HOW-TO

How-to Calculate CAN Checksums in ADU

Document version: 1.0
Software version: 122.0 or later
Published on: 26 March 2026

1. Description

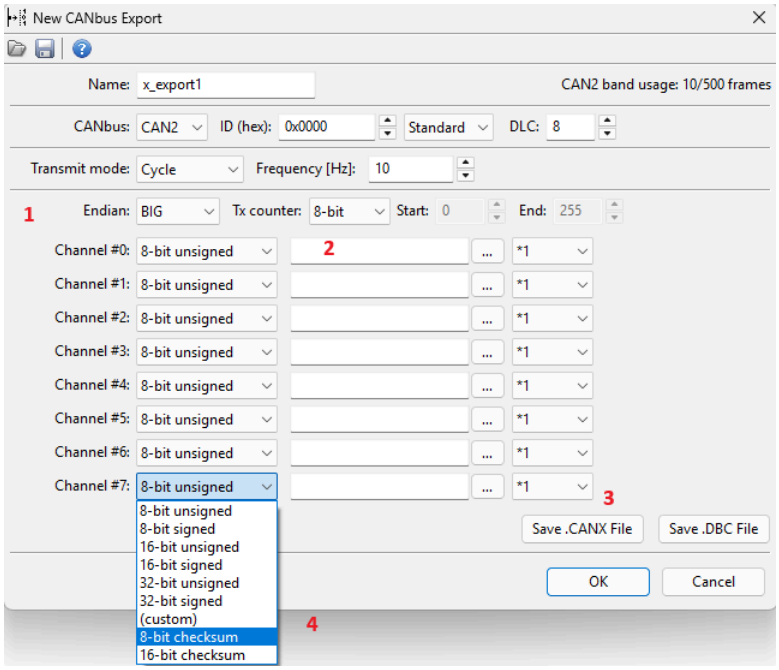
1. Introduction

With ADU Client and Firmware version 122.0, a new module was introduced to allow the definition of custom checksums.

The module enables calculation of a checksum that is appended to the CAN frame at a selected position.

2. *CANBus Export* Window Description

Only the new elements that have been added to the *CANBus Export* window are described below. The remaining elements are described in the ADU Manual: https://www.ecumaster.com/files/ADU/adu_manual_en.pdf



Endian setting (1)

An option for selecting byte order (*Endian*) has been added.

The option provides three possible settings:

- ***BIG*** - big-endian byte order (Motorola).
- ***little*** - little-endian byte order (Intel).
- ***mixed*** - legacy behavior that allows selecting little-endian or big-endian byte order for each channel individually, instead of applying it to the entire CAN frame.

> **Warning:**
> Selecting the *mixed* option disables the checksum calculation feature.

The default *Endian* setting is ***BIG***.

When importing a project from older version that contains *CANBus Export* elements, the *Endian* setting is automatically changed to ***mixed*** in order to maintain backward compatibility.

Tx Counter (2)

Allows selection of the size of the automatically transmitted frame counter.

The maximum counter size is 8 bits (range 0–255).

The following counter sizes are available:

- *8-bit* (0–255)
- *4-bit* (0–15)
- *2-bit* (0–3)
- *custom*

In ***custom*** mode, the user can define the counter operating range by specifying:

- *Start* value
- *End* value

If the *End* value is smaller than the *Start* value, the counter counts down.

***Edit* Button (Checksum Channels) (3)**

The ***Edit*** button is visible when ***8-bit checksum*** or ***16-bit checksum*** is selected for a given channel (marked as number 3 in the figure).

Pressing the ***Edit*** button opens the checksum editing window (described in detail later - see “4. Checksum Editing”).

Available Channel Types (4)

- ***8-bit signed/unsigned*** - the channel is transmitted as an 8-bit signed or unsigned value.
- ***16-bit signed/unsigned*** - the channel is transmitted as a 16-bit signed or unsigned value.
- ***32-bit signed/unsigned*** - the channel is transmitted as a 32-bit signed or unsigned value.
- ***custom*** - allows defining the number of bits used to transmit the channel and selecting the data type (signed or unsigned).
- ***8-bit/16-bit checksum*** - the transmitted value is the checksum defined by the user.

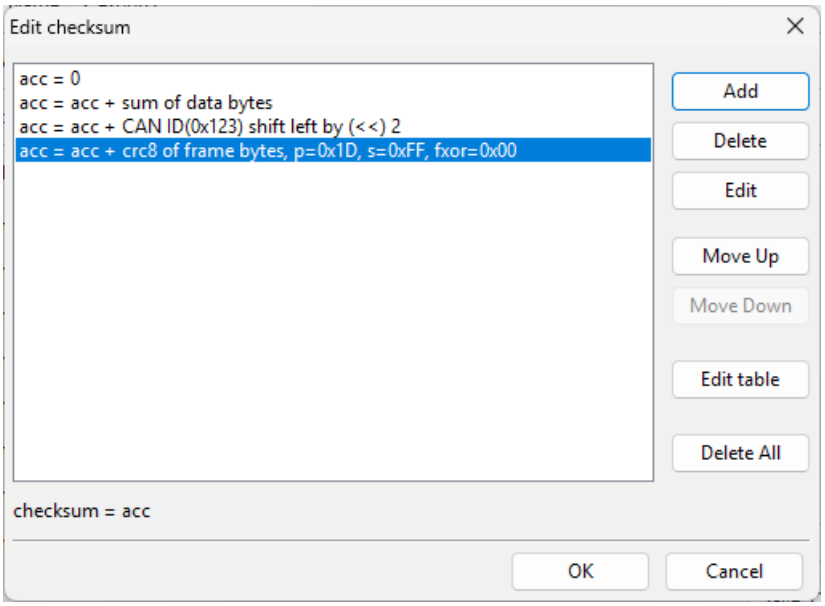
3. Data Placement in the CAN Frame

Data in the CAN frame is arranged from byte 0 to byte 7.

- Eight 8-bit channels occupy bytes 0 through 7 sequentially.
- Four 16-bit channels occupy the following byte pairs:
  (0,1), (2,3), (4,5), (6,7), etc.

4. Checksum Editing

Selecting ***8-bit checksum*** or ***16-bit checksum*** enables checksum editing.



In the checksum editing window, a list of checksum slots is displayed.

Each slot represents an operation performed on the accumulator.

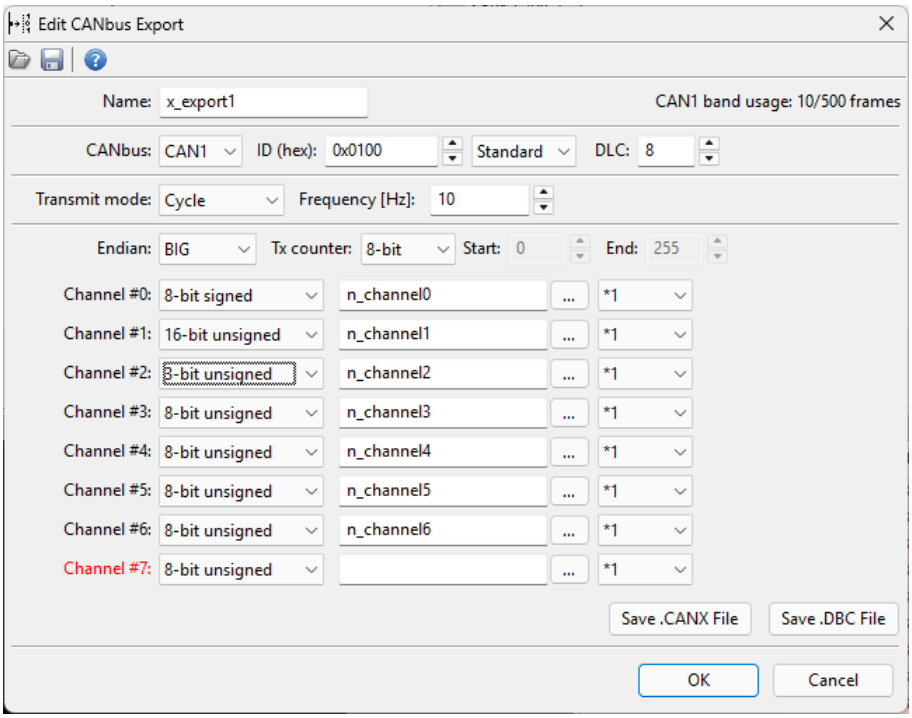
The checksum calculation is always read from top to bottom, following the operation order.

- ***Add*** - adds a new operation.
- ***Delete*** - deletes the selected operation.
- ***Edit*** - edits the selected operation.
- ***Edit table*** - edits a special table of constant values defined by the user.
- ***Move Up / Move Down*** - changes the execution order of operations.
- ***Delete All*** - removes all operations.

5. Custom Bit-Length Channels

If a ***custom*** channel uses a bit length that is not a multiple of 8, all channels that follow the *custom* channel are placed immediately after its data, without byte alignment.

Example 1:

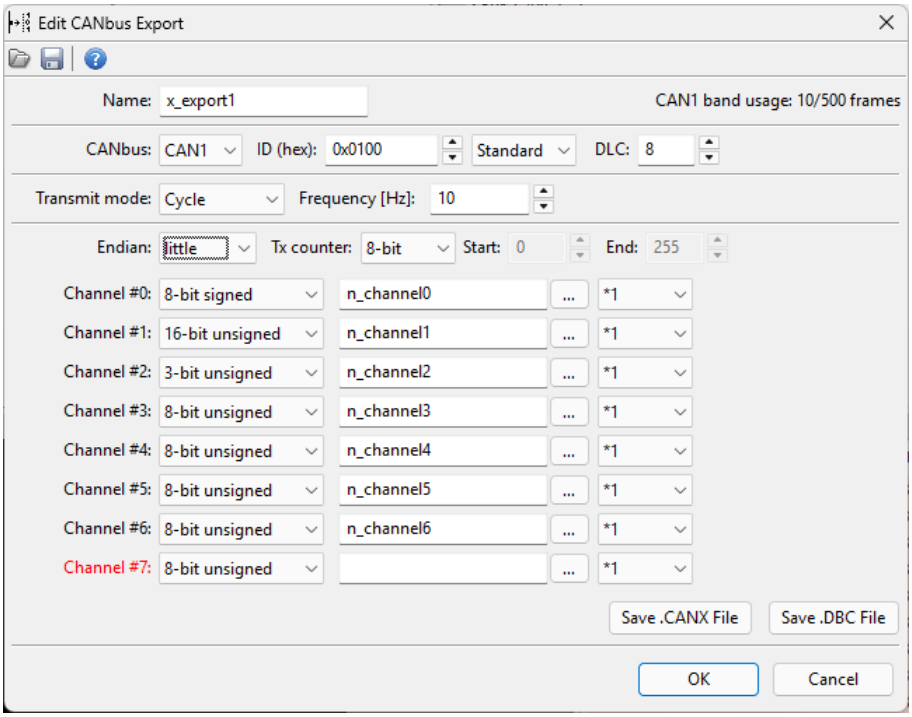


With the data arranged as shown in the figure above, the CAN frame layout will be as follows:

Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7 MSB	6	5	4	3	2	1	0 LSB
n_channel1	1	15 MSB	14	13	12	11	10	9	8
n_channel2	2	23	22	21	20	19	18	17	16 LSB
n_channel3	3	31 MSB	30	29 LSB	28 MSB	27	26	25	24
n_channel4	4	39	38	37 LSB	36 MSB	35	34	33	32
n_channel5	5	47	46	45 LSB	44 MSB	43	42	41	40
n_channel6	6	55	54	53 LSB	52 MSB	51	50	49	48
	7	63	62	61 LSB	60	59	58	57	56

Example 2:

Changing the byte order (*Endian*) from *BIG* to *little*.



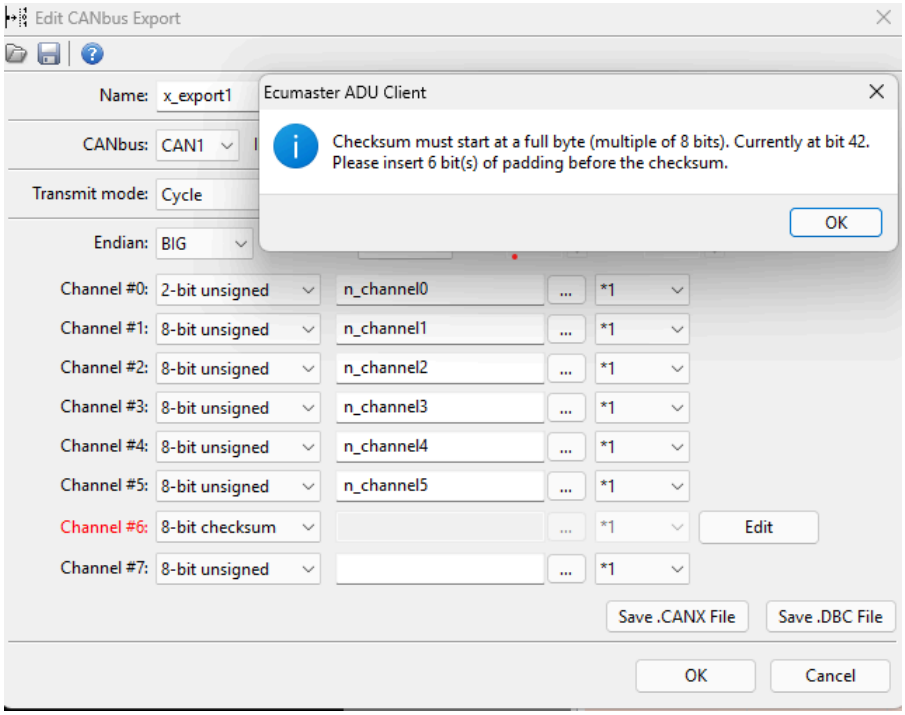
With the data arranged as shown in the figure above, the CAN frame layout will be as follows:

Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7 MSB	6	5	4	3	2	1	0 LSB
n_channel1	1	15	14	13	12	11	10	9	8 LSB
n_channel2	2	23 MSB	22	21	20	19	18	17	16
n_channel3	3	31	30	29	28	27 LSB	26 MSB	25	24 LSB
n_channel4	4	39	38	37	36	35 LSB	34 MSB	33	32
n_channel5	5	47	46	45	44	43 LSB	42 MSB	41	40
n_channel6	6	55	54	53	52	51 LSB	50 MSB	49	48
	7	63	62	61	60	59	58 MSB	57	56

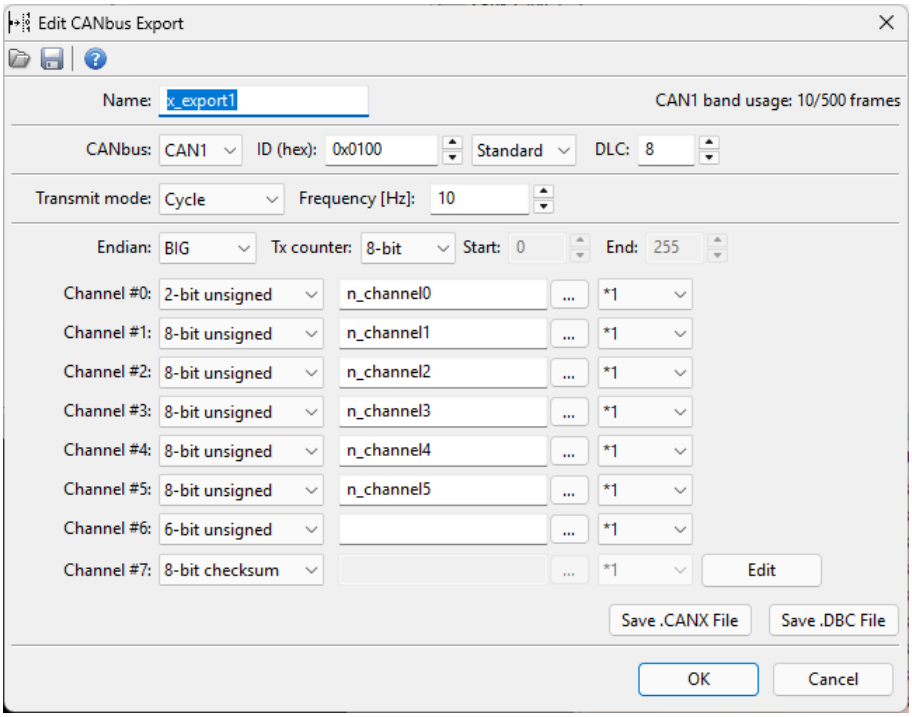
6. Checksum byte alignment

The checksum must be byte-aligned, meaning its start bit must be 0 or a multiple of 8.

If the checksum is not byte-aligned, the channel text will turn red, and an explanatory message will be shown after clicking **OK**.



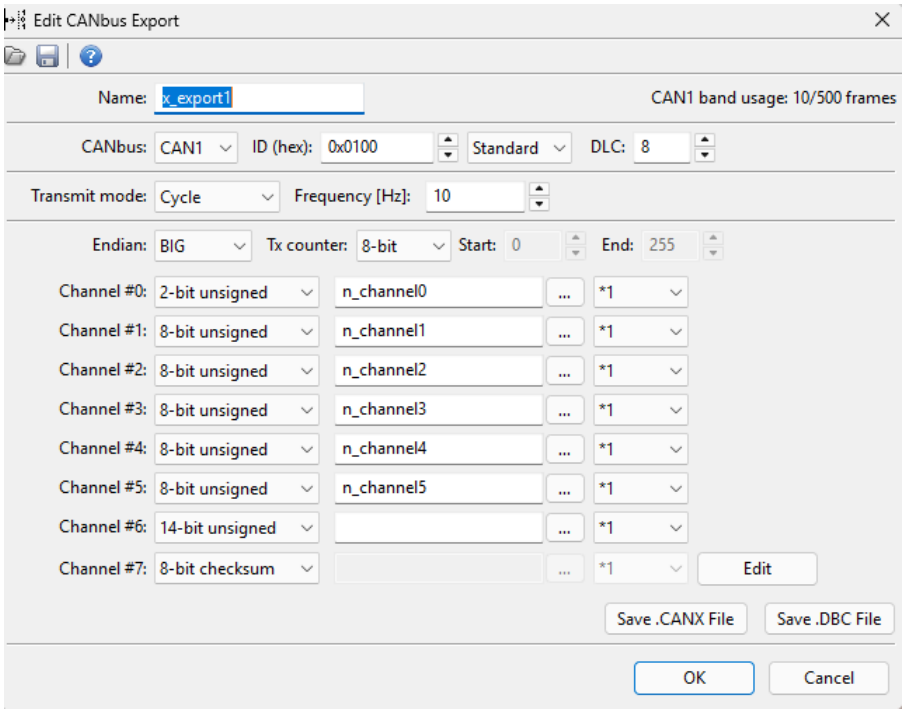
Example 1:



To meet the checksum byte-alignment requirement, 6 bits of padding are applied at Channel #6. For the CAN Bus Export settings described above, the following CAN frame layout is used:

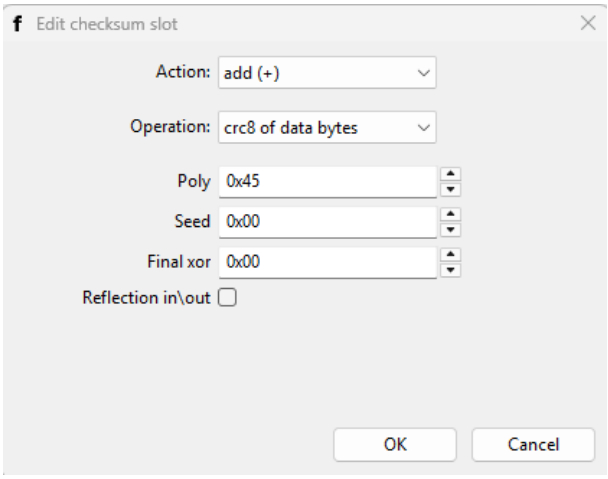
Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7 MSB	6 LSB	5 MSB	4	3	2	1	0
n_channel1	1	15	14 LSB	13 MSB	12	11	10	9	8
n_channel2	2	23	22 LSB	21 MSB	20	19	18	17	16
n_channel3	3	31	30 LSB	29 MSB	28	27	26	25	24
n_channel4	4	39	38 LSB	37 MSB	36	35	34	33	32
n_channel5	5	47	46 LSB	45 MSB	44	43	42	41	40 LSB
padding	6	55 MSB	54	53	52	51	50	49	48 LSB
8bit checksum	7	63	62	61	60	59	58	57	56

If the checksum must be at byte 7 - then the padding must be 14 instead of 6, as shown in the examples below.



Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7 MSB	6 LSB	5 MSB	4	3	2	1	0
n_channel1	1	15	14 LSB	13 MSB	12	11	10	9	8
n_channel2	2	23	22 LSB	21 MSB	20	19	18	17	16
n_channel3	3	31	30 LSB	29 MSB	28	27	26	25	24
n_channel4	4	39	38 LSB	37 MSB	36	35	34	33	32
n_channel5	5	47	46 LSB	45 MSB	44	43	42	41	40
padding	6	55	54	53	52	51	50	49	48 LSB
8bit checksum	7	63 MSB	62	61	60	59	58	57	56 LSB

7. Creating and Editing Operations



In the operation creation and editing window, the user selects an *Action* and an *Operation* available for that *Action*.

The *Action* defines how the accumulator is modified after the operation is executed.

Available Actions

- ***apply/set (=)*** - the result of the operation is assigned as the new accumulator value.
- ***add (+)*** - the result of the operation is added to the accumulator.
- ***xor (^)*** - the result of the operation is combined with the current accumulator value using the XOR operation, forming a new accumulator value.

All values can be entered in decimal or hexadecimal format.

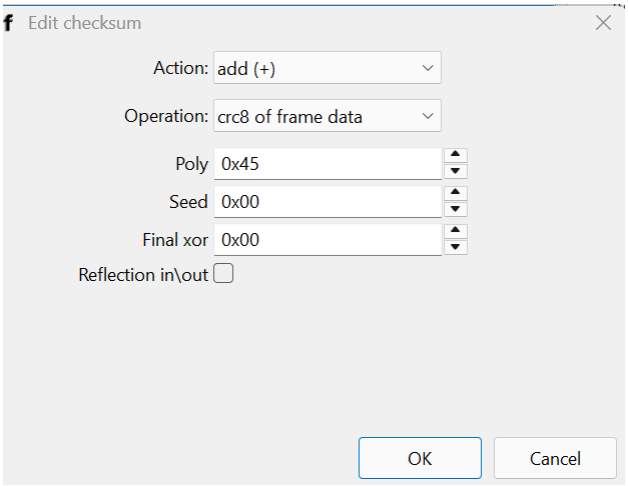
Hexadecimal values must be prefixed with 0x.

Depending on the selected *Operation*, values are displayed in:

- decimal format
- hexadecimal format
- or mixed format, for example: 1 (0x01)

Operations available for *add* and *xor* Actions

- ***constant value*** - a constant value is added to the accumulator or XORed with it.
- ***sum of data bytes*** - all bytes of the CAN frame, except the bytes reserved for the checksum, are summed.
  The resulting value is then added to or XORed with the accumulator.
- ***xor of data bytes*** - all bytes of the CAN frame, except the bytes reserved for the checksum, are XORed together
  (byte0 XOR byte1 XOR byte2 … XOR byteX).
  The resulting value is then added to or XORed with the accumulator.
- ***crc8 of data bytes***



The ***crc8 of data bytes*** operation calculates a CRC-8 checksum from all data bytes except the checksum bytes.
The operation allows configuring algorithm parameters, enabling implementation of most common CRC-8 variants.

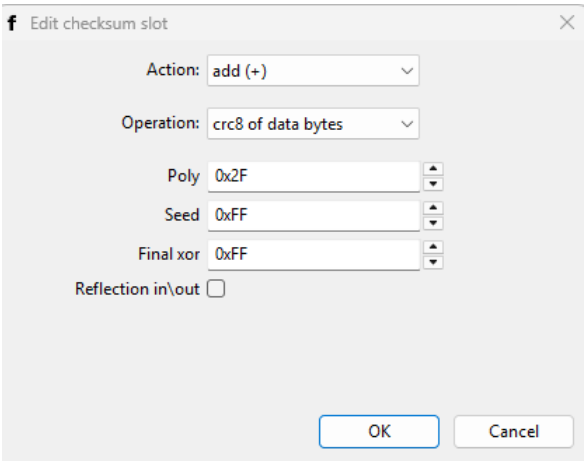
The result is added to or XORed with the accumulator.
Poly - polynomial used during CRC calculation.
Defines how each subsequent data byte affects the intermediate and final CRC result.
Seed - initial CRC accumulator value from which CRC calculation starts.
Final XOR - value used to XOR the final CRC calculation result.

Reflection in\out

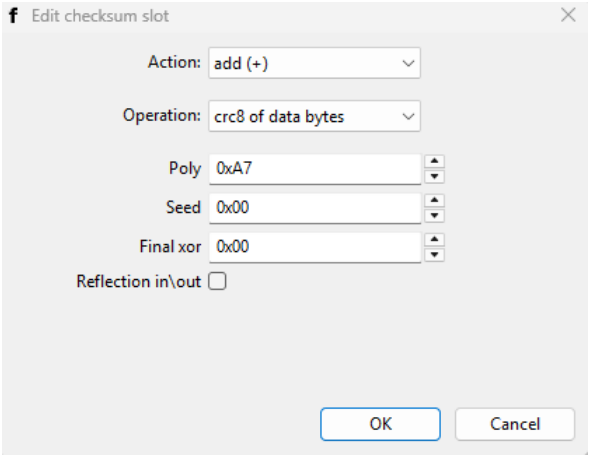
- ***Reflection in*** - reverses the bit order in each input byte before CRC calculation.
- ***Reflection out*** - reverses the bit order of the final CRC result.

Example:

CRC-8/AUTOSAR



CRC-8/BLUETOOTH



- ***CAN ID shift left/right by arg*** - this operation shifts the CAN ID value left or right by a specified number of bits.

  A left bit shift corresponds to multiplication by a power of 2.

  A right bit shift corresponds to division by a power of 2.

argument	Shift left <<	Shift right >>
0	CAN ID *1	CAN ID /1
1	CAN ID *2	CAN ID /2
2	CAN ID *4	CAN ID /4
3	CAN ID *8	CAN ID /8

argument	Shift left <<	Shift right >>
4	CAN ID *16	CAN ID /16
5	CAN ID *32	CAN ID /32
6	CAN ID *64	CAN ID /64
7	CAN ID *128	CAN ID /128
8	CAN ID *256	CAN ID /256

The result is added to or XORed with the accumulator.

- ***counter shift left/right by arg*** - this operation is analogous to CAN ID shift, with the difference that the bit shift is performed on the current transmitted frame counter value.
  The result is added to or XORed with the accumulator.
- ***acc shift left/right by arg*** - this operation is analogous to CAN ID shift, with the difference that the bit shift is performed on the current accumulator value.
  The result is added to or XORed with the accumulator.
- ***(counter + arg1) * arg2*** - this operation adds value *arg1* to the current counter value and then multiplies the result by *arg2*.
  The operation can also be used:
  - to only add a value to the counter (*arg2* = 1),
  - to only multiply the counter value (*arg1* = 0).

  The result is added to or XORed with the accumulator.
- ***table value[counter]*** - the result of the operation is a value taken from the lookup table (described in detail in "7. Lookup Table Editing"), selected based on the current transmitted frame counter value.
  Table index calculation:
  - **When the counter counts up:**
    index = (counter value − counter start value) modulo 16
  - **When the counter counts down:**
    index = (counter start value − counter value) modulo 16

  The result is added to or XORed with the accumulator.

Operations available for *apply/set* Action

- ***crc8 of accumulator*** - calculates CRC-8 from the current accumulator value.
  The result is directly assigned to the accumulator.

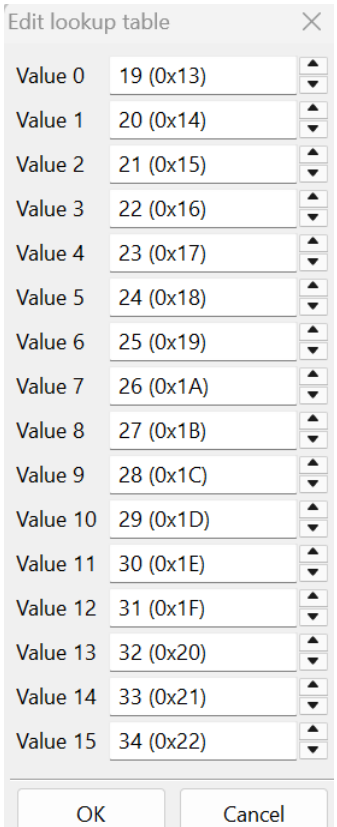
- ***crc8 of table val(seed: acc)*** - calculates CRC-8 from the lookup table value, using the current accumulator value as the initial seed.
  The result is assigned to the accumulator.
- ***mask*** - performs bit masking on the current accumulator value.
  Example:
  acc = 0x12
  To preserve only the four least significant bits of the accumulator, bit masking using the AND operation is performed.
  acc = (acc)0x12 & (mask)0x0F
  acc = 0x02
- ***acc shift left/right by arg*** - performs a bit shift on the accumulator value and assigns the result directly to the accumulator.

8. Lookup Table Editing

Pressing the ***Edit table*** button in the checksum editor allows defining up to 16 lookup table values.

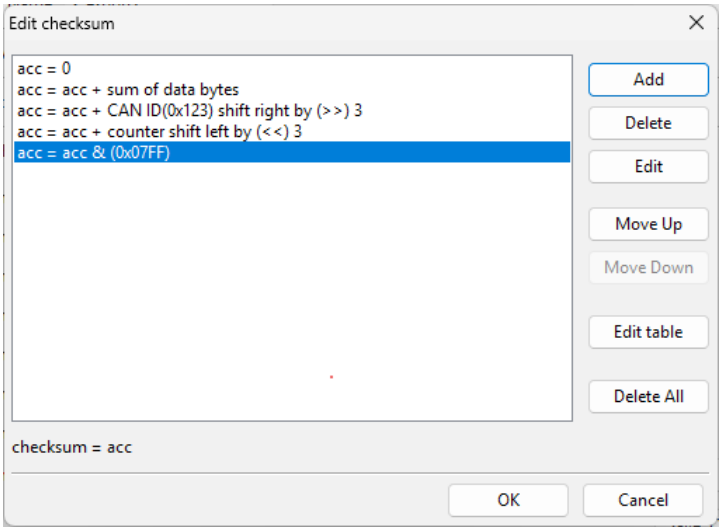
The maximum table size is 16 elements, each with a value in the range 0–255.

If the selected counter operating range is smaller than 16, the table size is automatically limited to the difference between the counter *Start* and *End* values.



9. How to Interpret the Created Checksum

Example:



Byte values: byte0 = 1, byte1 = 5, byte2 = 50, byte3 = 0, byte4 = 15, byte5 = 2

Counter value: 2

The checksum calculation is always read from top to bottom, following the operation order.

- **acc = 0**

  The initial accumulator value is always 0.

- **acc = acc + sum of data bytes,**

  Action: *add*, Operation: *sum of data bytes*

  The sum of the byte values (b0 + b1 + b2 + b3 + b4 + b5) is added to the accumulator:

  acc = 1 + 5 + 50 + 0 + 15 + 2 = 73

- **acc = acc + CAN ID (0x0123) shift right by >> 3**

  Action: *add*, Operation: *CAN ID shift right by arg*

  The CAN ID identifier has the value 0x123.

  A right bit shift by 3 positions corresponds to division by 8:

  0x123 >> 3 that is 0x123 / 8 = 0x24 = 36 (decimal)

  acc = 73 + 36 = 109

- **acc = acc + counter shift left by << 3**

  Action: *add*, Operation: *counter shift left by arg*

  The current counter value is 2.

  A left bit shift by 3 positions corresponds to multiplication by 8:

  counter value = 2 therefore 2 << 3 = 2 * 8 = 16

  acc = 109 + 16 = 125

- **acc = acc & (0x07FF)**

  Action: *apply/set*, Operation: *mask*

  A bit masking operation is performed on the accumulator:

  acc = 125 (0x71) & 0x07FF

  acc = 125

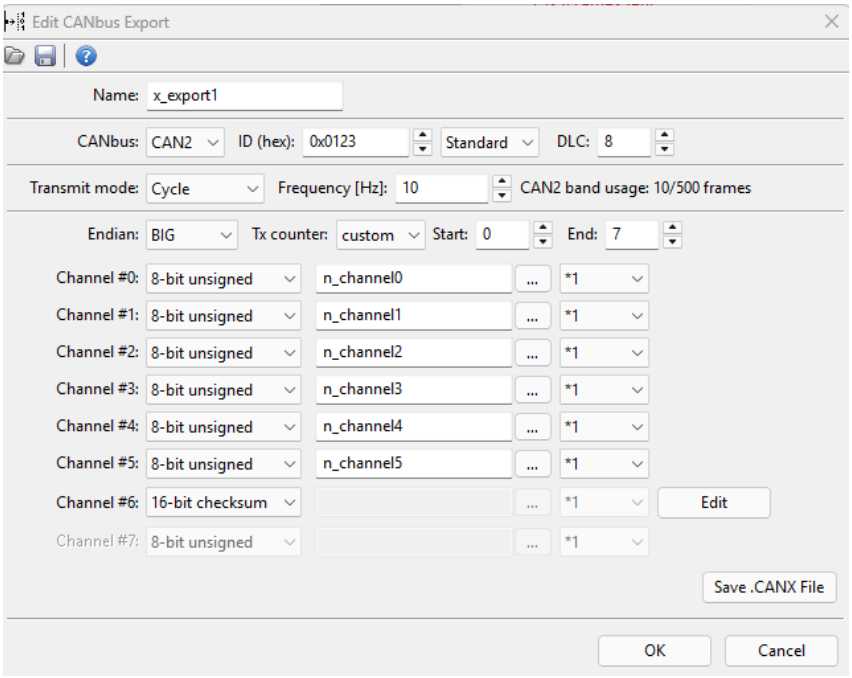
  checksum = acc = 125

2. Examples

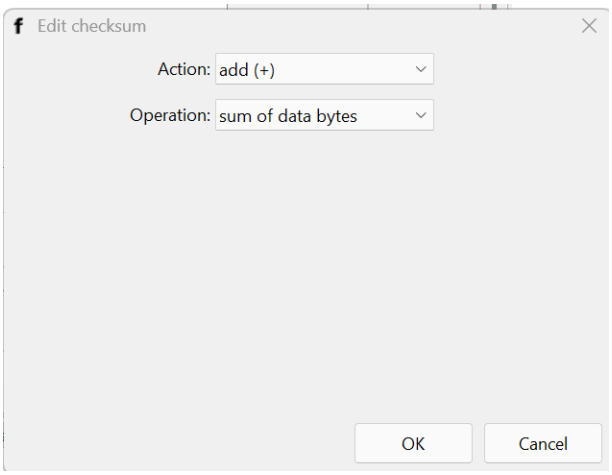
Example 1 – Checksum Creation

The goal is to calculate an 8-bit checksum according to the following equation, using **little-endian**:

checksum = (byte0 + byte1 + byte2 + byte3 + byte4 + byte5 + byte6) XOR 0x55



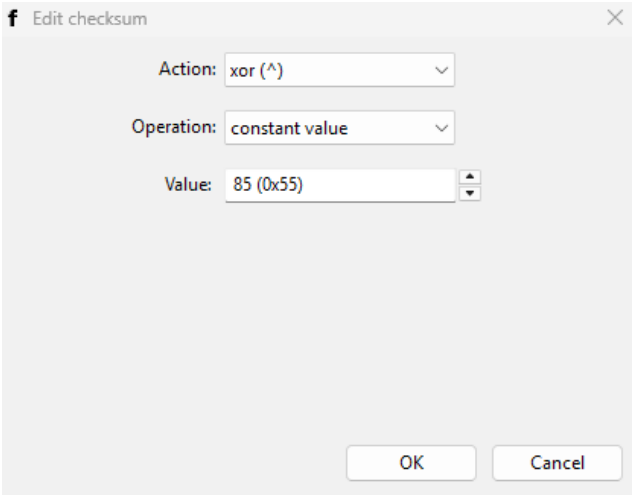
- **The first required operation is:**



After this operation, the accumulator value is:

acc = byte0 + byte1 + byte2 + byte3 + byte4 + byte5 + byte6

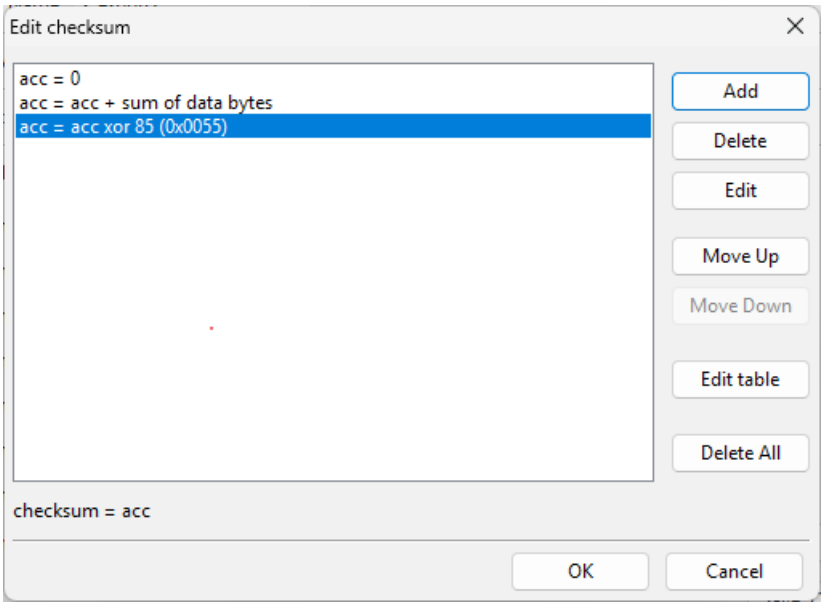
- **Second required operation:**



After this operation, the accumulator value is:

acc = (byte0 + byte1 + byte2 + byte3 + byte4 + byte5 + byte6) XOR 0x55

- **Final result**



The resulting CAN frame layout is shown below:

Legend:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7	6	5	4	3	2	1	0 LSB
n_channel1	1	15 MSB	14	13	12	11	10	9	8
n_channel2	2	23 MSB	22	21	20	19	18	17	16 LSB
n_channel3	3	31 MSB	30	29	28	27	26	25	24 LSB
n_channel4	4	39 MSB	38	37	36	35	34	33	32 LSB
n_channel5	5	47 MSB	46	45	44	43	42	41	40 LSB
8-bit checksum	6	55 MSB	54	53	52	51	50	49	48 LSB
	7	63 MSB	62	61	60	59	58	57	56 LSB

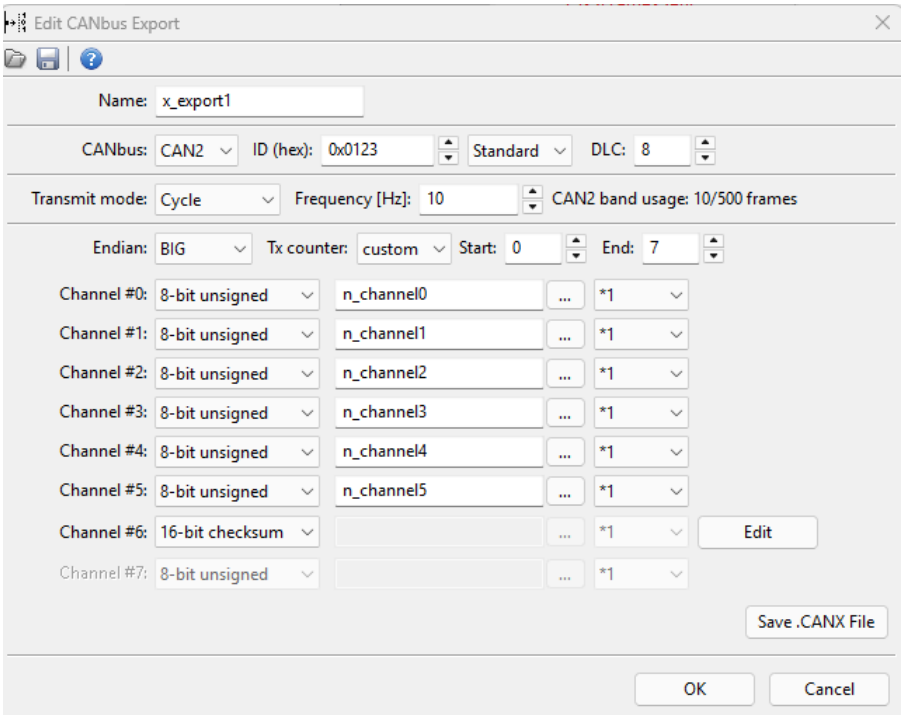
Example 2 – Checksum Creation

The goal is to calculate a checksum according to the following equation:

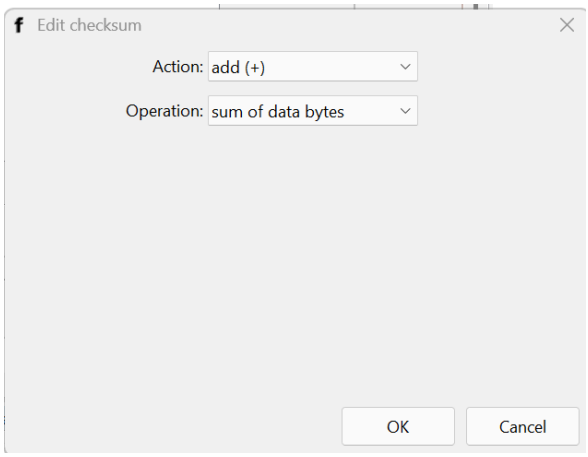
checksum = byte0 + byte1 + byte2 + byte3 + byte4 + byte5 + (CAN ID / 4) + (counter * 4)

Additionally:

- the checksum length is 9 bits,
- the checksum is placed in bytes 6 and 7 of the CAN frame,
- in byte 6, the three most significant bits are used to transmit the frame counter.



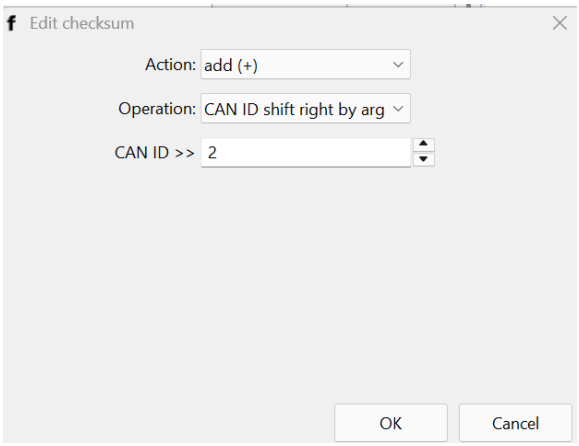
- **The first required operation is:**



After this operation, the accumulator value is:

acc = byte0 + byte1 + byte2 + byte3 + byte4 + byte5

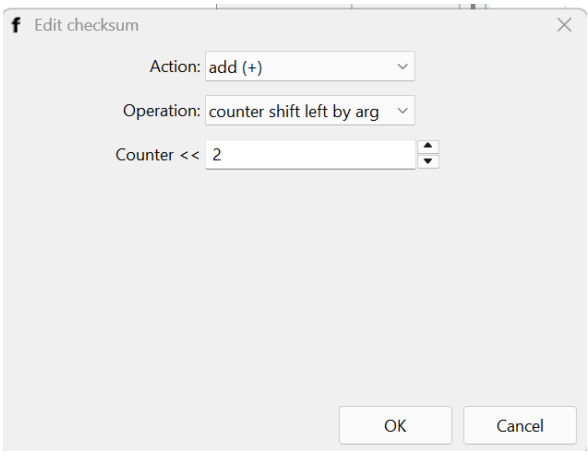
- **The next required operation is:**



After this operation, the accumulator value is:

acc = (byte0 + byte1 + byte2 + byte3 + byte4 + byte5) + (CAN ID / 4)

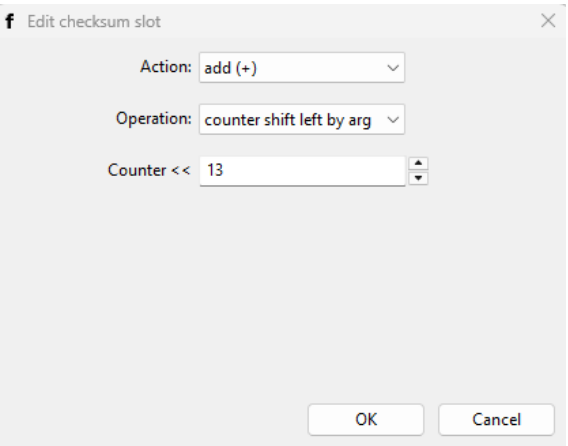
- **Next, the counter value is multiplied using a bit shift operation.**



The accumulator value after this operation is:

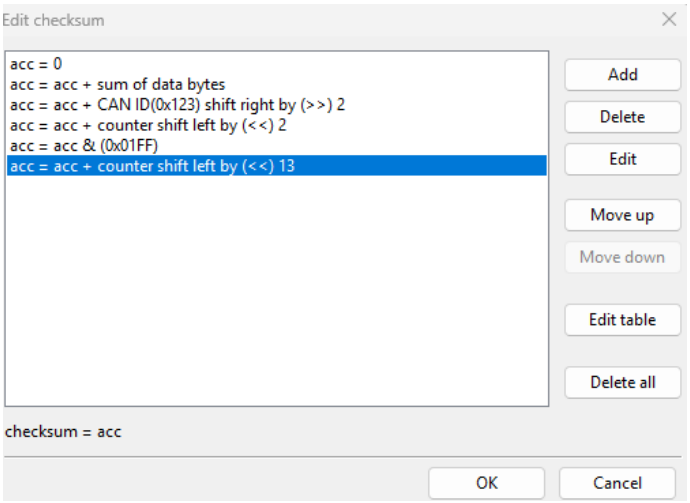
acc = (byte0 + byte1 + byte2 + byte3 + byte4 + byte5) + (CAN ID / 4) + (counter * 4)

- The checksum length is 9 bits, therefore the mask used for masking the accumulator is 0x1FF (binary: 0001 1111 1111).



- Finally, the counter value must be placed in the three most significant bits of the checksum to meet the original requirement: *"In byte 6, the three most significant bits are used to transmit the frame counter."*

Final result:



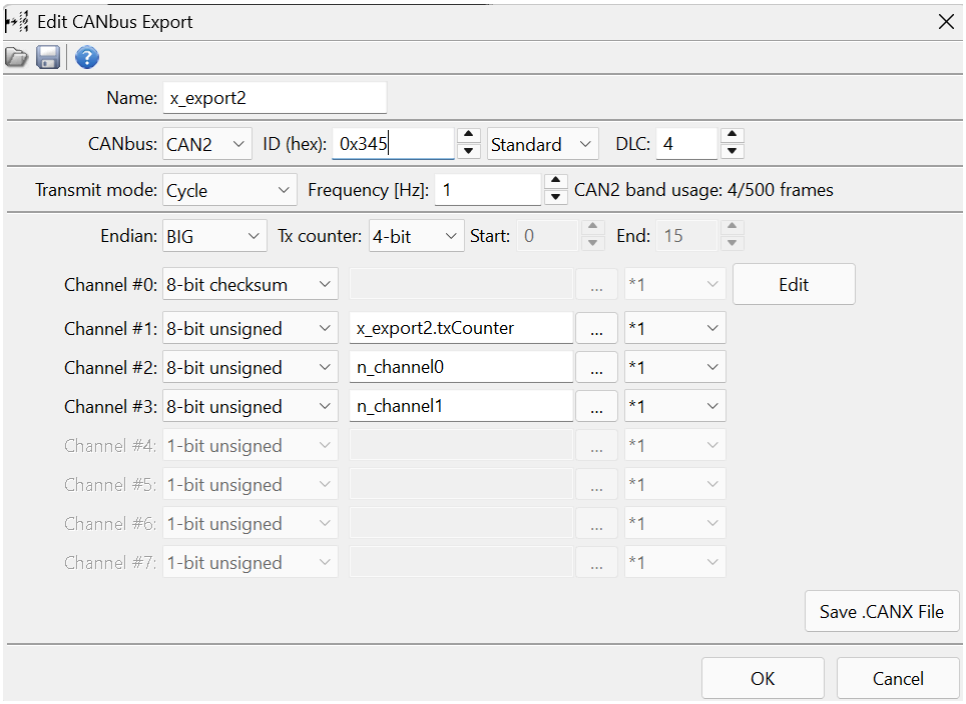
The layout of the CAN frame prepared in this way is shown below.

Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
n_channel0	0	7 MSB	6	5	4	3	2	1	0 LSB
n_channel1	1	15 MSB	14	13	12	11	10	9	8 LSB
n_channel2	2	23 MSB	22	21	20	19	18	17	16 LSB
n_channel3	3	31 MSB	30	29	28	27	26	25	24 LSB
n_channel4	4	39 MSB	38	37	36	35	34	33	32 LSB
n_channel5	5	47 MSB	46	45	44	43	42	41	40 LSB
txCounter	6	55 MSB	54	53 LSB	52	51	50	49	48 MSB
checksum	7	63	62	61	60	59	58	57	56 LSB

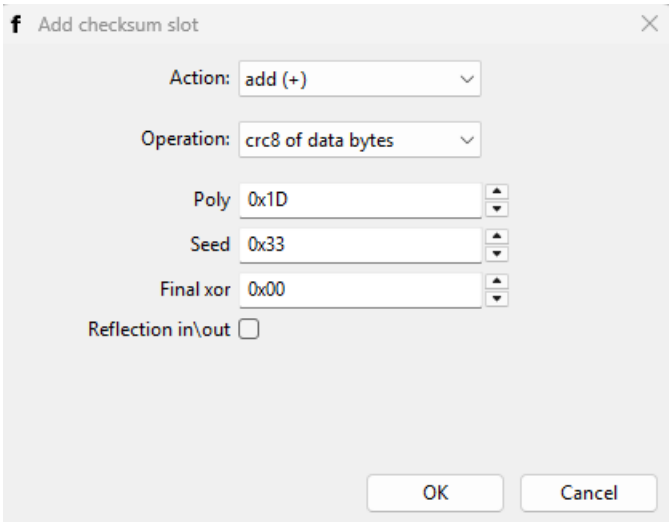
Example 3 – Checksum Creation

The expected CRC-8 checksum with polynomial 0x1D and initial value 0x33 is placed in byte 0 of the CAN frame. Byte 1 contains the transmitted frame counter, counting in the range from 0 to 15 (4-bit counter).

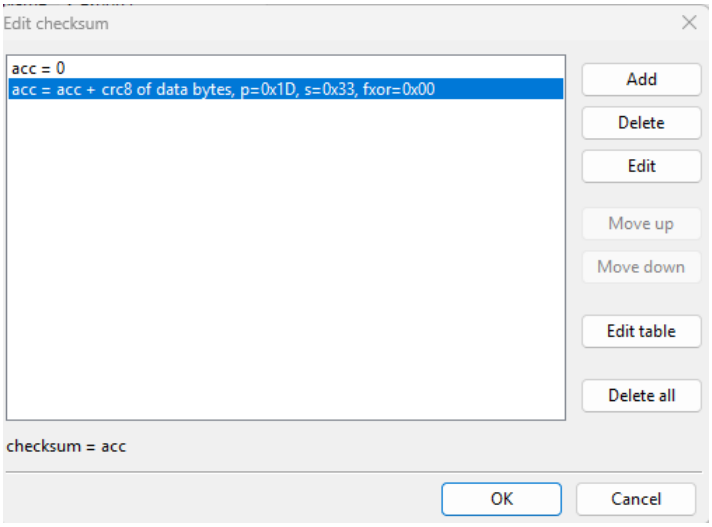
The total frame length is 4 bytes.



Only one operation is required to calculate this checksum.



The CRC-8 checksum is calculated from three bytes: byte1, byte2, and byte3.



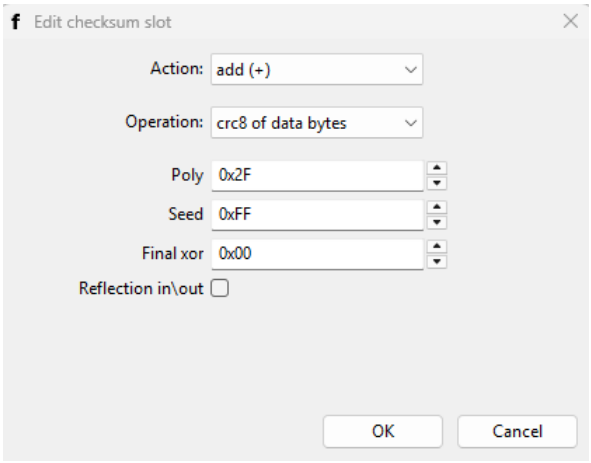
The resulting CAN frame layout is shown below.

Channels:	BYTE	BITS							
		7	6	5	4	3	2	1	0
checksum	0	7 MSB	6	5	4	3	2	1	0 LSB
txCounter	1	15 MSB	14	13	12	11	10	9	8 LSB
n_channel0	2	23 MSB	22	21	20	19	18	17	16 LSB
n_channel1	3	31 MSB	30	29	28	27	26	25	24 LSB
	4	39	38	37	36	35	34	33	32
	5	47	46	45	44	43	42	41	40
	6	55	54	53	52	51	50	49	48
	7	63	62	61	60	59	58	57	56

Example 4 – Checksum Creation

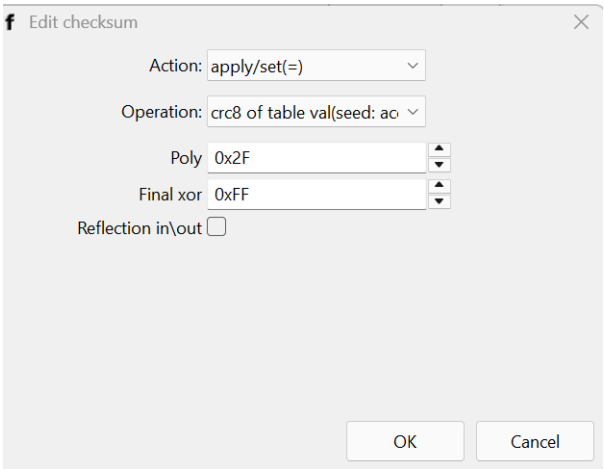
The expected checksum complies with the CRC-8/AUTOSAR algorithm
(*Poly* = 0x2F, *Seed* = 0xFF, *Final xor* = 0xFF),
with an additional byte taken from the lookup table.
The checksum is calculated from eight bytes, where the eighth byte comes from the lookup table:
CRC-8(byte0, byte1, byte2, byte3, byte4, byte5, byte6, lookupTableValue)
To calculate such a checksum, two operations are required.

- First step

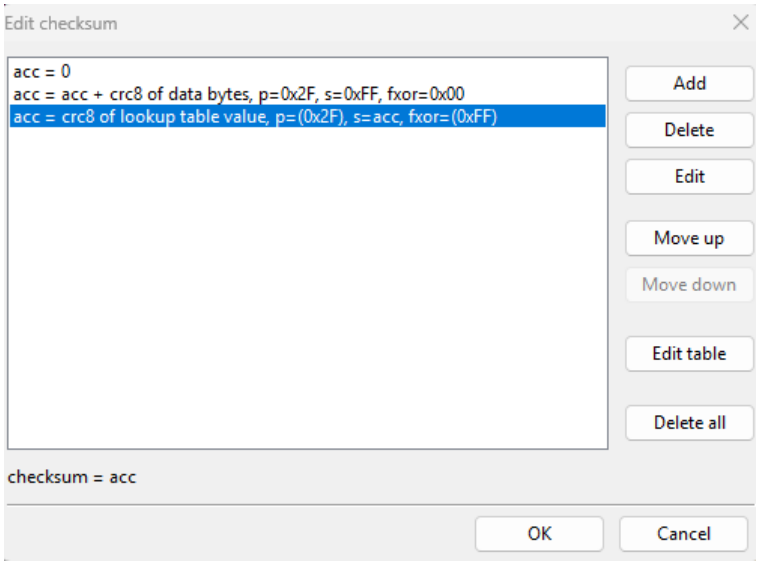


In the first step, the CRC-8 checksum is calculated for bytes byte0–byte6.
Since the additional eighth byte from the lookup table is not yet included, the *Final xor* parameter is not applied and is set to 0x00.
The obtained result is added to the accumulator.

- Second step:



In the second step, the CRC-8 checksum is calculated for the missing byte taken from the lookup table.
The current accumulator value (acc) is used as the initial value (seed).
Finally, a *Final xor* operation with the value 0xFF is performed, in accordance with the CRC-8/AUTOSAR algorithm.



3. Document history

Version	Date	Changes
1.0	2026.03.26	Initial release