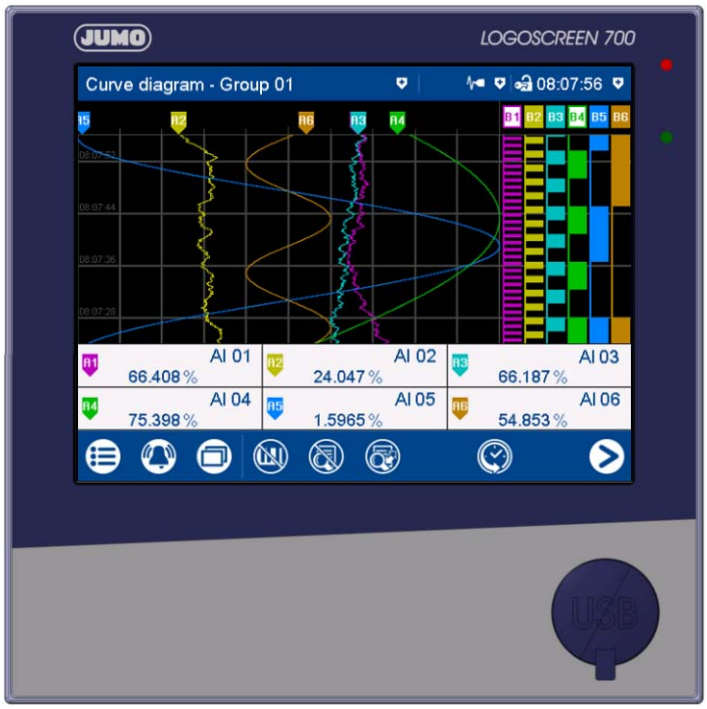
JUMO LOGOSCREEN 601/700

Paperless recorder



ST Editor Manual

70653000T96Z001K000

V1.00/EN/00710350

Contents

Contents

1.1 Safety information

Note symbols

NOTE!

This symbol refers to **important information** about the product, its handling, or additional benefits.

REFERENCE!

This symbol refers to **additional information** in other sections, chapters, or other manuals.

1.2 Intended use

The device is designed for use in an industrial environment as specified in the technical data. Other uses beyond those defined are not viewed as intended uses.

The device has been manufactured in compliance with applicable standards and directives as well as the applicable safety regulations. Nevertheless, improper use may lead to personal injury or material damage.

To avoid danger, only use the device:

- For the intended use
- When in good order and condition
- When taking the technical documentation provided into account

Risks resulting from the application may arise, e.g. as the result of missing safety provisions or wrong settings, even when the device is used properly and as intended.

1.3 Qualification of personnel

This document contains the necessary information for the intended use of the device to which it relates.

It is intended for staff with technical qualifications who have been specially trained and have the appropriate knowledge in the field of automation technology.

The appropriate level of knowledge and the technically fault-free implementation of the safety information and warnings contained in the technical documentation provided are prerequisites for risk-free mounting, installation, and startup as well as for ensuring safety when operating the described modules. Only qualified personnel have the required specialist knowledge to correctly interpret and implement the safety information and warnings contained in this document in specific situations.

1.4 Content of this document

NOTE!

This document applies to paperless recorders of types 706521 and 706530.

This document describes the application of the ST editor with which users can create their own applications in the PLC programming language "structured text" (ST) for the device. The document is intended for users with relevant programming knowledge.

The PLC programming language "Structured Text" (ST) is described in the standard DIN IEC 61131-3. Detailed information about programming can be found in this standard. The ST module of the respective device supports only a subset of the programming language described in the standard.

In addition to this document, the operating manual of the respective device must be observed:

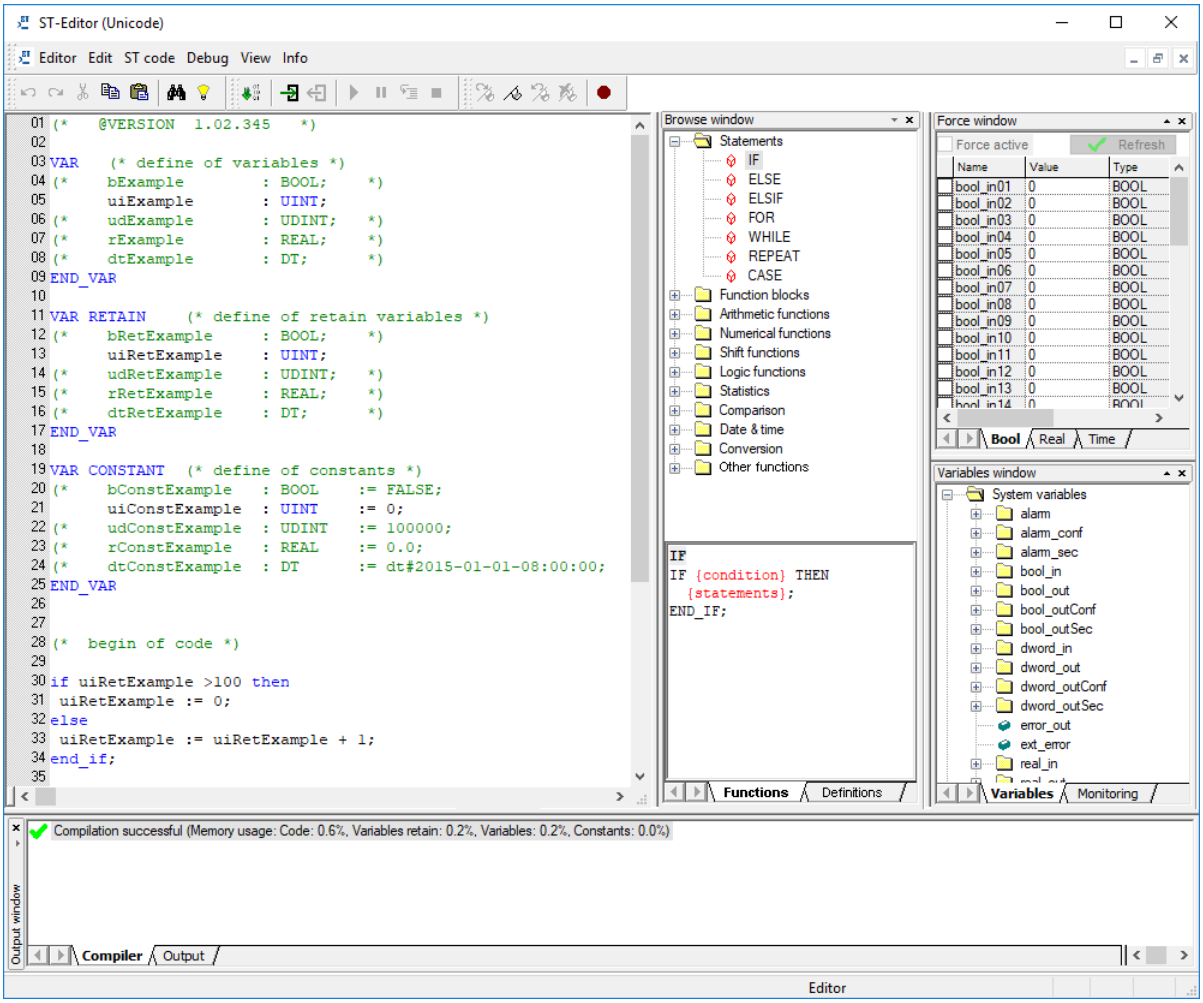
- Type 706521:
  Document 70652100T90Z...K...
- Type 706530:
  Document 70653000T90Z...K...

1 Introduction

The ST editor is part of the setup program and is started by clicking on the corresponding button in the "ST code" window (see operating manual).

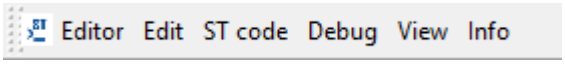
The finished application is transmitted to the device as ST code and continuously processed in the integrated ST module.

Overview



The user interface consists of several toolbars and windows, which are briefly described in the following sections.

Menu bar



The individual menus contain functions for editing and compiling a program, for troubleshooting, for displaying and hiding toolbars and windows of the user interface, as well as ST editor version information.

If "display guard" function is activated (in the "Editor" menu), the password assigned by the user is queried before the ST editor is restarted. If the password is not known, the possibility exists, to start the ST editor with the factory settings (code example). Previously created source code will be deleted!

Further information about the functions of the menu bar is displayed in the status bar when the mouse pointer hovers over the individual function in the respective menu.

Toolbar

Some functions of the menu bar are also available in the toolbar and can be selected by a simple mouse click. The meaning of the symbols is briefly described by a tool tip function (hover over the respective symbol with the mouse pointer). In addition, further information about the relevant function is then also displayed in the status bar.

Symbol	Meaning	Description
	Undo (Ctrl+Z)	Undo previous action
	Restore	Restore previously undone action
	Cut (Ctrl+X)	Remove selected data and transfer to clipboard
	Copy (Ctrl+C)	Copy selected data and transfer to clipboard
	Paste (Ctrl+V)	Insert data from the clipboard
	Find (Ctrl+F)	Search for entered text
	Function text (F4)	Insert function text from the browser window See also section "Browse window ", page 10.
	Compile (F7)	Compile ST code
	Start debugging	Start debugging (cold start) After starting, a program that is already on the device is terminated. Instead, the current program is loaded onto the device. See also section "Debugging ", page 12.
	Terminate debugging	Terminate debugging After terminating, the current program remains on the device and is run. See also section "End ST editor ", page 12.
	Start/continue (F5)	Start program or continue running after interruption
	Interrupt program	Interrupt program
	Single step (F11)	Run program in a single step
	Exit	Terminate program
	Next bookmark (F2)	Move insertion mark to the next bookmark
	Toggle bookmark (Ctrl+F2)	Toggle bookmark for the current line
	Previous bookmark (Shift+F2)	Move insertion mark to the previous bookmark
	Delete bookmark (Shift+Ctrl+F2)	Delete all bookmarks
	Toggle breakpoint (F9)	Toggle breakpoint for the current line

Edit window

```
(*   @VERSION  1.02.345   *)

VAR   (* define of variables *)
(*    bExample        : BOOL;    *)
      uiExample       : UINT;
(*    udExample       : UDINT;   *)
(*    rExample        : REAL;    *)
(*    dtExample       : DT;      *)
END_VAR


VAR CONSTANT  (* define of constants *)
(*    bConstExample   : BOOL    := FALSE;                    *)
      uiConstExample  : UINT    := 0;
(*    udConstExample  : UDINT   := 100000;                   *)
(*    rConstExample   : REAL    := 0.0;                      *)
(*    dtConstExample  : DT      := dt#2015-01-01-08:00:00;   *)
END_VAR


(*  begin of code *)

if uiExample >100 then
 uiExample := 0;
else
 uiExample := uiExample + 1;
end_if;
```

The program is created in the edit window by declaring variables and constants, creating program code and using commentary texts.

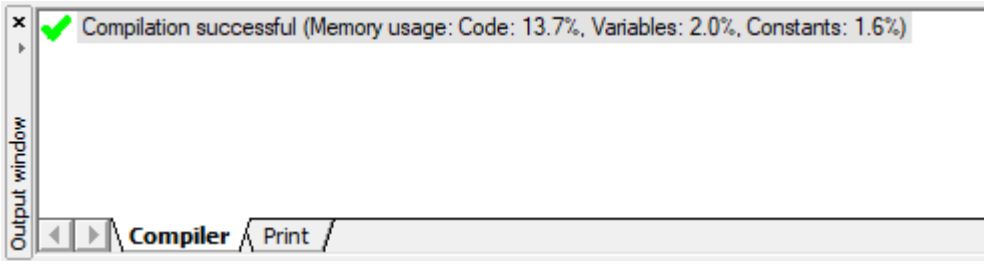
It is possible to change the font size using Ctrl+Mouse wheel or Ctrl+Shift+Arrow key.

Version name

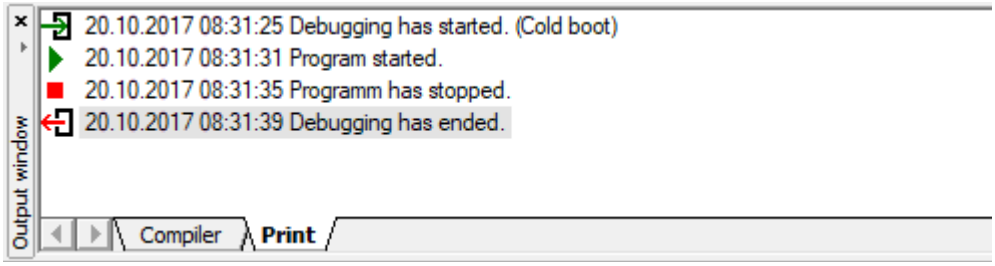
In order to assign a version name to the program, the keyword @VERSION followed by the version name must be used in a comment line. If the program contains several comment lines with this keyword, only the first comment line is evaluated. After the ST code has been permanently transferred to the device, the version name can be displayed on the device (device-dependent, e.g. Device info > Versions > ST code version).

Output window

The output window consists of two parts (tabs).

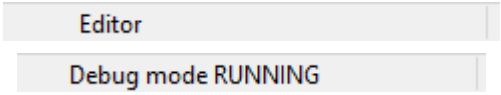


The "Compiler" tab shows messages concerning the compilation of the program code.



The "Output" tab contains messages concerning the debug mode.

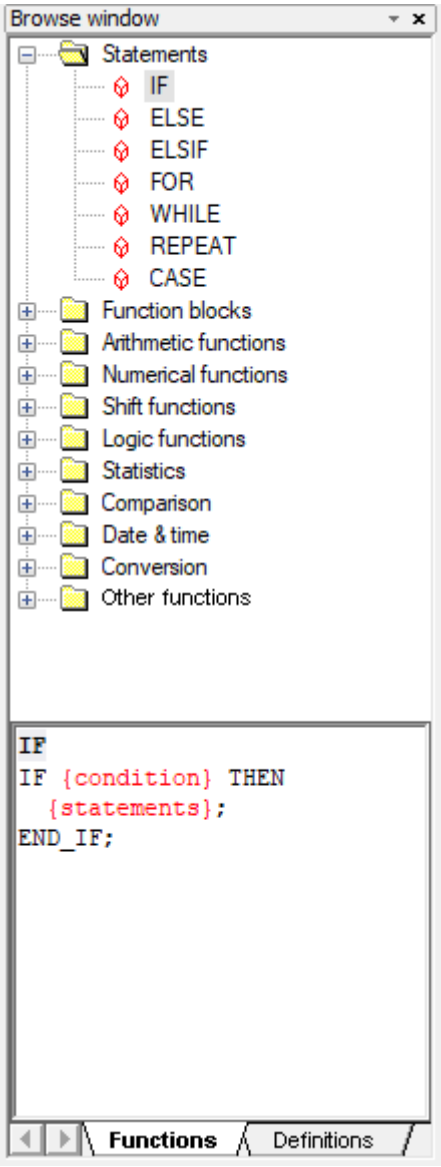
Status bar



The status bar is located at the bottom of the program interface and indicates whether the editor or debug mode is active. In the case of debug mode, the individual states (STOP, RUN, PAUSE) are displayed.

Further information about the functions of the menu bar and the toolbar is also displayed in the status bar (hover over the individual function in the respective menu or over the symbol in the toolbar with the mouse pointer).

Browse window



The browse window lists all the instructions and functions that can be used in the editor. The use of the instruction or function is shown in the bottom part of the window.

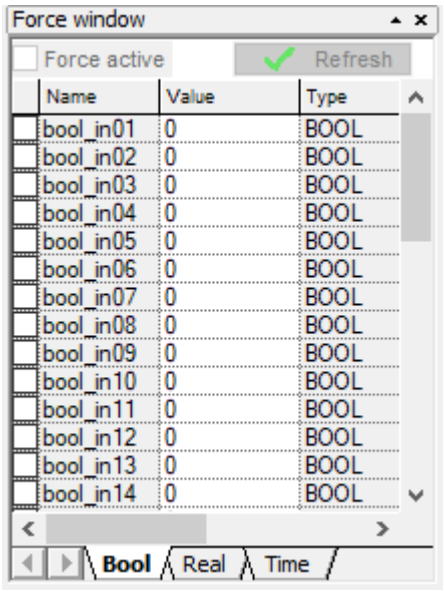
The function text displayed in the bottom part of the browse window can be inserted into the ST code in the edit window at the desired position (where the cursor is or instead of the selected text) by dragging and dropping (select text beforehand), by clicking on the "Function text" icon, or using the F4 function key. After inserting, the next placeholder is selected by clicking on the "Function text" icon again or pressing F4.

NOTE!

The "Definitions" tab is device-dependent and contains definitions (names for fixed values) that are factory-set or created by the user. If the ST module of the respective device supports definitions, these are described in a separate chapter "Definitions".

Force window

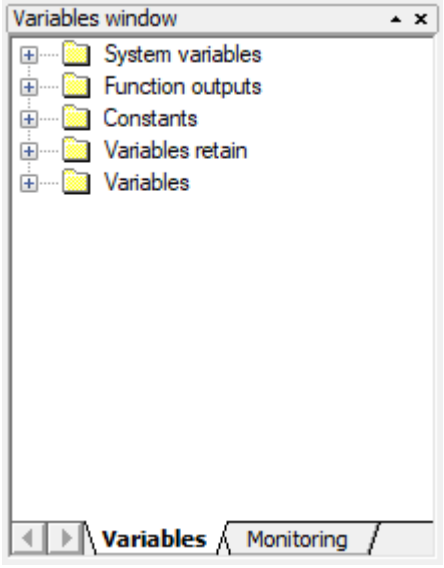
The force window consists of three parts (tabs). Only the "Bool" tab is shown here.



The "Bool" and "Real" input variables as well as date and time can be set by the user in the force window in order to test the program code.

Variables window

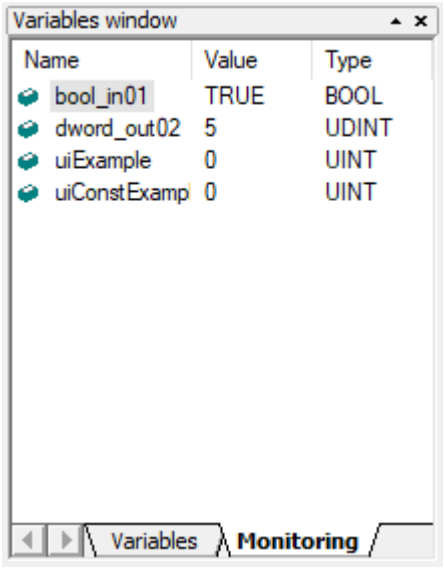
The variables window consists of two parts (tabs).



The "Variables" tab lists all system variables and function outputs (outputs of the function blocks) as well as the constants and variables (markers) declared by the user. They can be copied from the variable window by dragging and dropping and pasted into the ST code at the desired position in the edit window.

NOTE!

Variables that are stored via power off (retain variables) are not supported by all device types.



Variables and constants can be displayed and observed during debugging in the "Monitoring" tab. To do so, they must be copied from the edit window or the "Variables" tab by dragging and dropping and pasted into the "Monitoring" tab.

Pasting and removing is only possible if the program is not running. Different functions are then also available via the context menu (right mouse button). With the function "New..." a new variable can be added by entering a variable name (variable declaration may be required).

The user can assign alias names to system variables.

⇨ chapter 4.4 "Alias names", page 23

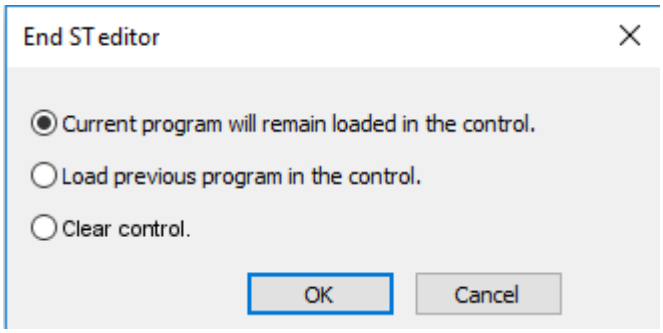
Debugging

The following variants are available with the *Debug > Start debugging* (menu bar) function:

- Cold start: A program that is already running on the device is terminated and the current program from the ST editor is loaded onto the device. All variables (retain and non-retain variables) are reset to their default values.
- Warm start: The retain variables are not reset.
- Service start: The program currently running on the device is used in the ST editor and can be analyzed. No variables are reset here. A prerequisite for the service start is that the code compiled in the ST editor corresponds to the code on the device.

End ST editor

When ending the ST editor with *Editor > Save and end (finish)* (menu bar), there are the following options (if debugging mode was active):



- *Current program will remain loaded on the control.*
  The program last loaded onto the device in debugging mode must be used.
- *Load previous program onto the control.*
  The program last loaded onto the device in debugging mode must not be used. Instead, the previous (continuously loaded onto the device) program is used, if available.
- *Clear control.*
  Neither the program last loaded onto the device in debugging mode nor the previous program must be used.

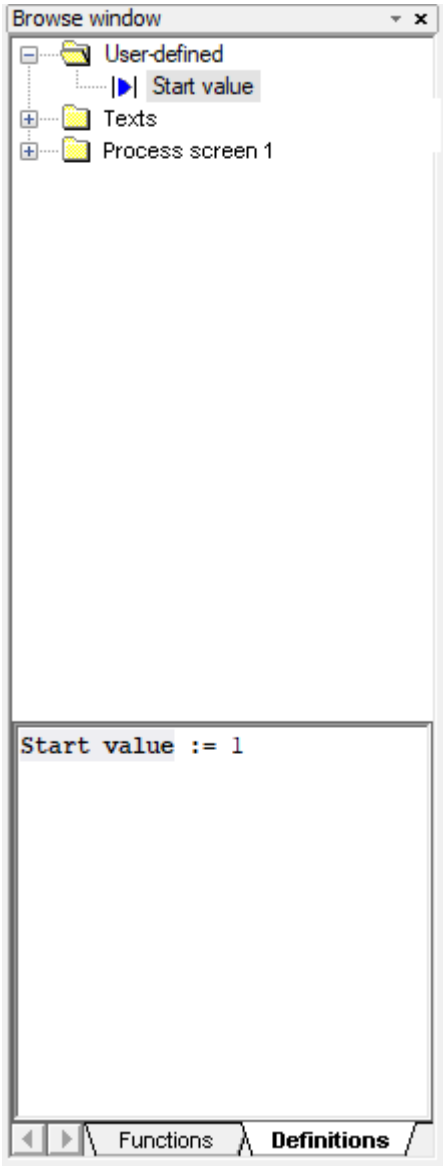
NOTE!

In debugging mode, the program is only temporarily loaded onto the device (no backup in case of a power failure). In order to make changes permanently effective on the device, the setup file must be transferred to the device after exiting the ST editor (close ST code window by pressing "OK", data transfer to the device).

2 Operation

The "Definitions" tab is device-dependent and contains definitions (names for fixed values) that are factory-set or created by the user. The "User-defined" (empty) and "Texts" folders are available per default. Depending on the configuration (setup program), further folders with the designation "Process screen x" (x = number of the process screen in the setup program) are used.

User-defined settings



The folder "User-defined" contains definitions created by the user in the ST editor. To create a definition, use the "New" function in the context menu (right-click on "User-defined").

A user-defined definition can be used to assign a name (here: start value) to any value (here: 1), which is used instead of the value in the ST code. The name should describe the meaning of the value, thereby increasing the comprehensibility of the ST code.

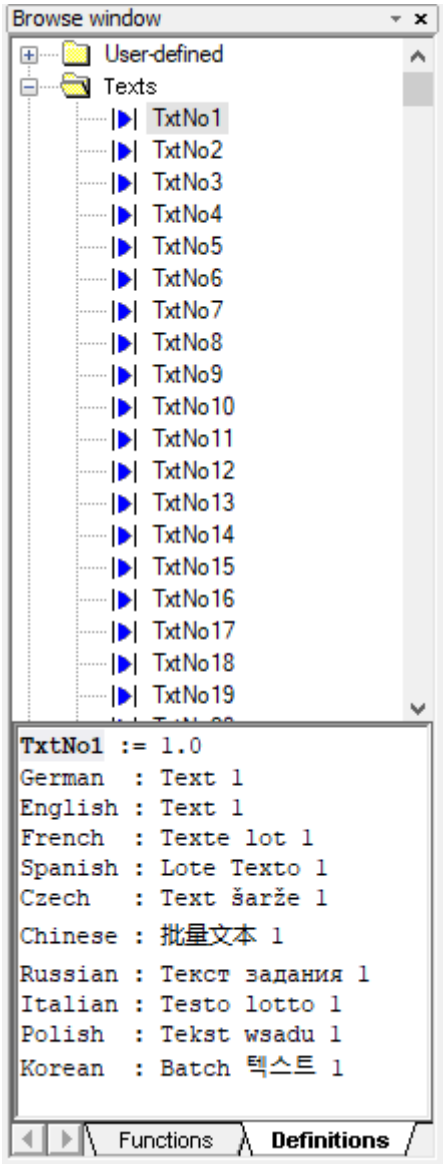
Another advantage of a user-defined definition is that it can be used several times in the ST code. If the value is changed, it is not necessary to change the ST code, only the value in the definition.

In the case of a CASE statement, definitions can be used instead of fixed values (constants are not allowed here).

Example:

```
CASE integer_variable OF
    definition1: integer_variable := 5;
    definition2, definition3: integer_variable := 10;
END_CASE;
```

Texts



The folder "Texts" lists all definitions (TxtNo1 to TxtNo...) which represent the numbers of the texts in the text list of the setup program. These definitions can be used in the ST code, for example, to address a batch text.

The definitions can be inserted from the upper part of the window into the ST code at the desired position in the editing window by dragging and dropping (or via the context menu: Copy).

In the lower part of the window, the definition is displayed in the first line (here: TxtIndex0 := 0). In the following lines, the respective text is displayed in the individual device languages.

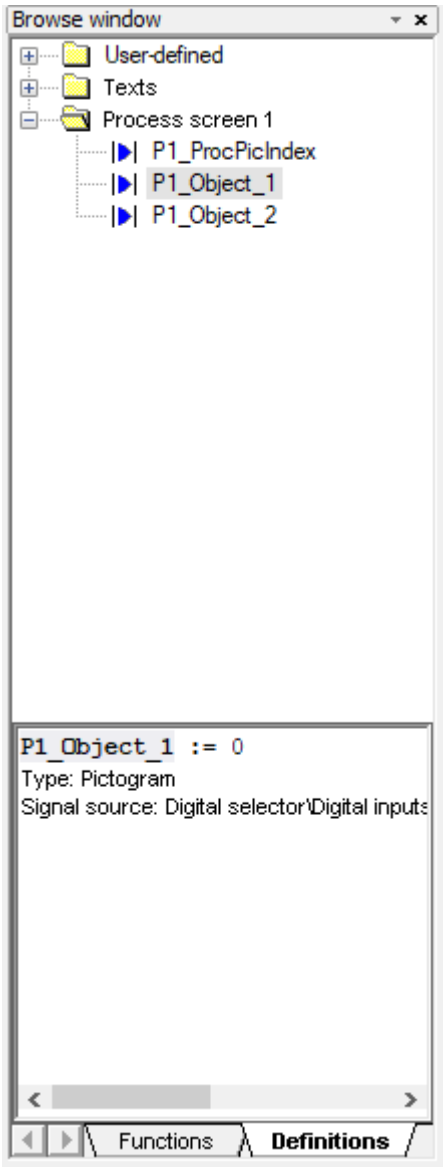
The texts themselves can also be changed in the ST editor (context menu: Alter... or Change...). When you exit the ST editor, the changes are transferred to the configuration in the setup program if you exit the "ST code" window in the setup program with "OK".

These definitions - as well as the possibility of changing texts in the ST editor - are intended in particular for the following purposes:

– Process screen: Control of the text in an object by an ST analog output (object "Text selection by value")
– Batch configuration: Specification of the text of the right column by a ST analog output (value as text number)

Further information can be found in the operating manual of the device.

Process screen x



If an object name was assigned to a process screen object in the setup program, it is displayed together with the process screen number in the browse window.

In such a case there is a folder for the corresponding process screen (here: process screen 1). In the folder name, the numbering of the process screens corresponds to that in the setup program (process screens 1 to 10).

3 Definitions

The folder contains the definitions for the process screen (here: P1_ProcPicIndex) and for each object for which a name was assigned in the process screen. The object name assigned by the user (here: object 1, ...) is preceded by a "Px_" for the relevant process screen (here: P1_Object_1, ...). This ensures that object names that are used in several process screens are unique in the ST code. A space in the object name is replaced by an underscore.

The lower part of the window contains information about the currently selected definition (selected in the upper part). The definition is listed in the first line (here: P1_Object_1 := 0). In the case of an object, there are additional lines that contain additional information.

The values assigned in the definitions correspond to the numbering of the process screens (0 to 9) or the objects (0 to 99) in the ST code.

Example: Definition for process screen 1

P1_ProcPicIndex := 0

Example: Definition for object 1 in process screen 1

P1_Objekt_1 := 0

If the number of the object in the process screen changes when the object is moved in the process screen editor (different order, for example, due to object overlays), the changed number is transferred to the ST editor the next time it is started. In the definition concerned, the previous number is then automatically replaced by the changed number so that the ST code does not have to be changed.

These definitions should be used in particular within the special functions that query and change certain properties of process screen objects.

⇨ chapter 8.9 "Other functions", page 46

NOTE!

In the ST code, the numbering of the process screens and objects begins with 0 (process screens 0 to 9, objects 0 to 99). In the setup program, numbering starts with 1 (process screens 1 to 10, objects 1 to 100).

NOTE!

If the process screen editor is closed with "OK" in the setup program, the ST code is recompiled. This ensures that the current definitions are used in the ST code.

The system variables include the input and output variables as well as the internal variables.

The input and output variables can be used to exchange values of different data types between the device and the ST module integrated in the device. The internal variables are only relevant within the ST module and can be used to implement certain functions.

4.1 Input variables

The input variables provide device values for use in the ST module.The input variables provide device values for use in the ST module.

The values are assigned to the input variables in the setup program by selecting the respective signals from the selectors (analog selector or digital selector).

Variables with fixed assignment in the device are not available in the selectors.

bool_in

Designation	Designation in the setup program	Description
bool_in01 ... bool_in40	bool_in01 ... bool_in40	Boolean input variables; flexible assignment in the setup program (digital selector)

real_in

Designation	Designation in the setup program	Description
real_in01 ... real_in40	real_in01 ... real_in40	Real input variables; flexible assignment in the setup program (analog selector)

dword_in

Designation	Designation in the setup program	Description
dword_in01 ... dword_in08	(no designation, no assignment)	These double-word input variables are not available on the device side. They can be used as a buffer in the ST code.

rtc.cdt

Designation	Designation in the setup program	Description
rtc.cdt	(no designation, fixed assignment)	This input variable of type DT supplies the current date and time of the device. ⇨ chapter 8.8 "Date and time", page 45

4.2 Output variables

The output variables are used to transfer the values generated in the ST module to the device.

The output variables are available for configuration in the setup program and in the selectors (analog selector or digital selector) on the device, and can be used individually.

For certain output variables there are associated variables in the variable window with the designations "...conf" and "...sec". These variables can be used to implement a defined behavior in the ST code in the event of an error.

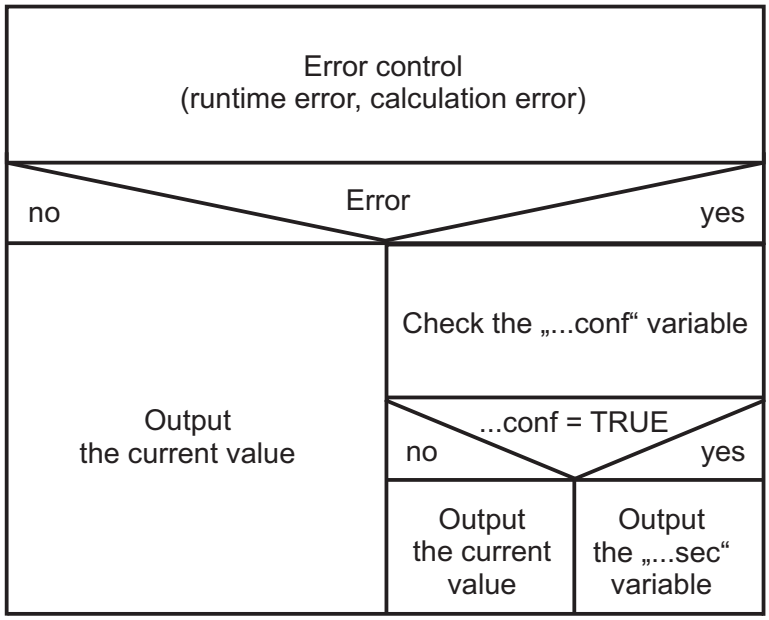
- The **variable "...conf"** can be set in the ST code to decide which value the corresponding output variable will accept in case of an error: FALSE = current value; TRUE = replacement value.

  Default setting: FALSE
- The **variable "...sec"** contains the replacement value; it can be set to a certain value or to the default value in the ST code.

  In case of REAL variables, the replacement value is checked for compliance with the scaling limits. If it is outside the limits, the default value is used (math error value 5.0E+37).

  The SET_DEFAULT function is available for setting the default value.

  ⇨ chapter 8.9 "Other functions", page 46



bool_out

Designation	Designation in the setup program and on the device	Description
bool_out01 ... bool_out40	ST digital output 1 ... ST digital output 40	Boolean output variables; flexible use in the setup program and on the device (digital selector) A description text can be assigned to each variable in the setup program.

real_out

Designation	Designation in the setup program and on the device	Description
real_out01 ... real_out40	ST analog output 1 ... ST analog output 40	Real output variables; flexible use in the setup program and on the device (analog selector) A description text can be assigned to each variable in the setup program. It also requires settings that affect the variable (e.g. scaling).

alarm

Designation	Designation in the setup program and on the device	Description
alarm01 ... alarm32	ST alarm output 1 ... ST alarm output 32	Boolean output variables; flexible use in the setup program and on the device (digital selector) Unlike the bool_out variables, this variables have a special designation. However, they can be freely used in the ST code.

error_out

Designation	Designation in the setup program and on the device	Description
error_out	ST error	Boolean output variable; flexible use in the setup program and on the device (digital selector) This variable is set to TRUE if the ST code cannot be processed further, such as in the following cases: • Division by 0 • Incorrect array access • Runtime exceeded The variable is set to FALSE again if the ST code has been completely run through once after the error occurred.

dword_out

Designation	Designation in the setup program and on the device	Description
dword_out01 ... dword_out08	(no designation, no assignment)	These double-word output variables are not available on the device side. They can be used as a buffer in the ST code.

4 System variables

4.3 Internal variables

The internal variables can be used in the ST code to implement certain functions.

ext_error

Designation	Description
ext_error	Integer variable The variable indicates errors that occurred during accesses outside the ST module. In the event of an error, the variable returns a value > 0 (no error = 0).

reset_flag

Designation	Description
reset_flag	Boolean variable The variable is set to TRUE after power-on and is used to initialize variables. Example: ⇨ chapter 7.1 "Selection instructions", page 31 If the ST code is completely run through once without an error after power on, the variable is set to FALSE. Otherwise, it stays as TRUE. Debug mode: The first time the program is started in debug mode or after a program stop, the variable is TRUE. After a second run through, it is set to FALSE.

sys_error

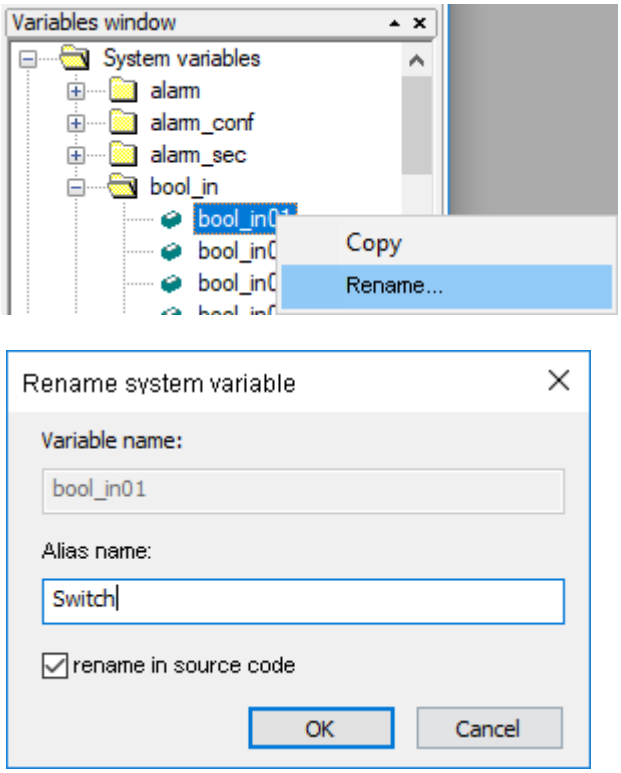
Designation	Description
sys_error	Real variable The variable returns an error value if an error occurs during program execution (no error = 0): 1.00E+37 = general error value 1.0E+37 = underrange (underflow) 2.0E+37 = overrange (overflow) 3.0E+37 = invalid input value 4.0E+37 = division by zero 5.0E+37 = incorrect math value 7.0E+37 = invalid value (general)

week_day

Designation	Description
week_day	Integer variable The variable includes the current day: 0 = Sunday, 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday The day is derived from the input variable rtc.cdt.

4.4 Alias names

Alias names can be assigned to system variables:

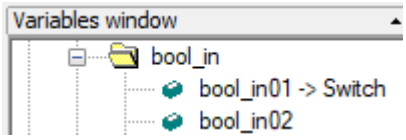


The function of the system variable can be described by using an alias name. In this example the alias name "Switch" is assigned to the system variable bool_in01.

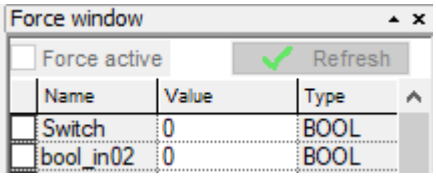
When the dialog is closed with "OK", the name entered is checked for uniqueness. If the name is already in use, a corresponding message appears. In this case another name must be assigned.

If the "rename in source code" function is active (checked), the variable name in the existing ST code is replaced by the alias name. Otherwise, the alias name is not used in the ST code until the variable is inserted again.

After assigning the alias name, the variable name and alias name are displayed in the variables window:



Only the alias name is displayed in the Force window:



4 System variables

5.1 Data types

The ST module supports the following data types:

- Boolean value (BOOL)
- Integer without prefix sign, 2 bytes (UINT)
- Double integer without prefix sign, 4 bytes (UDINT)
- Floating-point number (REAL)
- Date and time (DT or DATE_AND_TIME)

NOTE!

No range monitoring of the data types takes place during operations.
Exceptions: square root function (SRQT), reciprocal (1/x)

Boolean value

Keyword: BOOL

Value range: TRUE or FALSE

Declaration of a variable (example):

```
VAR
    bExample : BOOL;
END_VAR
```

Declaration of a constant (example):

```
VAR CONSTANT
    bConstExample : BOOL := FALSE;
END_VAR
```

Assignment (example):

```
bExample := TRUE;
```

Integer (2 bytes)

Keyword: UINT

Value range: 0 to 65535 (0 to 2^{16}-1)

Declaration of a variable (example):

```
VAR
    uiExample : UINT;
END_VAR
```

Declaration of a constant (example):

```
VAR CONSTANT
    uiConstExample : UINT := 0;
END_VAR
```

Assignment (example):

```
uiExample := 1;
```

Double integer (4 bytes)

Keyword: UDINT

Value range: 0 to 4294967295 (0 to 2^{32}-1)

Declaration of a variable (example):

```
VAR
    udExample : UDINT;
END_VAR
```

Declaration of a constant (example):

```
VAR CONSTANT
    udConstExample : UDINT := 100000;
END_VAR
```

Assignment (example):

```
udExample := 200000;
```

Floating-point number

Keyword: REAL

Value range: -1.5E-45 to 1.0E37
(Values ≥ 1.0E37 are interpreted as error values.)

Declaration of a variable (example):

```
VAR
    rExample : REAL;
END_VAR
```

Declaration of a constant (example):

```
VAR CONSTANT
    rConstExample : REAL := 0.0;
END_VAR
```

Assignment (example):

```
rExample := 1.0;
```

Date and time

Keywords: DT or DATE_AND_TIME

Value range: yyyy-mm-dd-hh:mm:ss

Declaration of a variable (example):

```
VAR
    dtExample : DT;
END_VAR
```

Declaration of a constant (example):

```
VAR CONSTANT
    dtConstExample : DT := dt#2017-12-31-23:59:59;
END_VAR
```

Assignment (example):

```
dtExample := dt#2017-01-01-08:00:00;
```

5.2 Fields

All data types can also be declared as a one-dimensional or two-dimensional field (array) (see standard DIN IEC 61131-3). The following sections use the data type REAL as an example.

One-dimensional field

Declaration of variables (example: field with 3 variables):

```
VAR
    rArrayExample : ARRAY [0..2] OF REAL;
END_VAR
```

Declaration of constants (example: field with 3 constants):

```
VAR CONSTANT
    rArrayConstExample : ARRAY [0..2] OF REAL := [0.0, 0.1, 0.2];
END_VAR
```

Assignment (example):

```
rArrayExample[0] := 0.0;
rArrayExample[1] := 0.1;
rArrayExample[2] := 0.2;
```

Two-dimensional field

Declaration of variables (example: field with 3 x 2 variables):

```
VAR
    rArrayExample : ARRAY [0..2, 0..1] OF REAL;
END_VAR
```

Declaration of constants (example: field with 3 x 2 constants):

```
VAR CONSTANT
    rArrayConstExample : ARRAY [0..2, 0..1] OF REAL :=
    [0.0, 0.1, 0.2,
    1.0, 1.1, 1.2];
END_VAR
```

Assignment (example):

```
rArrayExample[0,0] := 0.0;
rArrayExample[1,0] := 0.1;
rArrayExample[2,0] := 0.2;
rArrayExample[0,1] := 1.0;
rArrayExample[1,1] := 1.1;
rArrayExample[2,1] := 1.2;
```

5 Data types and fields

All operators supported by the ST module are shown in the following table. The order in the table depends on the ranking of the operators, starting with the highest rank.

Operation	Symbol	Admissible data types	Example
Brackets	(expression)		`a := 3.0 * (b - 1.0);`
Function	Identifier (argument list)		`i := MIN (3, j);`
Negation	-	REAL	`a := -a;`
Complement	NOT	BOOL, UINT, UDINT[a]	`a := NOT b;`
Multiplication	*	UINT, UDINT, REAL	`i := 5 * j;`
Division	/		`a := 5.0 / b;`
Modulo	MOD	UINT, UDINT	`j := i MOD 10;`
Addition	+	UINT, UDINT, REAL	`i := 5 + j;`
Subtraction	-		`a := b - 5.0E20;`
Comparison	<, >, <=, >=	UINT, UDINT, REAL, DATE_AND_TIME	`bExample := 5 <= j;`
Equivalence	=	BOOL, UINT, UDINT, REAL, DATE_AND_TIME	`bExample := 5 = i;`
Inequivalence	<>		`bExample := 5 <> j;`
AND	& or AND	BOOL, UINT, UDINT[a]	`bExample := x AND y;`
Exclusive OR	XOR	BOOL, UINT, UDINT[a]	`bExample := x XOR y;`
OR	OR	BOOL, UINT, UDINT[a]	`bExample := x OR y;`

[a] Boolean variables are logically linked, integer variables are linked bit-by-bit.

6 Operators

The ST module supports the following types of instructions:

- Selection instructions (IF, CASE)
- Repeat instructions (FOR, WHILE, REPEAT)

7.1 Selection instructions

A selection instruction executes an instruction or group of instructions based on a specified condition.

IF instruction

The IF instruction specifies that a group of instructions is only executed if the corresponding Boolean expression returns the value TRUE. If the condition is incorrect, either no instruction may be executed or the instruction group following the keyword ELSE (or the keyword ELSIF, if its associated Boolean condition is true) must be executed.

Keywords:

IF, THEN, ELSIF, ELSE, END_IF

Example:

```
VAR
    i : UINT;
END_VAR


IF reset_flag THEN
    bool_out01 := FALSE;
    bool_out02 := FALSE;
    i := 1;
END_IF;
(* after the reset, the outputs bool_out01 and bool_out02 are reset first and i is set
to 1 *)


IF i>0 AND i<=100 THEN
    bool_out01 := TRUE;
    bool_out02 := FALSE;
ELSIF i>100 AND i<=200 THEN
    bool_out01 := FALSE;
    bool_out02 := FALSE;
ELSE
    i := 1;
    bool_out01 := FALSE;
    bool_out02 := FALSE;
END_IF;
(* in cycle 1 to 100 the output bool_out01 is set and bool_out02 is reset *)
(* in cycle 101 to 200 the output bool_out01 is reset and bool_out02 is set *)
(* in cycle 201 i is set to 1 and the outputs bool_out01 and bool_out02 are reset; the
whole thing starts again *)


i := i + 1;
(* i is increased by 1 *)
```

CASE instruction

The CASE instruction combines several conditional instructions. It consists of an expression that contains an integer variable and a list of instruction groups. Each group is identified by a mark consisting of one or more integers (separated by commas). The value of the variable determines the mark and thus the instruction group to be executed. The keyword ELSE can also be used as an option. If no value applies, the instruction group following the keyword ELSE is executed.

Keywords:

CASE, OF, ELSE, END_CASE

Example:

```
VAR
    b1: BOOL;
    b2: BOOL;
END_VAR


CASE i OF (* variable i must be UINT or UDINT *)
    1: b1 := TRUE;
    2, 3, 4: b2 := TRUE;
ELSE
    b1 := FALSE;
    b2 := FALSE;
END_CASE;
```

7.2 Repeat instructions

With repeat instructions (iterations), instructions and groups of instructions are executed repeatedly.

There are three types of repeat instructions:

- FOR
- WHILE
- REPEAT UNTIL

FOR instruction

The FOR instruction is used if the number of repeats is fixed.

Keywords:

FOR, TO, BY, DO, END_FOR

Examples:

```
VAR
    i: UINT;
    j: UINT;
    a: UINT;
    b: UINT;
END_VAR
FOR i := 10 TO 100 BY 10 DO
    j := j + i;
END_FOR;
(* i runs from 10 to 100 in steps of 10 (10, 20, 30, …, 100)  *)
FOR a := 1 TO 5 DO
    b := b + a;
END_FOR;
(* if "BY x" is not specified, the control variable is increased by 1 in each cycle
(1, 2, 3, 4, 5) *)
```

WHILE instruction

The WHILE instruction causes a group of instructions to be repeatedly executed until the associated Boolean expression is incorrect (FALSE, untrue). If the Boolean expression is incorrect from the beginning, the group of instructions is not executed at all.

Keywords:

WHILE, DO, END_WHILE

Example:

```
WHILE j < 100 DO
    j := j + 2;
END_WHILE;
```

REPEAT instruction

The REPEAT instruction causes a group of instructions up to the keyword UNTIL to be repeatedly executed until the associated Boolean condition is true (TRUE).

Keywords:

REPEAT, UNTIL, END_REPEAT

Example:

```
REPEAT
    i := i - 2;
    a := a + 2.0;
UNTIL i = 0 END_REPEAT;
```

EXIT

With EXIT, a repeat instruction is terminated before the end condition of the iteration is fulfilled. If EXIT is executed within a nested repetition construct, the innermost loop is exited within the EXIT.

NOTE!

For repeat instructions, you must ensure that there are no endless loops or very long running loops. The runtime is monitored.

The ST module supports the following functions:

- Type conversion
- Arithmetic functions
- Numerical functions
- Bit sequence functions (shift functions)
- Logical functions
- Selection (statistics)
- Comparison
- Date and time
- Other functions

8.1 Type conversion

Admissible data types

Argument: UINT, UDINT

Result: BOOL, UINT, UDINT, REAL

INT_TO_REAL

Converts an INTEGER into a REAL number.

Example:

```
a := INT_TO_REAL(10); (* a := 10.0 *)
```

INT_TO_DINT

Converts an INTEGER into a DOUBLE INTEGER.

Example:

```
a := INT_TO_DINT(10); (* a := 10 *)
```

INT_TO_BOOL

Converts an INTEGER into a BOOL value.

The result is FALSE if the argument is 0. In all other cases, the result is TRUE.

Examples:

```
a := INT_TO_BOOL(0); (* a = FALSE *)
b := INT_TO_BOOL(1); (* b = TRUE *)
c := INT_TO_BOOL(8); (* c = TRUE *)
```

DINT_TO_REAL

Converts a DOUBLE INTEGER into a REAL number.

Examples:

```
a := DINT_TO_REAL(100000); (* a = 100000.0 *)
```

DINT_TO_INT

Converts a DOUBLE INTEGER into an INTEGER.

Only the lower two bytes are evaluated (bit 0 to bit 15).

Examples:

```
a := DINT_TO_INT(1); (* a = 1; 1 = 0000 0000 0000 0001 = 0x0001 *)
b := DINT_TO_INT(65535); (* b = 65535; 65535 = 1111 1111 1111 1111 = 0xFFFF *)
c := DINT_TO_INT(65536); (* c = 0; 65536 = 0001 0000 0000 0000 0000 = 0x10000 *)
d := DINT_TO_INT(65537); (* d = 1, 65537 = 0001 0000 0000 0000 0001 =0x10001 *)
```

DINT_TO_BOOL

Converts a DOUBLE INTEGER into a BOOL value.

The result is FALSE if the argument is 0. In all other cases, the result is TRUE.

Examples:

```
a := DINT_TO_BOOL(0); (* a = FALSE: *)
b := DINT_TO_BOOL(1); (* b = TRUE *)
c := DINT_TO_BOOL(100000); (* c = TRUE *)
```

REAL_TO_INT

Converts a REAL number with rounding into an INTEGER.

Examples:

```
a := REAL_TO_INT(1.378); (* a = 1 *)
b := REAL_TO_INT(1.897); (* b = 2 *)
```

REAL_TO_DINT

Converts a REAL number with rounding into a DOUBLE INTEGER.

Examples:

```
a := REAL_TO_DINT(100000.378); (* a = 100000 *)
b := REAL_TO_DINT(100000.897); (* b = 100001 *)
```

8.2 Arithmetic functions

Admissible data types

Argument: UINT, UDINT, REAL (only REAL for negation, only UINT or UDINT for modulo)

Result: UINT, UDINT, REAL

Addition

Addition is an expandable function. The sum of the arguments is returned as the result.

Example:

```
OUT := IN1 + IN2 + ... INn;
```

Subtraction

The second argument is subtracted from the first argument as a result.

Example:

```
OUT := IN1 - IN2;
```

Multiplication

Multiplication is an expandable function. The result is obtained by multiplying the arguments.

Example:

```
OUT := IN1 * IN2 * ...INn;
```

Division

The quotient of the two arguments is returned as the result.

The result of the division of integers (UINT or UDINT) is an integer with truncation of the decimal places (example: 7/3 = 2).

Example:

```
OUT := IN1 / IN2;
```

Negation

REAL numbers can be negated.

Example:

```
OUT := -IN;
```

Modulo

The arguments of the modulo function (%) must be integers (UINT or UDINT). The result is the remainder from the division of both numbers (example: 17/3 = 5, remainder 2).

Example:

```
OUT := IN1 % IN2;
```

8.3 Numerical functions

Admissible data types

Argument: REAL

Result: REAL

ABS (IN)

Returns the absolute value of IN.

Example:

```
IN := -2.3;
OUT := ABS(IN); (* OUT = 2.3 *)
```

SQRT (IN)

Returns the square root of IN.

If IN is negative, the error value 5.0E+37 is returned.

Example:

```
IN := 4.0;
OUT := SQRT(IN); (* OUT = 2.0 *)
```

LN (IN)

Returns the natural logarithm of IN.

Example:

```
IN := 2.718282;
OUT := LN(IN);
(* 2.718282 = number e (rounded); OUT = 1.0 (rounded) *)
```

LOG (IN)

Returns the base 10 logarithm of IN.

Example:

```
IN := 100.0;
OUT := LOG(IN); (* OUT = 2.0 *)
```

EXP (IN)

Returns the natural exponential function of IN.

Example:

```
IN := 2.0;
OUT := EXP(IN); (* OUT = 7.389056 *)
```

SIN (IN)

Returns the sine of IN (radian).

Example:

```
IN := 1.5708;
OUT := SIN(IN);
(* 1.5708 corresponds to 90°; OUT = 1.0 *)
```

COS (IN)

Returns the cosine of IN (radian).

Example:

```
IN := 0.0;
OUT := COS(IN);
(* 0.0 corresponds to 0°; OUT = 1.0 *)
```

TAN (IN)

Returns the tangent of IN (radian).

Example:

```
IN := 0.7854;
OUT := TAN(IN);
(* 0.7854 corresponds to 45°; OUT = 1.0 *)
```

ASIN (IN)

Returns the arc sine of IN (radian).

Example:

```
IN := 1.0;
OUT := ASIN(IN);
(* OUT = 1.5708; corresponds to 90° *)
```

ACOS (IN)

Returns the arc cosine of IN (radian).

Example:

```
IN := 1.0;
OUT := ASIN(IN);
(* OUT = 0.0; corresponds to 0° *)
```

ATAN (IN)

Returns the arc tangent of IN (radian).

Example:

```
IN := 1.0;
OUT := ATAN(IN);
(* OUT = 0.7854; corresponds to 45° *)
```

8.4 Bit sequence functions

Admissible data types

Argument: UINT, UDINT

Result: UINT, UDINT

SHL (IN, n)

Moves the bit sequence of the IN argument to the left by n bits. Subsequent digits are filled with 0 from the right.

Example:

```
IN := 255;          (* 0000 0000 1111 1111 *)
OUT := SHL(IN, 4); (* 0000 1111 1111 0000; OUT = 4080 *)
```

SHR (IN, n)

Moves the bit sequence of the IN argument to the right by n bits. Subsequent digits are filled with 0 from the left.

Example:

```
IN := 255;          (* 0000 0000 1111 1111 *)
OUT := SHR(IN, 4); (* 0000 0000 0000 1111; OUT = 15 *)
```

ROL (IN, n)

Rotates the bit sequence of the IN argument to the left by n bits in a circle.

Example:

```
IN := 43690;        (* 1010 1010 1010 1010 *)
OUT := ROL(IN, 1); (* 0101 0101 0101 0101; OUT = 21845 *)
```

ROR (IN, n)

Rotates the bit sequence of the IN argument to the right by n bits in a circle.

Example:

```
IN := 21845;        (* 0101 0101 0101 0101 *)
OUT := ROR(IN, 1); (* 1010 1010 1010 1010; OUT = 43690 *)
```

8.5 Logical functions

Boolean variables are linked logically, UINT and UDINT variables are linked bit-by-bit.

If there are more than two parameters, each pair is always linked from left to right.

Admissible data types

Operand: BOOL, UINT, UDINT

Result: BOOL, UINT, UDINT

AND

AND link

Examples:

Logical link:

```
x := TRUE;
y := FALSE;
z := x AND y; (* z = FALSE *)
```

Bit-by-bit link:

```
a := 5;        (* 0101 *)
b := 6;        (* 0110 *)
c := a AND b; (* 0100 = 4 *)
```

OR

OR link

Examples:

Logical link:

```
x := TRUE;
y := FALSE;
z := x OR y; (* z = TRUE *)
```

Bit-by-bit link:

```
a := 5;       (* 0101 *)
b := 6;       (* 0110 *)
c := a OR b; (* 0111 = 7 *)
```

XOR

Exclusive OR link

Examples:

Logical link:

```
x := TRUE;
y := FALSE;
z := x XOR y; (* z = TRUE *)
```

Bit-by-bit link:

```
a := 5;        (* 0101 *)
b := 6;        (* 0110 *)
c := a XOR b; (* 0011 = 3 *)
```

NOT

Inversion (form complement)

Examples:

```
IN := 1;
OUT := NOT IN; (* OUT = 0 *)

IN := 43690;   (* 1010 1010 1010 1010 *)
OUT := NOT IN; (* 0101 0101 0101 0101; OUT = 21845 *)
```

8.6 Selection

Admissible data types

Argument: UINT, UDINT, REAL (with SEL, additionally BOOL for the selector)

Result: UINT, UDINT, REAL

MAX

Returns the biggest argument as a result.

Example:

```
OUT := MAX(8, 9, 10, 11, 12); (* OUT := 12 *)
```

MIN

Returns the smallest argument as a result.

Example:

```
OUT := MIN(8, 9, 10, 11, 12); (* OUT := 8 *)
```

LIMIT

Checks whether a value (IN) is within a certain range (MIN, MAX). If the value is within the range, the value itself is returned. If the value is below the range, the lower range limit is returned. If the value is above the range, the upper range limit is returned.

Examples:

```
OUT := LIMIT(IN, MIN, MAX);
OUT := LIMIT(15, 10, 20); (* OUT = 15 *)
OUT := LIMIT(5, 10, 20); (* OUT = 10 *)
OUT := LIMIT(25, 10, 20); (* OUT = 20 *)
```

SEL

Selects one of two values (IN0, IN1) depending on a Boolean variable (G). The data type of the values IN0 and IN1 must be identical.

Examples:

```
OUT := SEL(G, IN0, IN1);
OUT := SEL(G, 10, 20); (* G = FALSE: OUT = 10 *)
OUT := SEL(G, 10, 20); (* G = TRUE: OUT = 20 *)
```

8.7 Comparison

Admissible data types

Argument with LT, LE, GT, GE: UINT, UDINT, REAL, DT

Argument with EQ and NE: BOOL, UINT, UDINT, REAL, DT

Result: BOOL

< (lower than)

Compares arguments on whether one is smaller than another.

Examples:

```
bOUT := IN1 < IN2; (* bOUT := TRUE, if IN1 is smaller than IN2 *)
bOUT := (IN1 < IN2) & (IN2 > IN3) & ... & (INn-1 > INn);
```

<= (lower or equal)

Compares arguments on whether one is smaller than or equal to another.

Examples:

```
bOUT := IN1 <= IN2; (* bOUT := TRUE, IN1 is smaller than or equal to IN2 *)
bOUT := (IN1 <= IN2) & (IN2 <= IN3) & ... & (INn-1 <= INn);
```

> (greater than)

Compares arguments on whether one is larger than another.

Examples:

```
bOUT := IN1 > IN2; (* bOUT := TRUE, if IN1 is larger than IN2 *)
bOUT := (IN1 > IN2) & (IN2 > IN3) & ... & (INn-1 > INn);
```

>= (greater or equal)

Compares arguments on whether one is larger than or equal to another.

Example:

```
bOUT := IN1 >= IN2; (* bOUT := TRUE, if IN1 is larger than or equal to IN2 *)
```

= (equal)

Compares arguments on whether they are equal.

Example:

```
bOUT := IN1 = IN2; (* bOUT := TRUE, if IN1 is equal to IN2 *)
```

<> (not equal)

Compares arguments on whether they are unequal.

Example:

```
bOUT := IN1 <> IN2; (* bOUT := TRUE, if IN1 is unequal to IN2 *)
```

8.8 Date and time

These are specific functions that are not part of the DIN IEC 61131-3 standard.

For example, these functions can be used to query the input variable rtc.cdt, which returns date and time of the device.

Admissible data types

Argument: DT, DATE_AND_TIME

Result: UINT

GET_YEAR

Returns the year (with century).

Examples:

```
iOUT := GET_YEAR(dt#2017-03-20-17:45:12); (* iOUT = 2017 *)
```

GET_MONTH

Returns the month.

Example:

```
iOUT := GET_MONTH(dt#2017-03-20-17:45:12); (* iOUT = 03 *)
```

GET_DAY

Returns the day.

Example:

```
iOUT := GET_DAY(dt#2017-03-20-17:45:12); (* iOUT = 20 *)
```

GET_HOUR

Returns the hour.

Example:

```
iOUT := GET_HOUR(dt#2017-03-20-17:45:12); (* iOUT = 17 *)
```

GET_MINUTE

Returns the minute.

Example:

```
iOUT := GET_MINUTE(dt#2017-03-20-17:45:12); (* iOUT = 45 *)
```

GET_SECOND

Returns the second.

Example:

```
iOUT := GET_SECOND(dt#2017-03-20-17:45:12); (* iOUT = 12 *)
iOUT := GET_SECOND(rtc_cdt); (* iOUT = 59, if rtc_cdt = 1970-01-01 00:00:59 *)
```

8.9 Other functions

These are specific functions that are not part of the DIN IEC 61131-3 standard.

IS_VALID

Checks the validity of a REAL number. The result is a Boolean value (valid = TRUE, invalid = FALSE).

Values ≥ 1.00E+37 (general error value) are considered as invalid, as are the values "Infinity" (INF) and "No number" (NaN).

Examples:

```
bOUT := IS_VALID(1.2); (* bOUT =  TRUE *)
```

SET_DEFAULT

Sets a REAL or a BOOL number to the default value.

Default value REAL: 5.0E+37 (math error value)
Default value BOOL: FALSE

Example:

```
SET_DEFAULT(bOUT); (* bOUT =  FALSE *)
SET_DEFAULT(rOUT); (* rOUT = 5.0E+37 *)
```

8.9.1 Process screen functions

The following functions can be used to query and change certain properties of process screen objects (colors, visibility, editability).

NOTE!

In the ST code, the numbering of the process screens and objects begins with 0 (e.g. process screens 0 to 9, objects 0 to 99). In the setup program, numbering starts with 1 (e.g. process screens 1 to 10, objects 1 to 100).

NOTE!

Instead of the numbers of the process screens and objects, their definitions can also be used.

Arguments of the functions

ProcPicIndex (UINT): Index (number) of the process screen (index 0 = process screen 1)

ObjectIndex (UINT): Index (number) of the object in the process screen (index 0 = object 1)

ColorType (UINT):
- 0 = Foreground color
- 1 = Background color
- 2 = Color for ON state (only for "Bar graph" object)
- 3 = Color for OFF state (only for "Bar graph" object)
- 4 = Color for LOW state (only for objects "Digital signal" and "Input digital value")
- 5 = Color for HIGH state (only for objects "Digital signal" and "Input digital value")

ValueRed (UINT): RGB value of the red portion of the color

ValueGreen (UINT): RGB value of the green portion of the color

ValueBlue (UINT): RGB value of the blue portion of the color

Value (BOOL): visibility or editability (TRUE, FALSE)

SET_OBJECT_COLOR

Sets the color of an object in a process screen.

The return value (BOOL) indicates whether the function was executed successfully (TRUE) or whether an error occurred (FALSE).

Order of arguments:
SET_OBJECT_COLOR (ProcPicIndex, ObjectIndex, ColorType, ValueRed, ValueGreen, ValueBlue)

Example:

```
bOUT := SET_OBJECT_COLOR(0, 9, 0, 255, 0, 0);
(* Sets the foreground color of process screen 1, object 10 to red. *)
```

GET_OBJECT_COLOR_RED

The return value (UINT) returns the RGB value (UINT) of the red portion of an object in a process screen.

Order of arguments:
GET_OBJECT_COLOR_RED (ProcPicIndex, ObjectIndex, ColorType)

Example:

```
iOUT := GET_OBJECT_COLOR_RED(0, 9, 0);
(* Returns the corresponding value of the foreground color of process screen 1, object
10. *)
```

GET_OBJECT_COLOR_GREEN

The return value (UINT) returns the RGB value (UINT) of the green portion of an object in a process screen.

Order of arguments:
GET_OBJECT_COLOR_GREEN (ProcPicIndex, ObjectIndex, ColorType)

Example:

```
iOUT := GET_OBJECT_COLOR_GREEN(0, 9, 1);
(* Returns the corresponding value of the background color of process screen 1, object
10. *)
```

GET_OBJECT_COLOR_BLUE

The return value (UINT) returns the RGB value (UINT) of the blue portion of an object in a process screen.

Order of arguments:
GET_OBJECT_COLOR_BLUE (ProcPicIndex, ObjectIndex, ColorType)

Example:

```
iOUT := GET_OBJECT_COLOR_BLUE(0, 9, 2);
(* Returns the corresponding color value for ON of process image 1, object 10. *)
```

SET_OBJECT_VISIBLE

Sets the visibility of an object in a process screen.

The return value (BOOL) indicates whether the function was executed successfully (TRUE) or whether an error occurred (FALSE).

Order of arguments:
SET_OBJECT_VISIBLE (ProcPicIndex, ObjectIndex, Value)

Value (BOOL): FALSE = not visible; TRUE = visible

Example:

```
bOUT := SET_ OBJECT_VISIBLE(1, 4, TRUE);
(* Sets the parameter "Visible" of process screen 2, object 5 to TRUE. *)
```

GET_OBJECT_VISIBLE

The return value (UINT) returns the status of the visibility of an object in a process screen.

Order of arguments:
GET_OBJECT_VISIBLE (ProcPicIndex, ObjectIndex)

Example:

```
bOUT := GET_ OBJECT_VISIBLE(1, 4);
(* Returns the status of the parameter "Visible" of process screen 2, object 5. *)
```

SET_OBJECT_EDITABLE

Sets the editability of an object in a process screen.

The return value (BOOL) indicates whether the function was executed successfully (TRUE) or whether an error occurred (FALSE).

Order of arguments:
SET_OBJECT_EDITABLE (ProcPicIndex, ObjectIndex, Value)

Example:

```
bOUT := SET_ OBJECT_EDITABLE(9, 19, FALSE);
(* Sets the parameter "Editable" of process screen 10, object 20 to FALSE. *)
```

GET_OBJECT_EDITABLE

The return value (UINT) returns the status of the editability of an object in a process screen.

Order of arguments:
GET_OBJECT_EDITABLE (ProcPicIndex, ObjectIndex)

Example:

```
bOUT := GET_ OBJECT_EDITABLE(9, 19);
(* Returns the status of the parameter "Visible" of process screen 10, object 20. *)
```

Instance

A function module – called a function block in the ST editor – is a program organizational unit that returns one or more values when executed (see also standard DIN IEC 61131-3). A function module can exist in several instances (copies). Each of these instances has a corresponding identifier. The instances of a function module are independent of each other.

If a function module called TON exists, for example in 4 instances, the individual instances are addressed with TON01, TON02, TON03, TON04.

Inputs and outputs

Parameters (one or more) are passed to the function module as input values.

The parameters follow, after the specification of the instance, in round brackets. The sequence of the parameters is fixed. Like any instruction, the call for a function module is completed with a ";".

Example: Calling the function module TON with instance 01:

```
TON01(TRUE,5,1);
```

The outputs are set by the function module and can be queried and used in any operation. An assignment of values is not possible.

Example: Querying of the output ET of the function module TON with instance 01:

```
OUT := TON01.ET;
```

The statuses of the inputs and outputs remain constant until the next call of the function module.

Function modules

The ST module of the paperless recorder supports the following function modules:

Identifier	Instances	Function
CTUD	32 1 – 16: non retain; 17 – 32: retain	Software up/down counter
TP	8 1 – 4: non retain; 5 – 8: retain	Impulse generator
TON	8 1 – 4: non retain; 5 – 8: retain	Switch on Delay
TOF	8 1 – 4: non retain; 5 – 8: retain	Switch off Delay
R_TRIG	8	Rising edge detection (0 -> 1)
F_TRIG	8	Falling edge detection (1 -> 0)
SR	8	Bistable function module (set priority)
RS	8	Bistable function module (reset priority)

9 Function modules

9.1 Software up/down counter

The counter provides the following functions:

- Counting up
- Counting down
- Reset the counter to 0
- Set an upper counter value
- Query in case the upper counter value is exceeded
- Query of counter status 0
- Query counter value

The counter counts the positive edges at the inputs CU (counting upwards) and CD (counting downwards).

Call

CTUD<instance> (CU, CD, R, LD, PV);

Inputs

The parameters R, LD, CU and CD are sorted in descending order in the table according to their priority.

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
R	BOOL	TRUE = set counter status to 0
LD	BOOL	TRUE = set upper counter value
CU	BOOL	Positive edge = count up
CD	BOOL	Positive edge = count down
PV	UINT	Upper counter value

Outputs

Parameter	Data type	Query; (* description *)
CV	UINT	OUT := CTUD<instance>.CV; (* OUT = counter status *)
QU	BOOL	OUT := CTUD<instance>.QU; (* OUT = TRUE, if counter status > upper counter value *)
QD	BOOL	OUT := CTUD<instance>.QD; (* OUT = TRUE, if counter status <= 0 *)

Comment

To set the upper counter value with LD, R must not be TRUE.

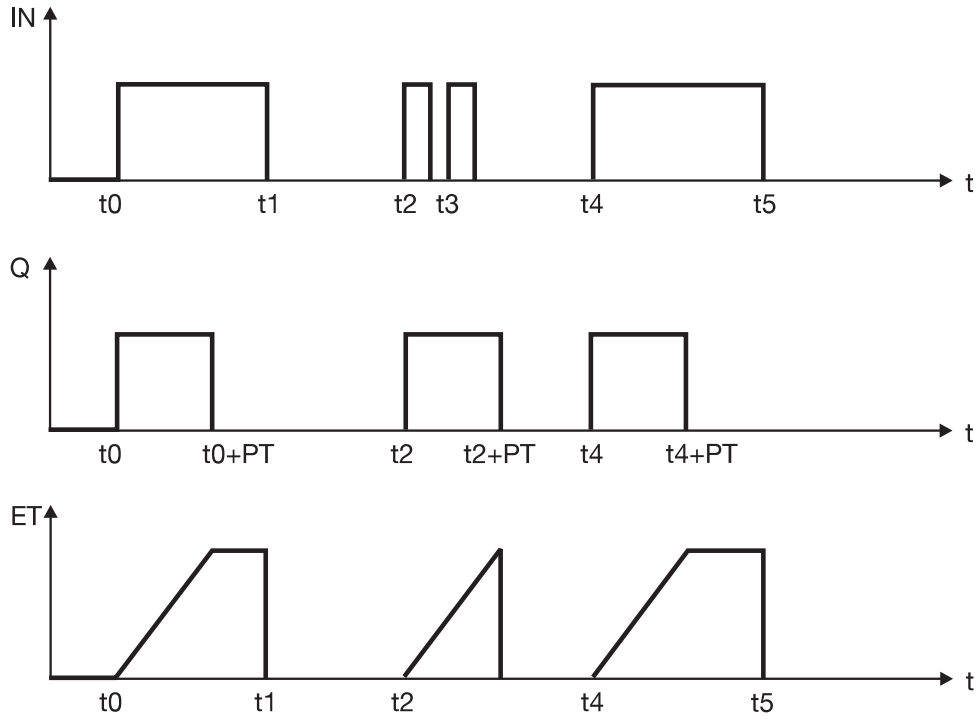
The values of the inputs/outputs are not retained when the power is turned off.

Examples

```
CTUD01 (IN, FALSE, FALSE, FALSE, 100);
(* count positive edges of IN upwards *)

CTUD01 (FALSE, IN, FALSE, FALSE, 100);
(* count positive edges of IN downwards *)

CTUD01 (IN, FALSE, TRUE, FALSE, 100);
(* set counter status to 0 *)

CTUD01 (IN, FALSE, FALSE, TRUE, 100);
(* set upper counter value: 100 *)
```

9.2 Impulse generator

The following diagram illustrates the operating principle of the impulse generator:



Call

TP<instance> (IN, PT, TimeBase);

Inputs

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
IN	BOOL	Positive edge at IN sets Q for the time PT to TRUE
PT	UINT	Time (unit: TimeBase), for Q = TRUE, IN = TRUE
TimeBase	UINT	Unit of PT: 1 = ms 2 = s 3 = min

Outputs

Parameter	Data type	Query; (* description *)
Q	BOOL	OUT := TP<instance>.Q; (* OUT = TRUE for the time PT, if IN := 0 -> 1 *)
ET	UINT	OUT := TP<instance>.ET; (* OUT = time passed since IN = TRUE until Q = FALSE (time passed since the start of the active impulse phase) *)

Comment

The TimeBase parameter is an extension to DIN IEC 61131-3. This extension does not constitute a limitation to the standard. According to the standard, the time limit is "implementation-dependent".

The values of the inputs/outputs are not retained when the power is turned off.

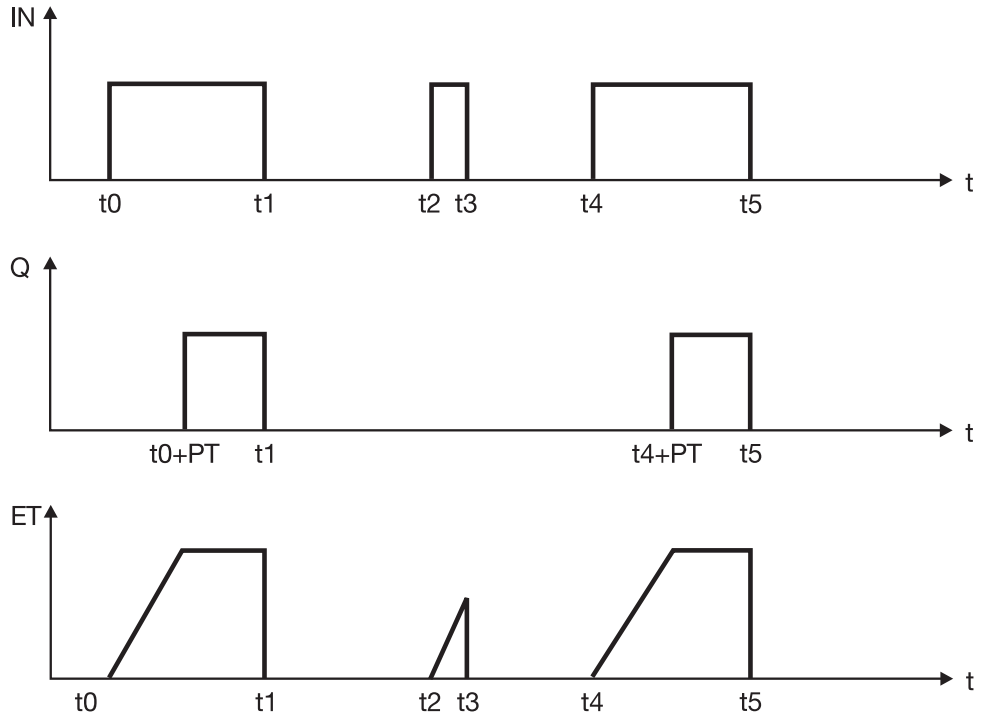
The time resolution of the pulse length depends on the cycle time of the device (example: 150 ms).

Example

```
TP01(IN, 5, 3);
(* Positive edge at IN sets TP01.Q for approx. 5 minutes to TRUE (note time resolu-
tion).
TP01.ET returns the elapsed time in minutes since the beginning of the active pulse
phase. *)
```

9.3 Switch-on delay

The following diagram illustrates the operating principle of the switch-on delay:



Call

TON<instance> (IN, PT, TimeBase);

Inputs

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
IN	BOOL	IN = TRUE sets Q = TRUE, **after** the time PT is expired
PT	UINT	Delay time (unit: TimeBase)
TimeBase	UINT	Unit of PT: 1 = ms 2 = s 3 = min

Outputs

Parameter	Data type	Query; (* description *)
Q	BOOL	OUT := TON<instance>.Q; (* OUT = TRUE, if IN = TRUE and PT expired *)
ET	UINT	OUT := TON<instance>.ET; (* OUT = time passed since IN = TRUE until Q = TRUE or until IN = FALSE *)

Comment

The TimeBase parameter is an extension to DIN IEC 61131-3. This extension does not constitute a limitation to the standard. According to the standard, the time limit is "implementation-dependent".

The values of the inputs/outputs are not retained when the power is turned off.

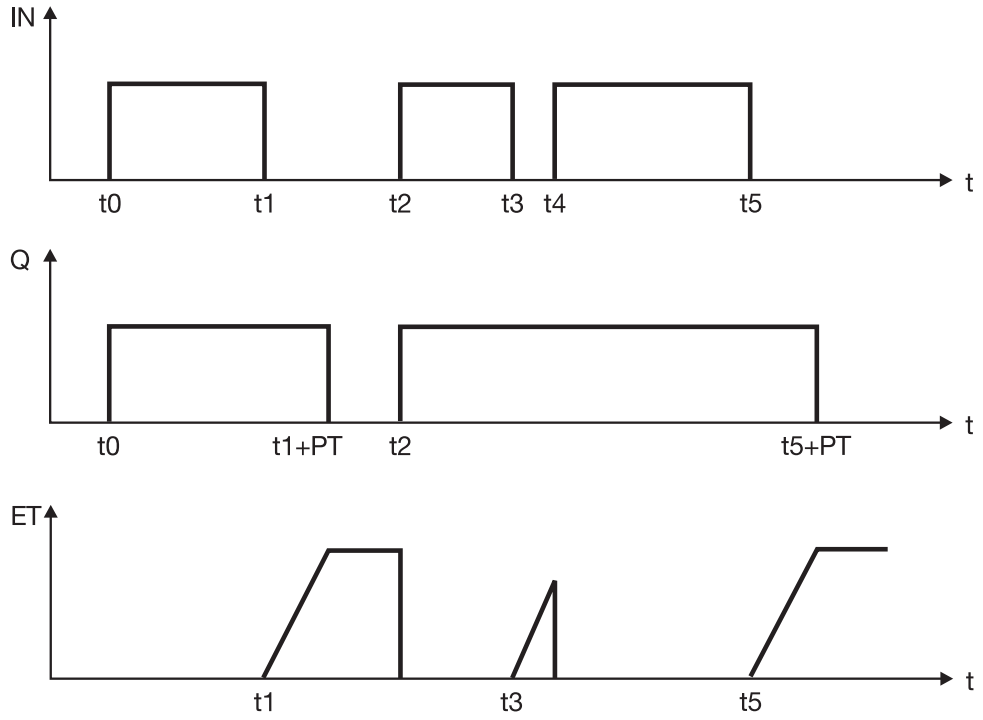
The time resolution of the delay time depends on the cycle time of the device (example: 150 ms).

Example

```
TON01(IN, 6, 2);

(* IN = TRUE sets after 6 seconds TON01.Q to TRUE.
TON01.Q remains TRUE as long as IN = TRUE.
TON01.ET returns the elapsed time in seconds from IN = TRUE to TON01.Q = TRUE (max. 6
seconds) or to IN = FALSE. *)
```

9.4 Switch-off delay

The following diagram illustrates the operating principle of the switch-off delay:



Call

TOF<instance> (IN, PT, TimeBase);

Inputs

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
IN	BOOL	IN = TRUE sets Q = TRUE IN = FALSE sets Q around the time PT delayed to FALSE
PT	UINT	Delay time (unit: TimeBase)
TimeBase	UINT	Unit of PT: 1 = ms 2 = s 3 = min

Outputs

Parameter	Data type	Query; (* description *)
Q	BOOL	OUT := TOF<instance>.Q; (* OUT = TRUE, if IN = TRUE or if IN = FALSE and PT has not yet expired *)
ET	UINT	OUT := TOF<instance>.ET; (* OUT = time passed since IN = FALSE until Q = FALSE or until IN = TRUE *)

Comment

The TimeBase parameter is an extension to DIN IEC 61131-3. This extension does not constitute a limitation to the standard. According to the standard, the time limit is "implementation-dependent".

The values of the inputs/outputs are not retained when the power is turned off.

The time resolution of the delay time depends on the cycle time of the device (example: 150 ms).

Example

```
TOF01(IN, 450, 1);

(* IN = TRUE sets TOF01.Q to TRUE.
After IN = FALSE, TOF01.Q remains TRUE until the time PT = 450  ms has expired.
TOF01.ET returns the time passed in milliseconds from IN = FALSE until TOF01.Q = FALSE
(max. 450 ms) or until IN = TRUE. *)
```

9.5 Rising edge detection

This function module detects a rising edge (transition 0 -> 1) .

Call

R_TRIG<instance> (CLK);

Input

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
CLK	BOOL	Due to a rising edge (0 -> 1) at CLK, Q = TRUE

Output

Parameter	Data type	Query
Q	BOOL	OUT := R_TRIG<instance>.Q;

Comment

The operating principle of the function module corresponds to the following program code:

VAR CLK : BOOL; END_VAR

VAR Q : BOOL; END_VAR

VAR M : BOOL := FALSE; END_VAR

Q := CLK AND NOT M;

M := CLK;

The Q output remains at its Boolean value from one call until the next. It follows the transition of the input signal (CLK) from "0" to "1" and returns to "0" the next time it is called.

Example

```
IF reset_flag THEN
bool_out01 := FALSE;
END_IF;

R_TRIG01 (bool_in01);
(* detection of a rising edge at input bool_in01 *)

bool_out01 := R_TRIG01.Q;
(* short pulse at output bool_out01 for rising edge at input *)
```

9.6 Falling edge detection

This function module detects a falling edge (transition 1 -> 0) .

Call

F_TRIG<instance> (CLK);

Input

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
CLK	BOOL	Due to a falling edge (1 -> 0) at CLK, Q = TRUE

Output

Parameter	Data type	Query
Q	BOOL	OUT := F_TRIG<instance>.Q;

Comment

The operating principle of the function module corresponds to the following program code:

VAR CLK : BOOL; END_VAR

VAR Q : BOOL; END_VAR

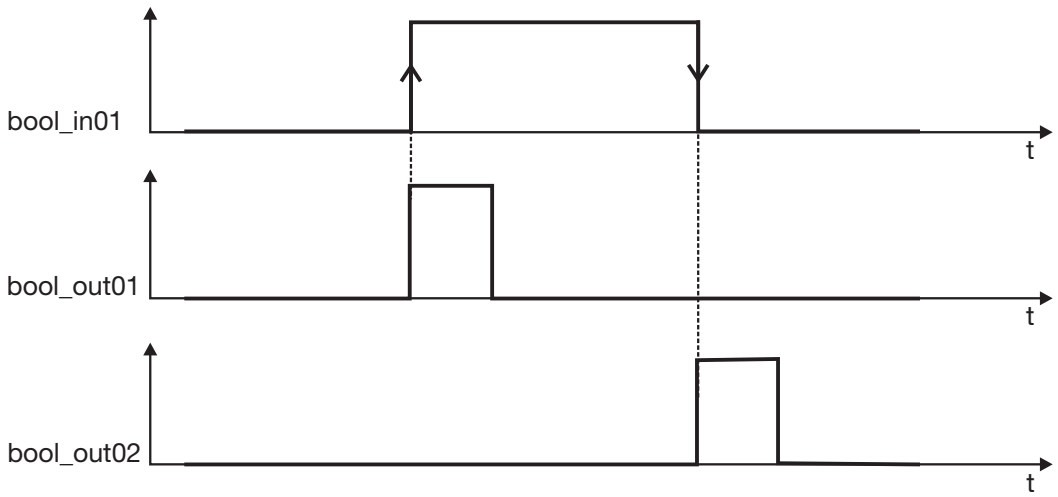
VAR M : BOOL := TRUE; END_VAR

Q := NOT CLK AND NOT M;

M := NOT CLK;

The Q output remains at its Boolean value from one call until the next. It follows the transition of the input signal (CLK) from "1" to "0" and returns to "0" the next time it is called.

Example with R_TRIG and F_TRIG




```
IF reset_flag THEN
bool_out01 := FALSE;
bool_out02 := FALSE;
END IF;

R_TRIG01 (bool_in01);
(* detection of a rising edge at input bool_in01 *)

bool_out01 := R_TRIG01.Q;
(* short pulse at output bool_out01 for rising edge at input *)
```

```
F_TRIG01 (bool_in01);
(* detection of a falling edge at input bool_in01 *)
bool_out02 := F_TRIG01.Q;
(* short pulse at output bool_out02 for falling edge at input *)
```

9 Function modules

9.7 Bistable function SR

Bistable function (set priority) with lock

Call

SR<instance> (S1, R);

Inputs

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
S1	BOOL	S1 = TRUE sets Q1 = TRUE
R	BOOL	R = TRUE sets Q1 = FALSE, if S1 = FALSE

Outputs

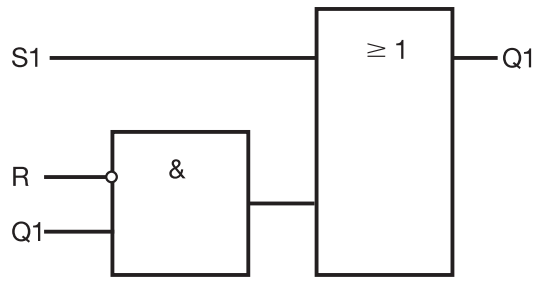
Parameter	Data type	Query
Q1	BOOL	OUT := SR<instance>.Q1;

Comment

Output Q1 remains TRUE after being set by S1 until it is reset with R.

If S1 and R are set simultaneously (TRUE), output Q1 is also set (TRUE).

Logic function



Example

```
SR01 (bool_in01, bool_in02);
(* set with input bool_in01, reset with input bool_in02 *)
bool_out01 := SR01.Q1;
(* The status is output via bool_out01. *)
```

9.8 Bistable function RS

Bistable function (reset priority) with lock

Call

RS<instance> (S, R1);

Inputs

Parameter	Data type	Description
<Instance>		01 to xx (specify instance with two digits)
R1	BOOL	R1 = TRUE sets Q1 = FALSE
S	BOOL	S = TRUE sets Q1 = TRUE, if R1 = FALSE

Outputs

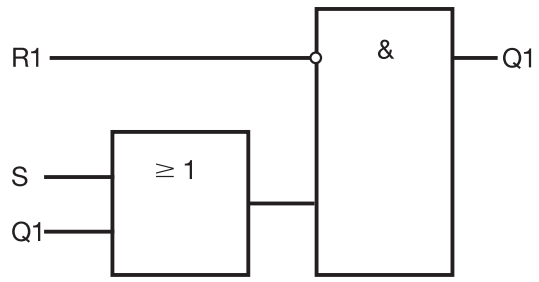
Parameter	Data type	Query
Q1	BOOL	OUT := RS<instance>.Q1;

Comment

After resetting with R1, output Q1 remains FALSE until it is set with S.

If R1 and S are set simultaneously (TRUE), output Q1 is reset (FALSE).

Logic function



Example

```
RS01 (bool_in01, bool_in02);
(* reset with input bool_in02, set with input bool_in01 *)
bool_out02 := RS01.Q1;
(* The status is output via bool_out02. *)
```

9 Function modules

JUMO GmbH & Co. KG
Street address:
Moritz-Juchheim-Straße 1
36039 Fulda, Germany
Delivery address:
Mackenrodtstraße 14
36039 Fulda, Germany
Postal address:
36035 Fulda, Germany
Phone: +49 661 6003-0
Fax: +49 661 6003-607
Email: mail@jumo.net
Internet: www.jumo.net

JUMO Instrument Co. Ltd.
JUMO House
Temple Bank, Riverway
Harlow, Essex, CM20 2DY, UK
Phone: +44 1279 63 55 33
Fax: +44 1279 62 50 29
Email: sales@jumo.co.uk
Internet: www.jumo.co.uk

JUMO Process Control, Inc.
6733 Myers Road
East Syracuse, NY 13057, USA
Phone: +1 315 437 5866
Fax: +1 315 437 5860
Email: info.us@jumo.net
Internet: www.jumousa.com