TSharkRex™

Programming Language

and platform

Table of contents

Introduction TSharkRex™ Programming language

TSharkRex™ Programming language is a high level programming language aimed to work and function as the Structured Text language according to IEC 61131-3 standard.

TSharkRex™ Programming Language

By reading this manual you will be given a better understanding how TSharkRex™ is working and how to use it to write your own code.

TSharkRex™ Introduction Manual

TSharkRex™ is similar to Structured Text as of the standard IEC 61131-3 but have some small differences. Let's write some code and analyze it after.

```
VAR
        A : BOOL; //It's a variable
        B : BOOL; (* This also *)
END_VAR;

VAR_OUTPUT
        OUT : OUTPUT;
END_VAR;

IF A OR B THEN
        OUT(VALUE := TRUE);
END_IF;
```

The above code is a simple snippets who activates an output (on a XBB PowerUnit™ for instance) when either A or B variable is TRUE. To comment in the code you can just simply use '*//*' or for multi line commenting using '*(* *)*'

TSharkRex™ have a few special declarations to be aware of; *VAR_OUTPUT* and *VAR_SIGNAL*. They are used to direct the work flow of the tool chain when compiling the code when using the Online TSharkRex™ Platform.

Both of them are equal to *VAR* and can hold all types of variable declarations, but the variables will show up as special variables when used on the Online TSharkRex™ Platform.

TSharkRex™ is just as IEC 61131-3 Structured Text case sensitive keep that in mind when writing your code.

Reserved keywords

Some keywords are reserved in TSharkRex™ that is used by either built-in function blocks or functions.

When using libraries in your code all variables, function blocks and functions in the library will be obfuscated and theoretical all of them is considered reserved keywords, but because of the obfuscation you will not likely be able to match the same name of variables or function blocks when writing your code.

The following keywords are reserved and can not to be used as variables.

- IF
- THEN
- END_IF
- CASE
- OF
- END_CASE
- ELSE
- ELSIF
- WHILE
- DO
- END_WHILE
- ARRAY
- TRUE
- FALSE
- NOT
- AND
- OR
- FUNCTION
- END_FUNCTION
- FUNCTION_BLOCK
- END_FUNCTION_BLOCK
- VAR
- VAR_SIGNAL
- VAR_INPUT
- VAR_OUTPUT
- CONSTANT
- END_VAR
- CAN_MODE_CONFIG
- CAN_MODE_NORMAL
- CAN_MODE_SLEEP
- CAN_MODE_DEEP_SLEEP
- CAN_MODE_SILENT

Please note that also built in function blocks and functions are considered reserved keywords, but they are not listed here.

Anatomy of TSharkRex™ code

The code below is a typical TSharkRex™ code. It is divided in sections for easy reference and described after each section.

```
VAR
	TIMER1 : TON;
	COUNTER : BYTE;
END_VAR;
```

Declaration of variables can be placed anywhere in the code but it will only be executed once. *TIMER1* is an instance of a built in function block called *TON.*

COUNTER is declared as a *BYTE* (*COUNTER* can be anything between 0 and 255).

```
TIMER1(IN := TRUE, PT := T#200ms);

IF TIMER1.Q THEN
	COUNTER := COUNTER + 1;
	IF COUNTER > 10 THEN
		TIMER1(IN := FALSE);
	END_IF;
END_IF;
```

TIMER1 is simply a timer that is started when *IN* is set to *TRUE,* and the *Q* flag will be *TRUE* when *PT* time has been fulfilled. Above code is activating the timer and after 200 milliseconds it will increment *COUNTER* to 11, after that it will reset the timer and every 200 milliseconds after it will increment *COUNTER* by one and reset the timer.

Comments

While writing your own code comments are useful to make notes and sometimes exclude code snippets that you for some reason don't want in your code but really don't want to remove completely.

Comments in *VAR_SIGNAL* will show up in the Online TSharkRex™ Platform for better understanding when you using libraries. All comments will be shown in green text and *Italic* style on the Online TSharkRex™ Platform.

Use '//' for single line comments and '(* *)' for multi line comments.

Constants

Constants can be in form of integer, BOOL, BYTE, INT and DINT. It is possible to specify the value in decimal, binary or hexadecimal notation.

```
VAR
        MYCONST : BYTE;
END_VAR;

MYCONST.0 := TRUE; // First bit in byte set to 1.
MYCONST := 10; // Set to decimal 10.
MYCONST := 0x10; // Set to hexadecimal 0x10.
```

VAR

The *VAR* statement is used to initiate the section where variables can be declared. It's allowed to have more than one VAR section within the same scope.

VAR statement must always be closed with *END_VAR;*

VAR_OUTPUT

The *VAR_OUTPUT* statement is used to initiate the section where output variables can be declared. In the Online TSharkRex™ Platform you can only have **one** *VAR_OUTPUT* section within the same scope. If you add more than one *VAR_OUTPUT* the Online TSharkRex™ Platform will only display the first section found in the source code.

Therefore declared Output variables will not be shown in the XBB Configurator App for instance. However the compiler will treat *VAR_OUTPUT* same as *VAR* and *VAR_SIGNAL*.

VAR_SIGNAL

The *VAR_SIGNAL* statement is used to initiate the section where signal variables for libraries can be declared. In the Online TSharkRex™ Platform you can only have one *VAR_SIGNAL* section within the same scope. If you add more than one *VAR_SIGNAL* the Online TSharkRex™ Platform will only display the first section found in the source code.

END_VAR

The *END_VAR* statement ends a section that was initiated with *VAR*, *VAR_OUTPUT* or *VAR_SIGNAL* statements.

Example of various *VAR* statements:

```
VAR
	MY_BYTE : BYTE;
END_VAR;

VAR_OUTPUT
	MY_OUTPUT : OUTPUT;
END_VAR;

VAR_SIGNAL
	MY_SIGNAL : BOOL;
END_VAR;

IF MY_BYTE = MY_BYTE2 THEN
	MY_SIGNAL := TRUE;
END_IF;

MY_OUTPUT(VALUE := MY_SIGNAL);

VAR
	MY_BYTE2 : BYTE;
END_VAR;
```

Above example has two VAR statements where the second one is at the end of the code which does not matter for the compiler. *MY_BYTE2* will be declared in the same way as the first *VAR* statement.

Code Flow Control

Following pages describe the various elements of the TSharkRex™ Programming Language that control the flow of a program. The elements are divided into two groups, conditional execution and iterative execution.

Conditional execution:

- *IF, THEN, ELSE, END_IF*
- *CASE OF, END_CASE*

Iterative execution:

- *WHILE, END_WHILE*
- *FOR, END_FOR*

IF, THEN, ELSE, ELSIF, END_IF statement

IF Statements are used for conditional execution of code in TSharkRex™.

```
IF <expression> THEN
        statement;
ELSIF <expression> THEN
        statement;
ELSE
        statement;
END_IF;
```

The <expression> is an expression that evaluates to a *BOOL*, and therefore it is always *TRUE* or *FALSE*, the statement after *THEN* is executed if the <expression> is *TRUE*. To evaluate an *INT* or *BYTE* you can only use single bit with the punctuation syntax or use operators to get a true/false state.

Example:

```
VAR
        MYBYTE : BYTE;
        MYBOOL : BOOL;
END_VAR;

IF (MYBYTE.0) OR (MYBYTE > 0) THEN
        MYBOOL := FALSE;
END_IF;
```

All operators can be used in the IF statements
Example:

```
VAR
        BYTE1   : BYTE;
        BYTE2   : BYTE;
        MYBOOL : BOOL;
END_VAR;


BYTE1 := 2;
BYTE2 := 10;

IF ((((BYTE1 * 5) = BYTE2) + 5) = 5) OR MYBOOL = TRUE THEN
        MYBOOL := FALSE;
END_IF;
```

CASE OF, ELSE, END_CASE statement

CASE Statements are used for conditional execution of code in TSharkRex™.

Syntax:

```
CASE <variable> OF
<number 1>:
        statement;
<number 2>:
        statement;
<number 7>:
        statement;
ELSE
        statement;
END_CASE;
```

<variable> evaluates to a number. If one of the <number..> values has the same value as <variable> the statement will be executed. If none of the <number..> value matches the <variable> value the statement after the *ELSE* statement will be executed.

If there are no *ELSE* statement, the code after *END_CASE* will be executed. The <variable> can only be BOOL, *INT, BYTE* or *DINT.*

Example:

```
VAR
        COUNTER : BYTE;
        MYBOOL : BOOL;
END_VAR;

CASE COUNTER OF
        1: MYBOOL := FALSE;
        2: MYBOOL := TRUE;
ELSE
        MYBOOL := NOT MYBOOL;
END_CASE;
```

WHILE, END_WHILE

WHILE statements are used for repetitive conditional execution of code in TSharkRex™.

Syntax:

```
WHILE <expression> DO
       statement;
END_WHILE;
```

The <expression> is an expression that evaluates to a *BOOL*, and therefore it is always *TRUE* or *FALSE*, the statement after *DO* is executed **if** the <expression> is *TRUE*.

To evaluate an *INT* or *BYTE* you can only use single bit with the punctuation syntax or use operators to get an *TRUE*/*FALSE* state.

The *WHILE* statement will repeat until <expression> is *FALSE*.

Example:

```
VAR
       COUNTER : BYTE;
END_VAR;

WHILE COUNTER < 10 DO
       COUNTER := COUNTER + 1;
END_WHILE;
```

FOR, END_FOR

FOR statements are used for repetitive conditional execution of code in TSharkRex™.

Syntax:

```
FOR <variable> := <start_value> TO <end_value> DO
        statement;
END_FOR;
```

The *statement;* are executed as long as the counter <variable> is not greater than the <end_value>. This is checked before executing the *statement;*. As clarification, *statement;* will not be executed if <variable> is greater than <end_value>.

When *statement;* are executed the <variable> is increased by 1.

Example:

```
VAR
        COUNTER : BYTE;
        MYARRAY : ARRAY[0..7] OF BYTE;
END_VAR;

FOR COUNTER := 0 TO 7 DO
        MYARRAY[COUNTER] := 0xFF;
END_FOR;
```

Data types

Different variables and data types can be used in TSharkRex™. Variables are used as placeholders for values. A variable can hold data of different sizes and types. TSharkRex™ only supports integer data types.

Data types and variables

BOOL, BYTE, INT and DINT are supported in TSharkRex™

Type	Lower limit	Upper limit	Memory space
BOOL	0	1	8 Bit
BYTE	0	255	8 Bit
INT	-32768	32767	16 Bit
DINT	-2147483648	2147483647	32 Bit

As a result when larger types are converted to smaller types, information may be lost.

BOOL

BOOL type variables may be given the values *TRUE* or *FALSE*. 8 bits of memory space will be reserved.

BYTE

BYTE type variables may be given the values 0 to 255. 8 bits of memory space will be reserved.

INT

INT type variables may be given the values -32768 to 32767. 16 bits of memory space will be reserved.

DINT

DINT type variables may be given the values -2147483648 to 2147483647. 32 bits of memory space will be reserved.

ARRAY

An Array is used to group different data types. TSharkRex™ supports 1 dimensional arrays. The actual number and size of an array is only limited by the available memory (with an upper theoretical border of 145 232 534 555 328 511 arrays).

Syntax:

```
<name_of_array> : ARRAY[<lower_limit>..<upper_limit>] OF <data_type>
Where <lower_limit> is 0, <upper_limit> is theoretical 145232534555328511 and <data_type> is BOOL,
BYTE, INT or DINT.
```

Example:

```
VAR
       MYDATA1 : ARRAY[0..5] OF BOOL;
       MYDATA2 : ARRAY[3..6] OF DINT;
       MYDATA3 : ARRAY[0..7] OF BYTE;
END_VAR;

MYDATA1[4] := FALSE;
MYDATA2[3] := 123456789;
MYDATA3[0] := 0xFF;
```

Accessing an array component is used by the following syntax:

```
<name_of_array>[<index>]
```

Please see above Example code.

Predefined constants

There are a number of constants defined in TSharkRex™

TRUE

TRUE is used to define variables of the type BOOL and it value are 1.

FALSE

FALSE is used to define variables of the type BOOL and it value are 0.

CAN_MODE_CONFIG

CAN_MODE_CONFIG is used to define which state function block CAN_MODE are in. It evaluates to 0.

CAN_MODE_NORMAL

CAN_MODE_NORMAL is used to define which state function block CAN_MODE are in. It evaluates to 1.

CAN_MODE_SLEEP

CAN_MODE_SLEEP is used to define which state function block CAN_MODE are in. It evaluates to 2.

CAN_MODE_DEEP_SLEEP

CAN_MODE_DEEP_SLEEP is used to define which state function block CAN_MODE are in. It evaluates to 3.

CAN_MODE_SILENT

CAN_MODE_SILENT is used to define which state function block CAN_MODE are in. It evaluates to 4.

Function and Function blocks

Currently TSharkRex™ only supports built-in function and function blocks. Listed below is all currently built-in function and function blocks that is supported.

- CAN_RX
- CAN_TX
- CAN_MODE
- CAN_FILTER
- CAN_MASK
- TON
- TOF
- R_TRIG
- F_TRIG
- OUTPUT
- DEBUG
- HARDWARE

CAN_RX (ID, EXT, ENABLE, DATA, DATALENGTH, AVAILABLE)

CAN_RX is a function block that receives a CAN message (8 bytes long) on specified *CAN ID*. *CAN_RX* supports standard 11-bit identifier and also extended 29-bit identifiers.

As standard, if not CAN_MODE function block has been initialed transmission speed will be 500 kbit/s.

Function block inputs

CAN_RX have 3 inputs *ID, EXT,* and *ENABLE*.

- ***ENABLE***
  To initiate an instance of *CAN_RX,* the function block must be executed in the code. When ***ENABLE*** flag is set to *TRUE*, the hardware CAN message queue will update and receive messages that corespondents to selected *ID*.

  If ***ENABLE*** is set to *FALSE* the function block will still be executed but no messages will arrive in the hardware CAN message queue.

- ***EXT***
  To read extended 29-bit Identifiers the ***EXT*** flag must be set to *TRUE*, if ***EXT*** flag is set to *FALSE* 29-bit identifiers will be ignored.

- ***ID***
  To set up the CAN hardware message queue correctly user must specify which ***ID*** to read. 11-bit identifiers allows a total of 2^11 different messages. A 29 bit identifier allows a total of 2^29 different messages.

Function block outputs

CAN_RX have 3 outputs *DATA, DATALENGTH and AVAILABLE.*

- ***DATA***
  When the CAN hardware message queue receives a message that match the specified *ID* it is stored in ***DATA*** and available to use in the code.

- ***DATALENGTH***
  When the CAN hardware message queue receives a message that match the specified *ID* it will store the length of the data array in ***DATALENGTH***.

- ***AVAILABLE***
  When one or more messages are received in the CAN hardware message queue the output variable ***AVAILABLE*** will represent the number of messages in the queue.

Examples of CAN_RX:

```
VAR
	READCAN : CAN_RX;
	SENDCAN : CAN_TX;
	MYDATA : ARRAY[0..7] OF BYTE;
END_VAR;

READCAN(ENABLE := TRUE, EXT := FALSE, ID := 0x100, DATA := MYDATA);

IF (READCAN.AVAILABLE > 0) THEN
	SENDCAN(ID := 0x101, DATA := MYDATA, DATALENGTH := READCAN.DATALENGTH);
END_IF;
```

Above example will wait for a message on *ID 0x100*, copy the data to array *MYDATA,* send it out on CAN again with *ID 0x101,* with the *DATALENGTH* of recently received message on *0x100*.

CAN_TX(EXT, ID, DATA, DATALENGTH)

CAN_TX is a built in function block that sends a CAN message (maximum 8 bytes long) on specified CAN *ID*. *CAN_TX* supports standard 11-bit identifier and also extended 29-bit identifiers.

As standard, if not CAN_MODE function block has been initialed transmission speed will be 500 kbit/s.

Function block inputs

CAN_TX have 2 inputs *EXT* and *ID*.

- ***EXT***
  To send extended 29-bit Identifiers the ***EXT*** flag must be set to *TRUE*, if ***EXT*** flag is set to *FALSE* only 11-bit identifiers can be sent out.

- ***ID***
  To send out a message on specific ***ID***. 11-bit identifiers allows a total of 2^11 different message IDs. A 29 bit identifier allows a total of 2^29 different message IDs.

Function block outputs

CAN_TX have 2 outputs *DATA* and *DATALENGTH.*

- ***DATA***
  Maximum an 8 byte long *ARRAY* of *BYTE* that sends out on CAN.

- ***DATALENGTH***
  Length of the CAN message that will be sent out on CAN.

Examples of *CAN_TX:*

```
VAR
        SENDCAN : CAN_TX;
        MYDATA : ARRAY[0..7] OF BYTE;
END_VAR;

SENDCAN(ID := 0x100, DATA := MYDATA, DATALENGTH := 8);
```

Above example will send out a CAN message, 8 bytes long every cycle with the data contained in *MYDATA*.

CAN_MODE (MODE, BAUDRATE)

CAN_MODE is a built in function block where the user can change different settings and baud rate for the hardware CAN controller. 2 inputs are available, *MODE* and *BAUDRATE.*

4 different modes flags can be selected.

- ***CAN_MODE_CONFIG***
  Must be selected if user wants to change *BAUDRATE*, *CAN_FILTER* and *CAN_MASK* after *BAUDRATE*, *CAN_FILTER* or *CAN_MASK* has been changed/set user must initialize *CAN_MODE* with flag *CAN_MODE_NORMAL* again.
- ***CAN_MODE_NORMAL***
  Normal mode where CAN can send and receive data on the CAN network.
- ***CAN_MODE_SLEEP***
  The hardware CAN controller has an internal sleep mode that is used to minimize the current consumption of the device. The hardware CAN controller interface remains active for reading even when the hardware CAN controller is in sleep mode.

  When in sleep mode, the wake-up interrupt is still active. When in sleep mode, the hardware CAN controller stops its internal oscillator. The hardware CAN controller will wake-up when bus activity occurs. The transmit will remain in the recessive state while the hardware CAN controller is in sleep mode.
- ***CAN_MODE_SILENT***
  Provides a means for the hardware CAN controller to receive all messages (including messages with errors).

  This mode can be used for bus monitor applications. *CAN_MODE_SILENT* mode is a silent mode, meaning no messages will be transmitted while in this mode (including error flags or Acknowledge signals).

  In *CAN_MODE_SILENT*, both valid and invalid messages will be received, regardless of filters and masks.
- ***CAN_MODE_DEEP_SLEEP***
  This flag turns off the hardware CAN transceiver and no CAN messages can be sent or received. This flag is to reduce power consumption.

See examples of *CAN_MODE* on next page:

Example of *CAN_MODE (CAN_MODE_CONFIG)*:

```
VAR
	SETCAN		: CAN_MODE;
	SETFILTER	: CAN_FILTER;
	GETCAN		: CAN_RX;

	MYDATA		: ARRAY[0..7] OF BYTE;
	FIRSTCYCLE	: BOOL;
END_VAR;

IF NOT FIRSTCYCLE THEN
	SETCAN(MODE := CAN_MODE_CONFIG);
	SETFILTER(SLOT := 0, ID := 0x100);
	SETCAN(MODE := CAN_MODE_NORMAL);
	FIRSTCYCLE := FALSE;
END_IF;

GETCAN(ENABLE := TRUE, ID := 0x100, EXT := FALSE, DATA := MYDATA);
```

Above example shows how *CAN_MODE* is used to set the hardware CAN controller to configuration mode, after that we activate the first *CAN_FILTER* to 0x100. The hardware CAN controller will only allow messages with identifier 0x100 to pass.

After that we set *CAN_MODE* to *CAN_MODE_NORMAL* again.

Example of *CAN_MODE (CAN_MODE_SILENT)*:

```
VAR
	SETCAN		: CAN_MODE;
	GETCAN		: CAN_RX;

	MYDATA		: ARRAY[0..7] OF BYTE;
	FIRSTCYCLE	: BOOL;
END_VAR;

IF NOT FIRSTCYCLE THEN
	SETCAN(MODE := CAN_MODE_SILENT);
	FIRSTCYCLE := FALSE;
END_IF;

GETCAN(ENABLE := TRUE, ID := 0x100, EXT := FALSE, DATA := MYDATA);
```

Above example shows how the hardware CAN controller is set to Silent mode.

CAN_FILTER (SLOT, ID, EXT)

CAN_FILTER is a built in function block. The hardware CAN controller can utilize up to 6 different filters simultaneously. Each filter (0-5) is selected with function block input parameter *SLOT,* function block input *ID* sets the desired identifier which can be both 11-bit or 29 bit depending on function block input *EXT* (*FALSE* = 11-bit, *TRUE* = 29-bit).

Filter & Mask Truth table

Mask Bit *n*	Filter Bit *n*	Message Identifier Bit	Accept or Reject Bit *n*
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Note: x = don't care.

Note!

The hardware CAN Controller must be in *CAN_MODE_CONFIG* before *CAN_FILTER* can be used. Please see example below.

Example of CAN_FILTER:

```
VAR
	SETCAN		: CAN_MODE;
	SETFILTER	: CAN_FILTER;
	SETMASK		: CAN_MASK;
	GETCAN		: CAN_RX;

	MYDATA		: ARRAY[0..7] OF BYTE;
	FIRSTCYCLE	: BOOL;
END_VAR;

IF NOT FIRSTCYCLE THEN
	SETCAN(MODE := CAN_MODE_CONFIG);
	SETMASK(SLOT := 0, ID := 0x07FF);
	SETFILTER(SLOT := 0, ID := 0x100); //Only this message will be received
	SETFILTER(SLOT := 1, ID := 0x110); //Only this message will be received
	SETCAN(MODE := CAN_MODE_NORMAL);
	FIRSTCYCLE := FALSE;
END_IF;

GETCAN(ENABLE := TRUE, ID := 0x100, EXT := FALSE, DATA := MYDATA);
```

CAN_MASK(SLOT, ID, EXT)

CAN_MASK is a built in function block. The hardware CAN controller can utilize up to 2 different mask simultaneously. Each mask (0-1) is selected with function block input parameter *SLOT,* function block input *ID* sets the desired identifier mask which can be both 11-bit or 29 bit depending on function block input *EXT* (*FALSE* = 11-bit, *TRUE* = 29-bit).

Filter & Mask Truth table

Mask Bit *n*	Filter Bit *n*	Message Identifier Bit	Accept or Reject Bit *n*
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Note: x = don't care.

Note!

The hardware CAN Controller must be in *CAN_MODE_CONFIG* before *CAN_MASK* can be used. Please see example below.

Example of CAN_MASK:

```
VAR
	SETCAN		: CAN_MODE;
	SETFILTER	: CAN_FILTER;
	SETMASK		: CAN_MASK;
	GETCAN		: CAN_RX;

	MYDATA		: ARRAY[0..7] OF BYTE;
	FIRSTCYCLE	: BOOL;
END_VAR;

IF NOT FIRSTCYCLE THEN
	SETCAN(MODE := CAN_MODE_CONFIG);
	SETMASK(SLOT := 0, ID := 0x07FF);
	SETFILTER(SLOT := 0, ID := 0x100); //Only this message will be received
	SETFILTER(SLOT := 1, ID := 0x110); //Only this message will be received
	SETCAN(MODE := CAN_MODE_NORMAL);
	FIRSTCYCLE := FALSE;
END_IF;

GETCAN(ENABLE := TRUE, ID := 0x100, EXT := FALSE, DATA := MYDATA);
```

TON (IN, PT, Q)

The function block TON (Timer on Delay) implements a turn-on delay. *IN*, *PT* and *Q* are input variables of data types *BOOL*, *TIME* and *BOOL* respectively.

If *IN* = *FALSE*, *Q* is *FALSE*. As soon as *IN* becomes *TRUE*, the timer will begin to count and when the internal timer is equal to *PT, Q* will be *TRUE.*

The *PT TIME* data type can be written in different forms see examples below:

```
VAR
        TIMER1 : TON;
        TIMER2 : TON;
        TIMER3 : TON;
        TIMER4 : TON;
        TIMER5 : TON;
        TIMER6 : TON;
END_VAR;

TIMER1(IN := TRUE, PT := T#10ms);     //10 milliseconds
TIMER2(IN := TRUE, PT := T#10s);      //10 seconds
TIMER3(IN := TRUE, PT := T#10m);      //10 minutes
TIMER4(IN := TRUE, PT := T#10h);      //10 hours
TIMER5(IN := TRUE, PT := T#10d);      //10 days
TIMER6(IN := TRUE, PT := T#10d10h10m10s10ms); //combined values, 10 days, 10 hours, 10 minutes...
```

TOF (IN, PT, Q)

The function block TOF implements a turn-off delay. *IN*, *PT* and *Q* are input variables of data types *BOOL*, *TIME* and *BOOL* respectively.

Q = *FALSE* when *IN* = *FALSE* and when the internal timer is equal to *PT*, otherwise *Q* = *TRUE*. The internal timer will start counting when *IN* = *FALSE*.

The *PT TIME* data type can be written in different forms see examples under *TON*:

```
VAR
        TIMER1 : TOF;
END_VAR;

TIMER1(IN := FALSE, PT := T#10s);     // after 10 seconds TIMER1.Q will be FALSE.
```

R_TRIG (CLK, Q)

Function block *R_TRIG* detects a rising edge. The output *Q* will remain *FALSE* as long as the input variable *CLK* is *FALSE*.

As soon as *CLK* returns *TRUE*, *Q* will first return *TRUE* and the next program cycle *Q* will return *FALSE* until *CLK* has falling edge followed by a rising edge again.

Example:

```
VAR
        SIGNAL  : R_TRIG;
        INPUT   : BOOL;
        COUNTER : BYTE;
END_VAR;

SIGNAL(CLK := INPUT);                   //Waiting for a rising edge on INPUT

IF SIGNAL.Q THEN                        //INPUT has been set to TRUE
        COUNTER := COUNTER + 1;         // COUNTER will increment by 1
END_IF;
```

Above example will only increment *COUNTER* every time the *INPUT* has changed it state from *TRUE-FALSE-TRUE*.

When *INPUT* is *TRUE* the *IF* statement will only be executed once. Next program cycle will turn *SIGNAL.Q* to *FALSE* and will remain false until *CLK* has been *FALSE* and *TRUE* again.

F_TRIG (CLK, Q)

Function block F_*TRIG* detects a falling edge. The output *Q* will remain *FALSE* as long as the input variable *CLK* is *TRUE*.

As soon as *CLK* returns *FALSE*, *Q* will first return *TRUE*, then the next program cycle *Q* will return *FALSE* until *CLK* has a rising followed by a falling edge.

Example:

```
VAR
        SIGNAL  : F_TRIG;
        INPUT   : BOOL;
        COUNTER : BYTE;
END_VAR;

SIGNAL(CLK := INPUT);                   //Waiting for a falling edge on INPUT

IF SIGNAL.Q THEN                        //INPUT has a falling edge
        COUNTER := COUNTER + 1;         // COUNTER will increment by 1
END_IF;
```

Above example will only increment *COUNTER* every time the *INPUT* has changed it state from *FALSE-TRUE-FALSE*. When *INPUT* is *FALSE* the *IF* statement will only be executed once.

Next program cycle will turn *SIGNAL.Q* to *FALSE* and will remain false until *CLK* has been *TRUE* and *FALSE* again.

OUTPUT (VALUE)

Function block *OUTPUT* is a hardware specific function block where *VALUE* is a *BOOL* and activates board specific outputs on the XBB Dongle™ and XBB PowerUnit™ hardware.

When using the Online TSharkRex™ Platform the declared name of the function block will be showed and user can select it to drive specific outputs on the XBB PowerUnit™ hardware when declared in the *VAR_OUTPUT* declaration.

See example:

```
VAR_OUTPUT
        HIGHBEAM        : OUTPUT;
        LOWBEAM         : OUTPUT;
END_VAR;

HIGHBEAM(VALUE := CAN_SIGNAL_HIGHBEAM);
LOWBEAM(VALUE := CAN_SIGNAL_LOWBEAM);
```

DEBUG (ID, DATA)

Function block *DEBUG* is used for debugging purpose over CAN, where *ID* identifies the *DATA* sent over CAN. *DEBUG* will send out *DATA* over the *CAN* with the *ID* packed in the CAN data.

DEBUG function block will listen for a message on identifier *0xFFFFFFFF* when a message with identifier *0xFFFFFFFF* occurs *DEBUG* will send out the *ID* and *DATA* on identifier *0x0FFFFFF* with the following CAN transmission message for 1 second.

ID	ID-name	Cycle time in ms	Launch type	Signal byte no.	Signal name	Signal length (bit)	Normalization	Value range
0x0FFFFFF	Debug msg	10	Cyclic for 1000 ms.	0	Debug ID	8	0x0-0xFF	0x0-0xFF
0x0FFFFFF	Debug msg	10	Cyclic for 1000 ms	1-7	Debug DATA	56	0x0-0xFF FFFF FFFF FFFF	0x0-0xFF FFFF FFFF FFFF

As above table, byte 0 contains the ID number and byte 1-7 DATA is sent in.

Example code:

```
VAR
        DMSG : DEBUG;
        VAR1 : DINT;
        VAR2: BYTE;
END_VAR;

DMSG(ID := 0, VALUE := VAR1);  //Sending out VAR1 value on CAN
DMSG(ID := 1, VALUE := VAR2);  //Sending out VAR2 value on CAN
```

HARDWARE(SUPPLY_VOLTAGE, GYRO_X, GYRO_Y, GYRO_Z)

Function block *HARDWARE* is a hardware specific function block for XBB Dongle™ hardware. The function block *HARDWARE* returns the supply voltage and the built in gyro values for X, Y and Z direction.

It also have *GYRO_X_HIGHPASS, GYRO_Y_HIGHPASS and GYRO_Z_HIGHPASS* as outputs for high pass filtrated values. Function block *HARDWARE* does not require any input values and every time is executes it updates the values.

SUPPLY_VOLTAGE is returned in mV and the *GYRO X,Y,Z* outputs is in mG.

Example:

```
VAR
        HW : HARDWARE;
        SYSTEM_ON : BOOL;
END_VAR;

IF HW.SUPPLY_VOLTAGE > 12000 THEN
        SYSTEM_ON := TRUE;
END_IF;
```

Operators

AND

The logical operator AND operates on expressions with BOOL, BYTE, INT and DINT data types. When operating on data types other than BOOL the operation is bit wise.

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

NOT

The logical operator NOT operates on expressions with BOOL, BYTE, INT and DINT data types. When operating on data types other than BOOL the operation is bit wise (all ‘1’ will be ‘0’, and all ‘0’ will be ‘1’).

Input	Output
1	0
0	1

OR

The logical operator OR operates on expression with BOOL, BYTE, INT and DINT data types. When operating on data types other than BOOL the operation is bit wise.

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Operator: ’ / ’

The ‘/’ operator divides two integer numbers.

Example:

```
MYBYTE := 200 / 4;
MYDINT := MYBYTE / 2;
```

Operator: ’ + ’

The ‘+’ operator adds two integer numbers.

Example:

```
MYBYTE := 100 + 10;
MYDINT := MYBYTE + 10;
```

Operator: ’ - ’

The ‘-’ operator subtracts two integer numbers.

Example:

```
MYBYTE := 100 - 10;
MYDINT := MYBYTE - 10;
```

Operator: ’ * ’

The ‘*’ operator multiplies two integer numbers.

Example:

```
MYBYTE := 2 * 10;
MYDINT := MYBYTE * 10;
```

Operator: ’ = ’

The ‘=’ operator compares two numbers or data types and returns TRUE if they are equal.

Example:

```
IF MYBYTE = MYDINT THEN // if MYBYTE and MYDINT has the same value the IF condition is met.
        ...
END_IF;

MYBOOL := MYBYTE = 10; //if MYBYTE is 10 then MYBOOL will be TRUE
```

Operator: ’ < > ’

The ‘< >’ operator compares two numbers or data types and returns TRUE if they are not equal.

Example:

```
IF MYBYTE <> MYDINT THEN // if MYBYTE and MYDINT has different values the IF condition is met.
        ...
END_IF;

MYBOOL := MYBYTE <> 10; //if MYBYTE is 10 then MYBOOL will be FALSE
```

Operator: ’ < ’ and ’ > ’

The ‘<’ operator compares two numbers or data types and returns TRUE if the left expression is smaller than the right one. The ‘>’ operator compares two numbers or data types and returns TRUE if the left expression is greater than the right one.

Example:

```
IF MYBYTE < MYDINT THEN // if MYBYTE is smaller than MYDINT the IF condition is met.
        ...
END_IF;

MYBOOL := MYBYTE > 10; //if MYBYTE is greater than 10 then MYBOOL will be TRUE.
```

Operator: ’ <= ’ and ’ >= ’

The ‘<=’ operator compares two numbers or data types and returns TRUE if the left expression is smaller or equal than the right one. The ‘>=’ operator compares two numbers or data types and returns TRUE if the left expression is greater or equal than the right one.

Example:

```
IF MYBYTE <= MYDINT THEN // if MYBYTE is smaller or equal to MYDINT the IF condition is met.
        ...
END_IF;

MYBOOL := MYBYTE >= 10; //if MYBYTE is greater or equal to 10 then MYBOOL will be TRUE.
```

Parenthesis ’ () ’

The brackets ‘()’ are used in expressions to force a specific order of execution.

Example:

```
MYBYTE := 10 + 10 * 5; // MYBYTE will be 60
MYBYTE := (10 + 10) * 5; // MYBYTE will be 100
```

Code examples

Volkswagen wake-up and sleep routine.

```
VAR
	CAN_RX_STATUS			: CAN_RX;
	CAN_RX_STATUS_DATA		: ARRAY[0..7] OF BYTE;
	TIMER_FORDROJ_AVSTANGNING	: TON;
	TIMER_FORDROJ_NOLLSTALL		: TON;
	TIMER_RESTART			: TON;
	SYSTEM_INIT			: BOOL;
END_VAR;

VAR_SIGNAL
	SIGNAL_TANDNING : BOOL;
END_VAR;

CAN_RX_STATUS(ENABLE := TRUE, ID := 0x17F00010, EXT := TRUE, DATA := CAN_RX_STATUS_DATA);
IF CAN_RX_STATUS.AVAILABLE THEN
	TIMER_FORDROJ_NOLLSTALL(IN := FALSE);
	TIMER_FORDROJ_AVSTANGNING(IN := NOT SIGNAL_TANDNING, PT := T#120s);
	IF NOT TIMER_FORDROJ_AVSTANGNING.Q THEN
		SYSTEM_INIT := TRUE;
		TIMER_RESTART(IN := FALSE);
	END_IF;

	IF TIMER_FORDROJ_AVSTANGNING.Q THEN
		SYSTEM_INIT := FALSE;
		TIMER_RESTART(IN := TRUE, PT:=T#12s);
		TIMER_FORDROJ_AVSTANGNING(IN := NOT TIMER_RESTART.Q);
	END_IF;
END_IF;

IF NOT CAN_RX_STATUS.AVAILABLE THEN
	TIMER_FORDROJ_NOLLSTALL(IN := TRUE, PT := T#1s);
	IF TIMER_FORDROJ_NOLLSTALL.Q THEN
		SYSTEM_INIT := FALSE;
		TIMER_FORDROJ_AVSTANGNING(IN := FALSE);
	END_IF;
END_IF;
```

Above code is a wake-up and sleep routine for VAG vehicles where there is not specific ignition signal on the CAN network. What the code does is to check 0x17F00010 message ID and when ignition turns off if waits 120 seconds before it stops sending out message and wait 12 seconds to see if the vehicle stops sending out messages.

Library & recipe for Opel Vectra 2007

```
VAR
	DATA1		: ARRAY[0..7] OF BYTE;
	CANRECV1	: CAN_RX;

	CAN_TIMER1	: TON;
	HELLJUS_R_TRIG : R_TRIG;
END_VAR

VAR_SIGNAL
	SIGNAL_HELLJUS		: BOOL;
	SIGNAL_HALVLJUS		: BOOL;
END_VAR;

CANRECV1(ENABLE := TRUE, ID := 0x381, EXT := FALSE, DATA := DATA1);

HELLJUS_R_TRIG(CLK := DATA1[1].3 AND DATA1[0].4);

IF (HELLJUS_R_TRIG.Q) THEN
	SIGNAL_HELLJUS := NOT SIGNAL_HELLJUS;
END_IF;

IF (NOT SIGNAL_HALVLJUS) THEN
	SIGNAL_HELLJUS := FALSE;
END_IF;

SIGNAL_HALVLJUS := DATA1[0].4;
CAN_TIMER1(IN := NOT CANRECV1.AVAILABLE , PT := T#500ms);

IF CAN_TIMER1.Q THEN
	SIGNAL_HELLJUS := FALSE;
	SIGNAL_HALVLJUS := FALSE;
END_IF;
```

Above library is used in below recipe. CAN message over OBD-II is read and signal for highlight lever is read out. When low beam is activated then a *R_TRIG* will shift high beam output. When low beam is turned off (ignition off or light switcher is set to off) the high beam output will be off.

```
VAR_OUTPUT
	HIGHBEAM	: OUTPUT;
	LOWBEAM		: OUTPUT;
END_VAR;

HIGHBEAM(VALUE := SIGNAL_HELLJUS);
LOWBEAM(VALUE := SIGNAL_HALVLJUS);
```

Traceability

Reference document

Denomination	Publication number
Reference instruction	Nr:EN-20200925-1

Revision

The following significant changes have taken place since the previous version:

Rev	Page	Description of revision	Approved by tech. manager	Date	App. by doc. officer.	Date
1	ALL	Creation of doc.	KHS	20-09-25	KHS	20-09-25
2						
3						
4						
5						