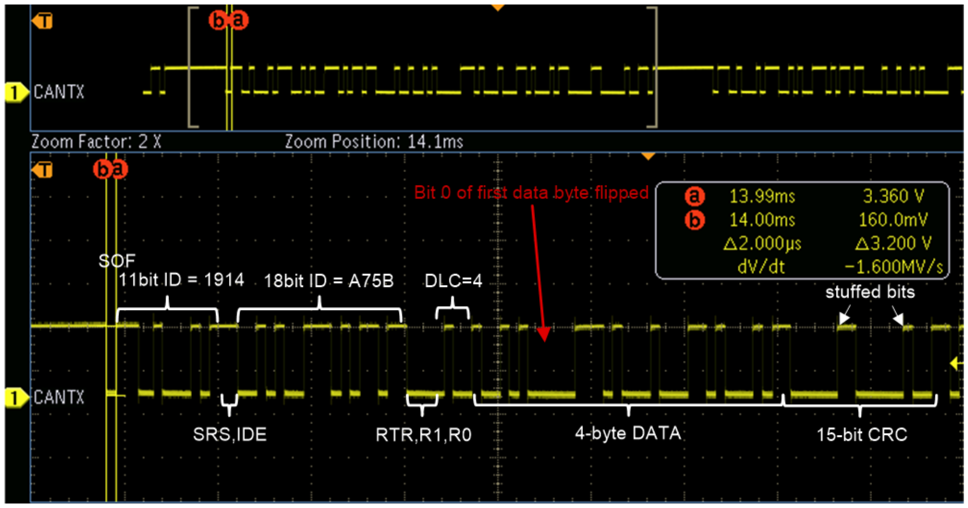


Figure 6-8. Extended ID With Bit0 of Data0 Flipped (LEC = 6; CRC Error)

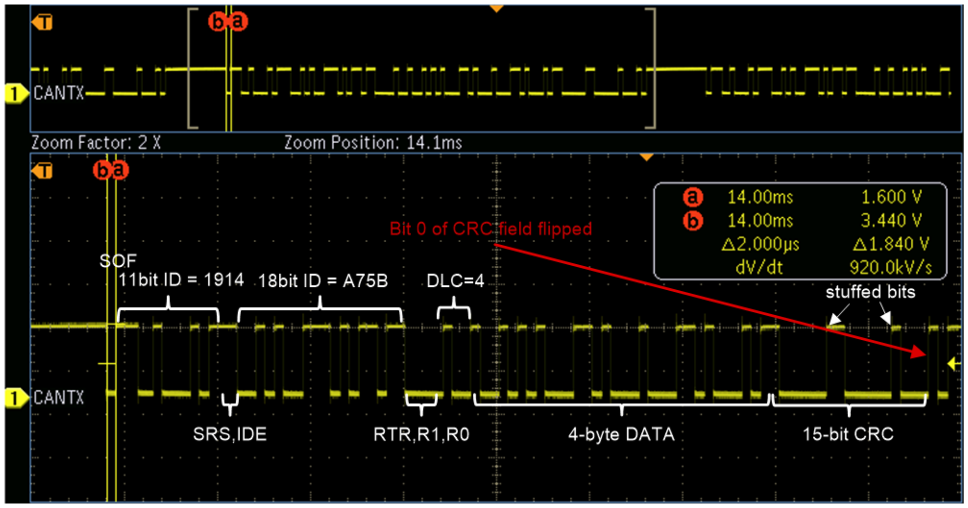


Figure 6-9. Extended ID With Bit0 of CRC Flipped (LEC = 6; CRC Error)

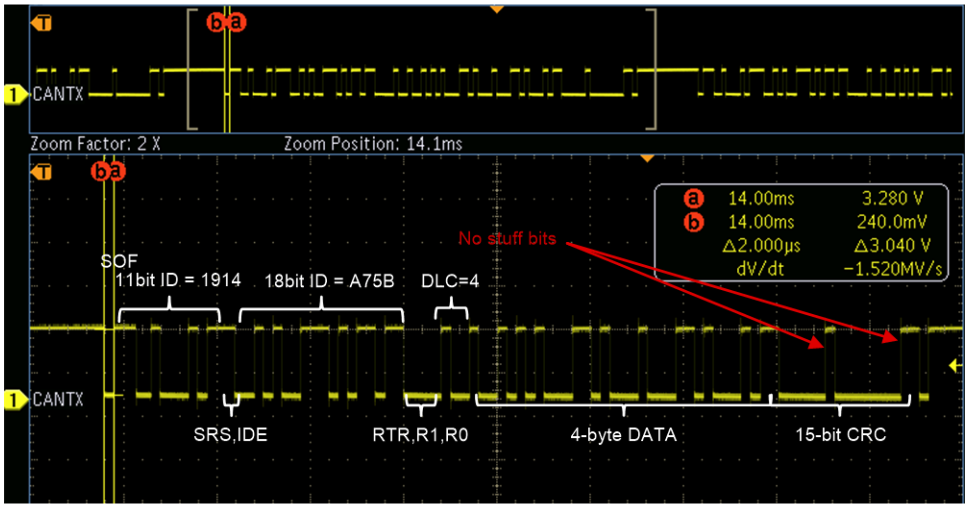


Figure 6-10. Extended ID With No Bit Stuffing (LEC = 1; Stuff Error)

7 References

- Texas Instruments: *Introduction to the Controller Area Network (CAN)*
- Texas Instruments: *Controller Area Network Physical Layer Requirements*
- Texas Instruments: *Basics of debugging the controller area network (CAN) physical layer*
- Texas Instruments: *Calculator for CAN Bit Timing Parameters*
- Texas Instruments: *Overview of 3.3V CAN (Controller Area Network) Transceivers*
- Texas Instruments: *Simplify CAN bus implementations with chokeless transceivers*
- Texas Instruments: *Critical Spacing of CAN Bus Connections*
- Texas Instruments: *Improved CAN network security with TI's SN65HVD1050 transceiver*
- Texas Instruments: *Message priority inversion on a CAN bus*
- Texas Instruments: *Piccolo MCU CAN Module Operation Using the On-Chip Zero-Pin Oscillator*
- Texas Instruments: *C2000 Real-Time Control MCU Peripherals Reference Guide*
- Texas Instruments: *Programming Examples for the TMS320x28xx eCAN*
- Texas Instruments: *Programming Examples and Debug Strategies for the DCAN Module*