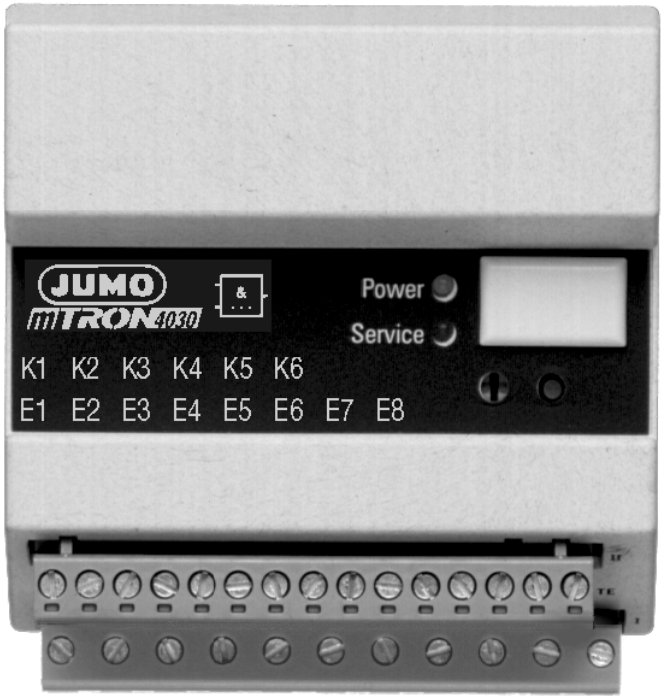


JUMO mTRON

Logic module

70.4030
System Manual Part 7

Contents

1.1 Preface

The System Manual is addressed to the OEM (original equipment manufacturer) and to the user with the appropriate technical know-how. It describes the range of features of the JUMO mTRON automation system with its modules, and provides all the information required for system design and start-up.

This Part 7 of the System Manual “JUMO mTRON logic module” contains all the module-specific information.
The System Manual Part 1 “General section” summarises the information which applies to all modules.
Part 2 of the System Manual “JUMO mTRON-iTOOL project design software” describes project design for the JUMO mTRON automation system.

1.2 Type designation

The type designation contains all factory-configured settings of the inputs (1), outputs (2) and the supply (3). The connected supply voltage must correspond to the voltage specified on the label. The label is affixed to the housing.

	(1)		(2)		(3)
704030/0-	. . .	-	. . .	-	. .

(1) Inputs . . .

Inputs	Code
8 logic inputs, electrically isolated from the system	178
8 voltage inputs 0/24 V	188

(2) Outputs . . .

Outputs	Code
6 logic outputs (Relay, n.o.(make))	156
6 open-collector outputs (transistor)	176

(3) Supply . . .

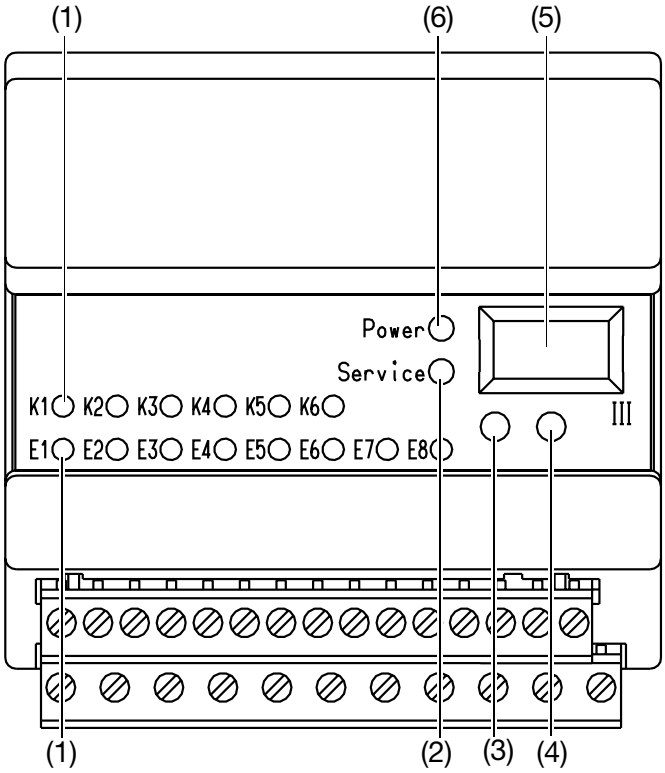
Type	Code
110 — 240 V +10/-15 % AC 48 — 63 Hz	23
20 — 53 V AC/DC 48 — 63 Hz	22

Neuron-ID

Each module has a 12 digit number, by which it can also be clearly identified in the JUMO mTRON-iTOOL project design software.

It can be found next to the label.

1 Introduction



LEDs

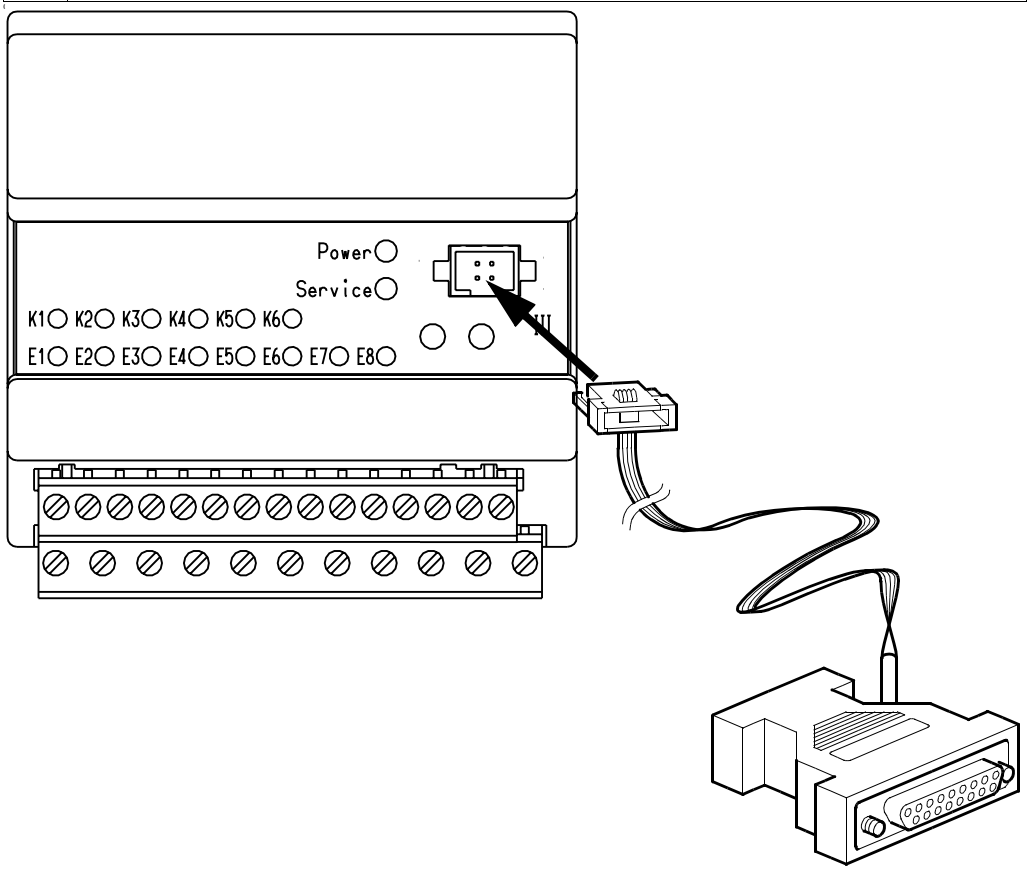
(1)	**Status LED (yellow)** for the outputs K1 to K6 and the logic inputs E1 to E8, lights up if the output is active / contact closed or voltage on the logic input
(2)	**Service LED (red)** - lights up /blinks continuously at one second intervals on operating fault ✱ replace module - blinks at one second intervals for 10 sec when the network connection from the JUMOmTRON-iTOOL project design software or the operating unit to the module is checked by a test signal (“wink”). - long blink pulses (3 sec on, 1sec off) when a Plug & Play error has occurred. - blink pulses (2sec on; 2sec off) when the instrument is in calibration mode.
(6)	**Power LED (green)** lights up when the supply is switched on

Keys/switches

(3)	**Switches (termination resistance)** ⇨ System Manual Part 1 “General section”, Section 4.2 “Network connection”
(4)	**Installation key** the module reports to the JUMO mTRON-iTOOL project design software

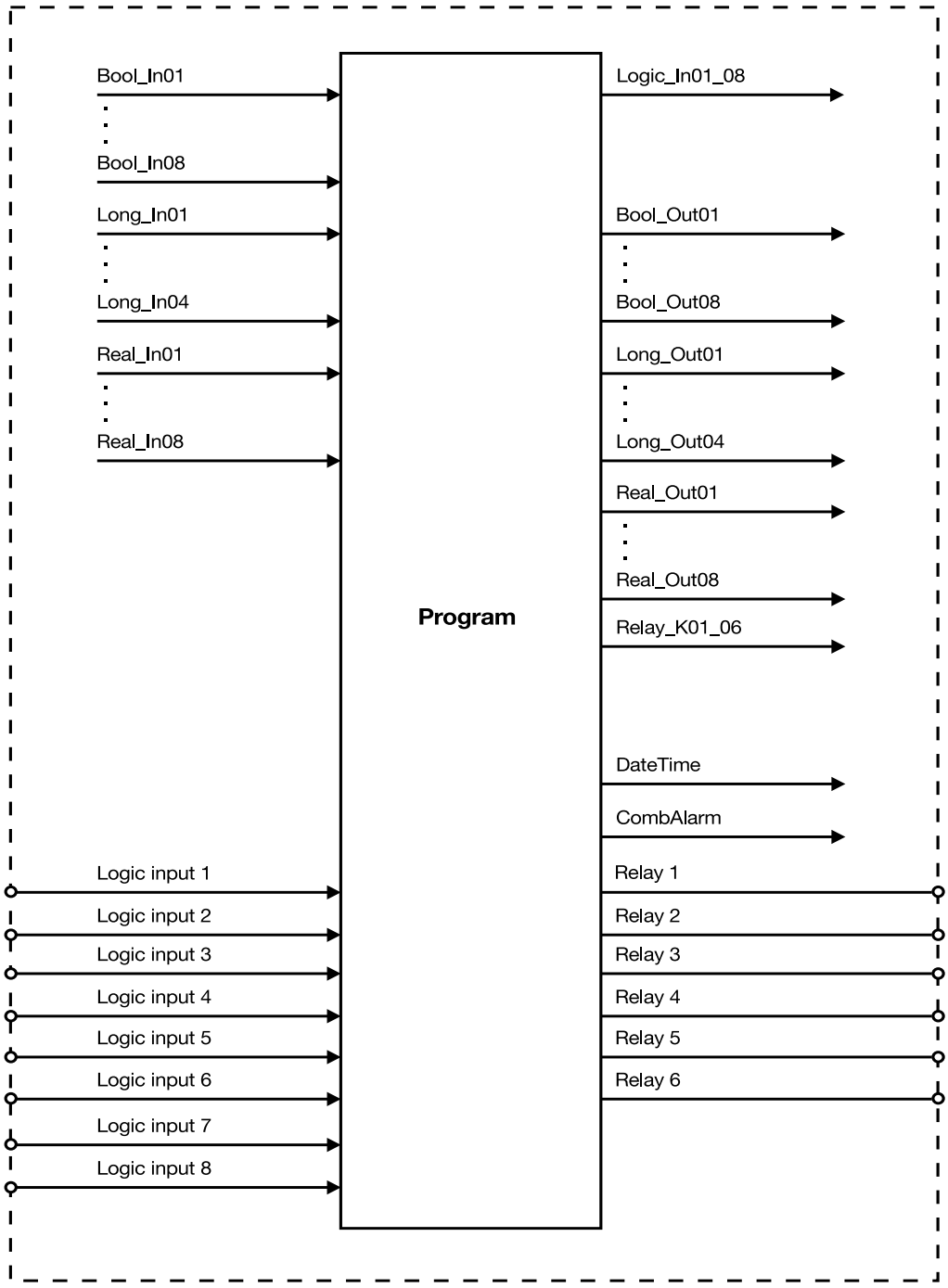
Interface

(5)	**Setup interface** for the setup interface line which connects the module to the PC. The parameters can be set via this connector not only for the logic module, but also **for all the modules connected the the LON bus.**



When the interface line is connected, the module has the sole function of a PC-LON interface converter. All other module functions are switched off.

The diagram shows the input and output variables which can be processed through a structured-text program.



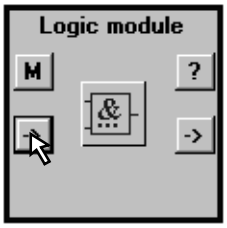
Explanation of symbols

Symbol	Meaning
Logic_In3 →	Network variable ⇨Chapter 4 “Network variables”
⊢	Hardware input
⊣	Hardware output

3 Overview of functions

4.1 Input network-variables

List of input network-variables



The input network-variables are available to transmit values and operating signals from other modules to the logic module via the network.

The output network-variables are updated every 420 msec.

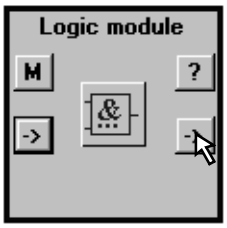
Name	Type	Description
Bool_In01 Bool_In02 Bool_In03 Bool_In04 Bool_In05 Bool_In06 Bool_In07 Bool_In08	logic	Logic input signals ST: BOOL
Long_In01 Long_In02 Long_In03 Long_In04	long	Input values of type “long” (16 bits) ST: UINT
Real_In01 Real_In02 Real_In03 Real_In04 Real_In05 Real_In06 Real_In07 Real_In08	float value (floating point number)	Input values of type “float value” (32 bits) ST: REAL

Unbound and bound input network-variables which are not refreshed have the following standard values:
Float: 0.0
Bool: False
Long: 0

4 Network variables

4.2 Output network-variables

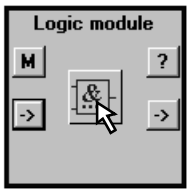
List of output network-variables



The output network-variables can be used to transmit values and operating signals from the logic module to other modules via the network.

Name	Type	Description
Log_01_08	long (UINT)	Produces the switching states of the logic inputs 1 — 8 Coding by bit: bit 0 = input 1 . . . bit 7 = input 8
Bool_Out01 Bool_Out02 Bool_Out03 Bool_Out04 Bool_Out05 Bool_Out06 Bool_Out07 Bool_Out08	logic	Logic outputs 1 — 8 ST: BOOL
DateTime	SNVT	Standard network-variable for date and time: 7 bytes LON-SNVT type
Long_Out01 Long_Out02 Long_Out03 Long_Out04	long	Output values of type “long” (16 bits) ST:UINT
Real_Out01 Real_Out02 Real_Out03 Real_Out04 Real_Out05 Real_Out06 Real_Out07 Real_Out08	float value (floating point number)	Output values of type “float value” (32 bits) ST: REAL
Relay_K01_06	long	Switching states of relays 1 — 6 Coding by bit: bit 0 = relay 1 . . . bit 5 = relay 6
CombAlarm	logic	Produces a combination alarm after 18 sec in the event of faulty communication on a bound input network-variable. ⇨ Chapter 5.3

Basic menu



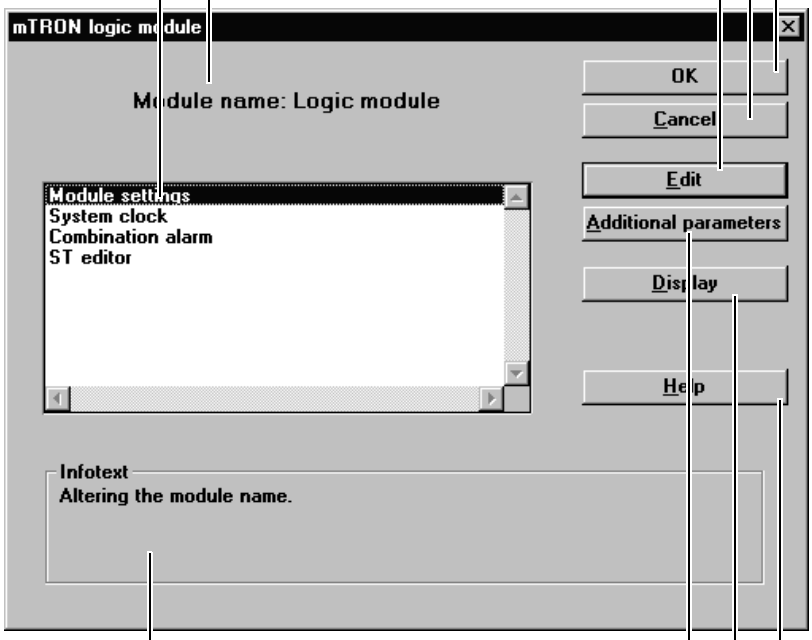
Module name
Name of the module

OK
for entering and storing all inputs

Setup dialog
The functions of the module are assigned to setup dialogs.

Cancel
for aborting inputs.
The data are not stored.

Edit
for editing parameters in the setup dialog which is marked



Additional parameters
Further settings can be made here when there are differences between the versions of module software and setup program

Display
Using this function, individual parameters can be removed from the operating unit (parameter level)

Info text
provides information on the setup dialog which is marked

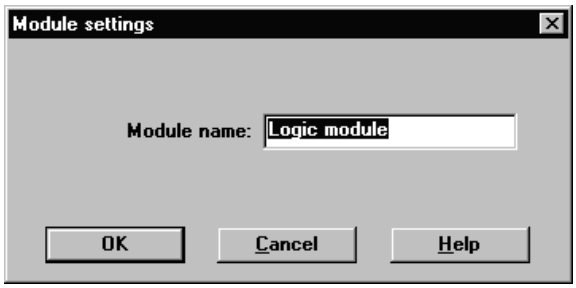
Help
calls up help text for the basic menu

5 Parameter setting

5.1 Module settings

The name of the module is determined in the module setting.

Setup dialog



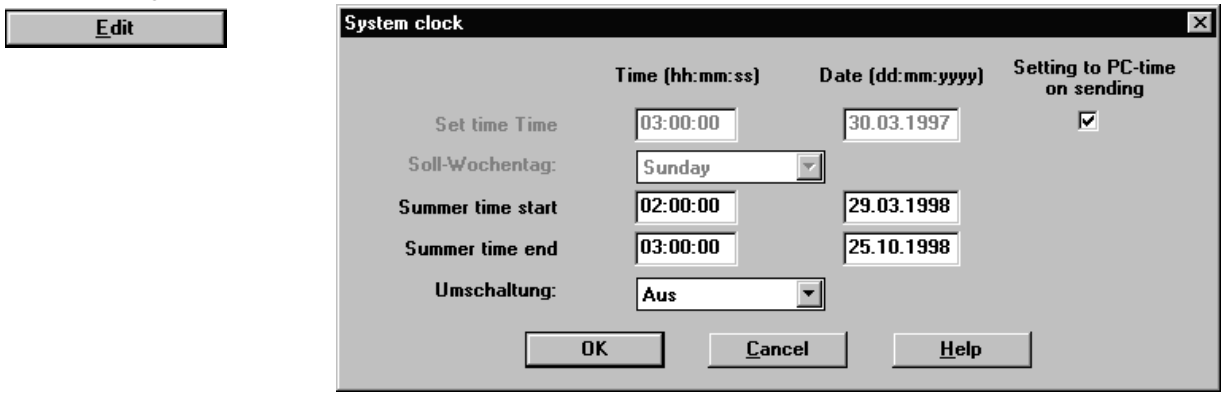
Parameters

Parameter	Selection/settings	Explanation
Module name	(Text)	Name of the module (16 characters)

5.2 System clock

The system clock of the logic module is set and the times for the switch over to summer / winter time are defined here.

Setup dialog



Parameters

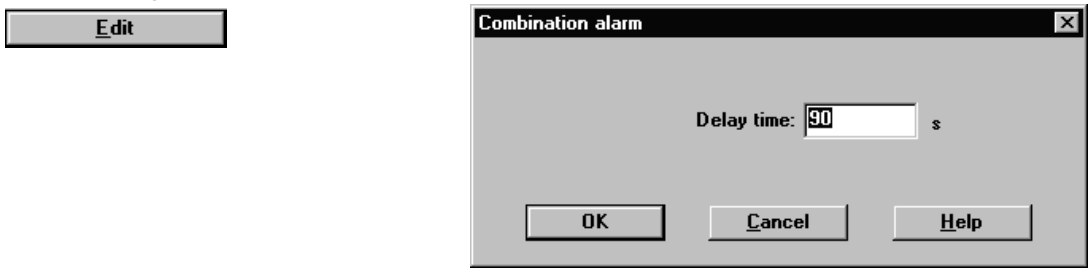
Parameter	Selection/settings	Description
Set time [SetTime]	00:00:00 — 23.59.59 Date	Current date and time There is a choice of transmitting the date, the day of the week, and the time on the system clock of the PC.
Set weekday [SetDay]	Monday — Sunday Sunday	Current weekday
Summer time start [SumTimeSt]	00:00:00 — 23.59.59 Date 02:00:00 / 29.03.1998	Start of summer time
Summer time end [SumTimeEnd]	00:00:00 — 23.59.59 Date 03:00:00 / 25.10.1998	End of summer time
Switch over [Switchover]	Off Manual Automatic	Manual: switch over of summer time at the time given Automatic: Switch over of summer time at previously calculated times (internal table to 2099).

= factory setting [ ] = short name in the operating unit

5.3 Combination alarm

The delay time for the combination alarm is set here.

Setup dialog



Parameters

Parameter	Selection/settings	Description
Delay time [Delay]	0 — 255s 90s	The combination alarm can be delayed by the time which can be set (+18sec normal delay).

= factory setting [ ] = short name in the operating unit

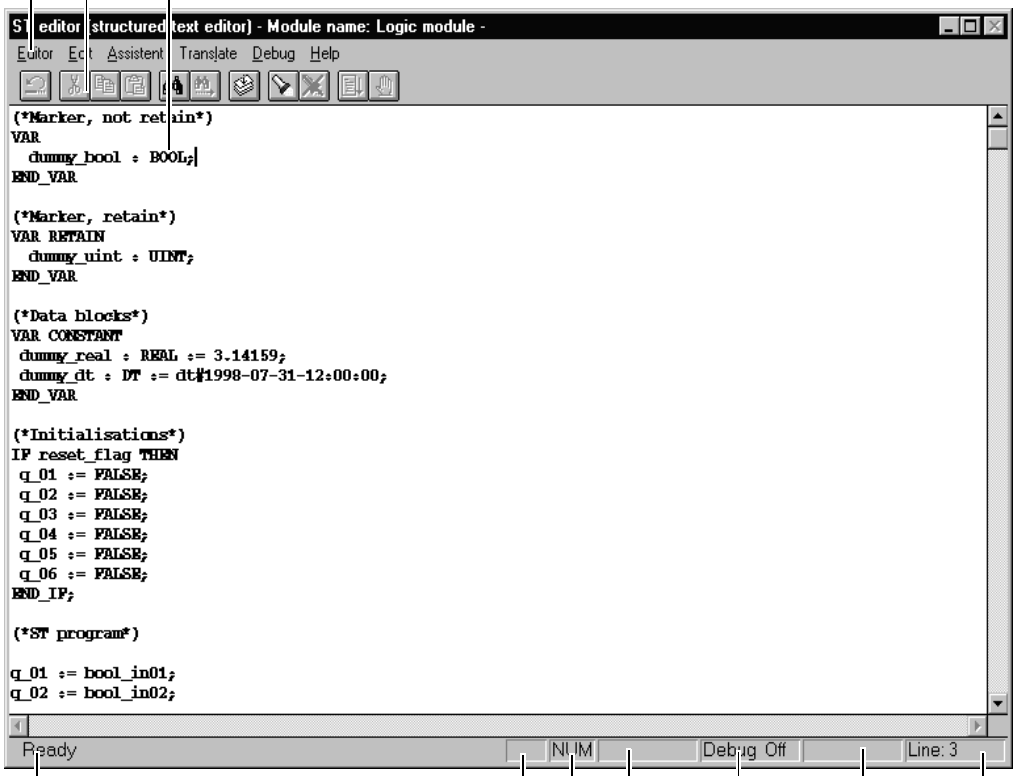
5.4 ST editor

The ST editor is available for creating and compiling programs.

Menu bar
Commands for editing or compiling programs

Symbol bar
Important functions can be reached directly by a mouse click through the symbols in the symbol bar

Window for editing
The program code is shown and edited here



CAPS lock

NUM lock

Force
Shows “force” if one or several inputs are forced

Debug status

Status

Program status
“Prg. stop” means that program is stopped
“Prg. run” means that program is running

Program line
in which the curser is positioned

Symbol bar

Undo last entry
Edit → *Undo last entry*
Undoes the last entry

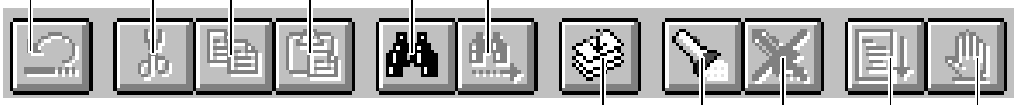
Cut
Edit → *Cut*
Removes the text which was marked and stores it in the clipboard

Copy
Edit → *Copy*
Copies the text which was marked to the clipboard

Insert
Edit → *Insert*
Inserts a text from the clipboard at the cursor position

Search
Edit → *Search*
Searches for character sequences in the program

Continue search
Edit → *Continue search*
Searches for further matches of character strings



Compile
Compiles the program

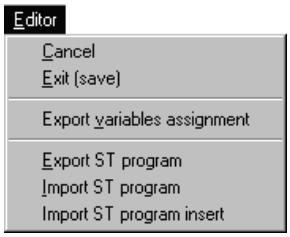
Start debug
Debug → *Start debug*
Starts the debugger

End debug
Debug → *End debug*
Ends the debugger

Program restart
Debug → *Program restart*
Starts the program in the debugger

Program stop
Debug → *Program stop*
Stops the program in the debugger

5.4.1 Opening and storing programs

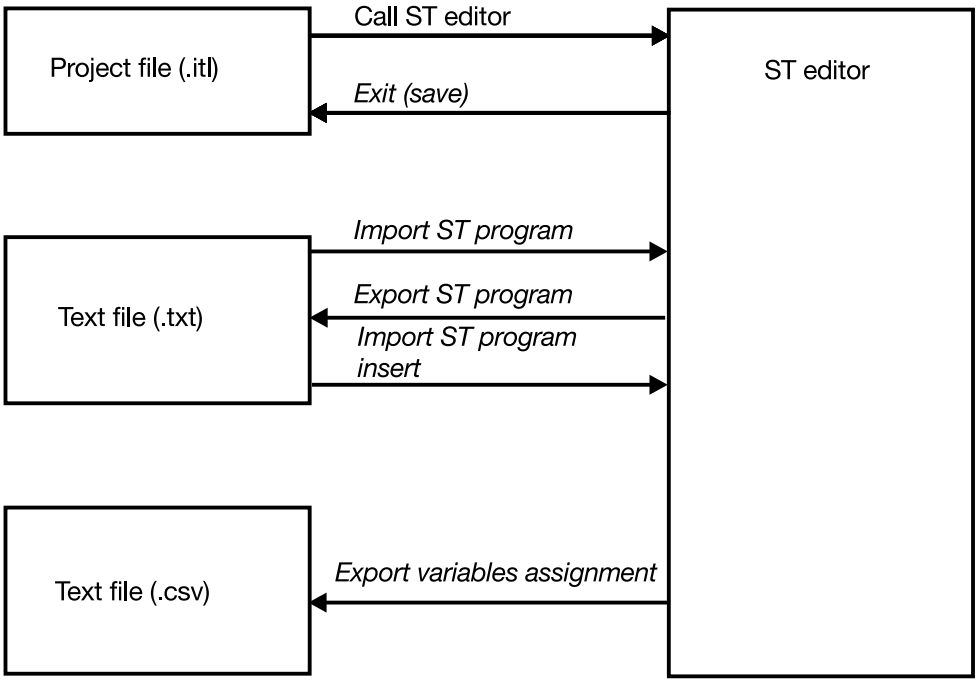


When the ST editor is called up, the program stored in the project will be loaded automatically.

Other programs can be loaded via the *Editor* → *Import ST program* function. *Editor* → *Insert ST program* inserts a program, or a program part, at the cursor position into the program in the editor.

The *Editor* → *Exit (Save)* terminates the ST editor and stores the program in the project file.

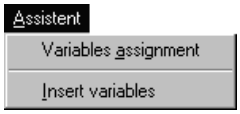
Furthermore, the program in the ST editor is able to store the assignment of variables as a text file. → Section 5.4.3 “Assistant”



5.4.2 Edit programs

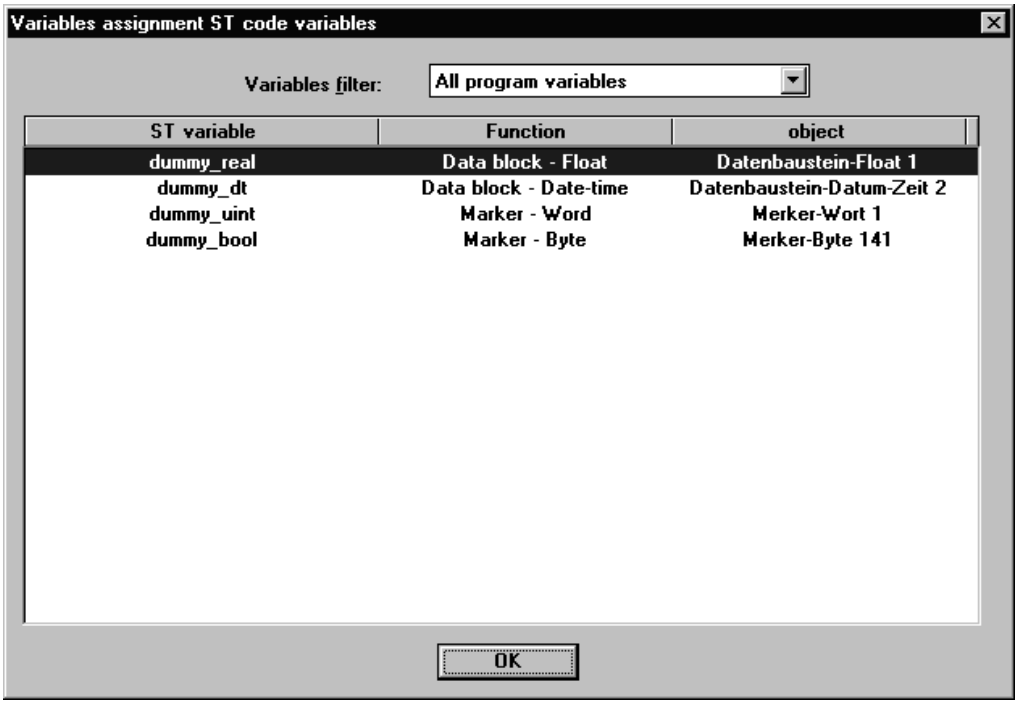
Standard Windows commands are available for program editing in the ST editor.

5.4.3 Assistant



Assignment of variables

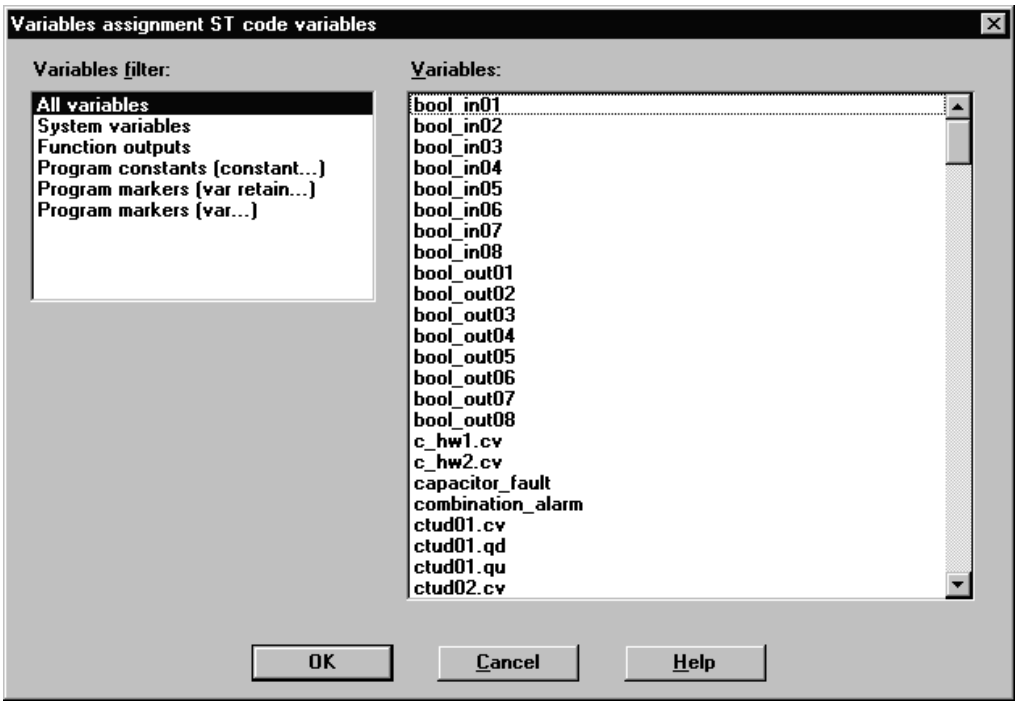
The assignment of variables to functions and objects can be indicated via *Assistant* ➔ *assignment of variables*.



☞ The Object column contains information for PLC applications via the JUMO mTRON communication module.

Insert variables

The variable definitions can be transferred directly from a list to the program in the ST editor.



* Position the cursor at the point where the variable is to be inserted
* The window shows *Assistant* → *Insert variables*
* Select variable
* Confirm with *OK*

5.4.4 Compile

The ST code for the logic module is compiled in a program code.
Any errors will be displayed.

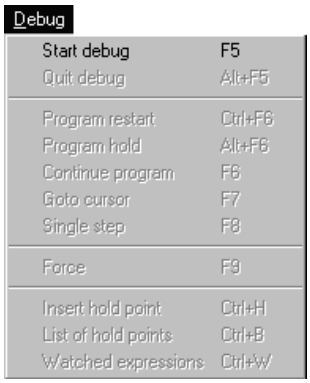
The program is either loaded into the logic module and started via the download function of the iTOOL, or it is loaded through the debugger.

A downloading is only possible if the program in the project is free of faults.

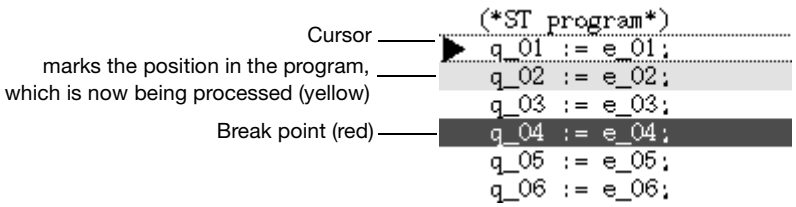
5.4.5 Debugger

The debugger is used to test the program.

In order to be able to start the debugger, the project must be online.
When the debugger is called, the program is automatically compiled and transferred to the logic module.
All “retain” variables from the previous ST program in the logic module will be deleted.
The setup connector must not be plugged into the module which is to be debugged. A second module is always required for communication.



Markers in the program text



Break point

You can use þ-H to insert break points into the program. The program run will be stopped at this point.

* Continue the program with F6, F7 or F8

Delete break point

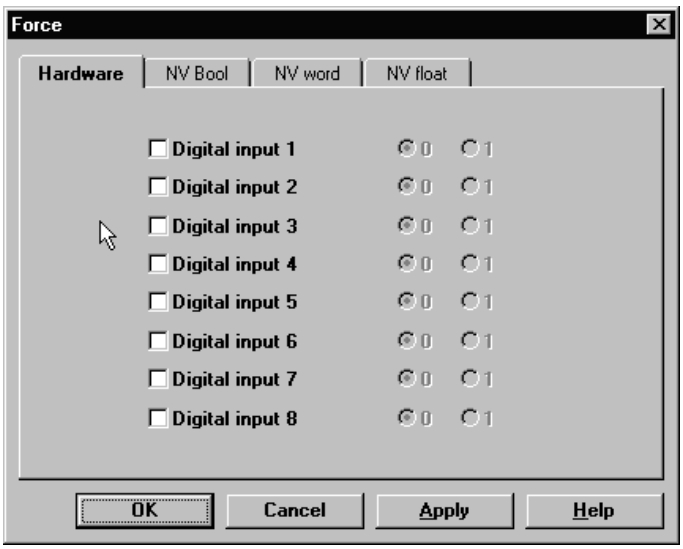
* Position the cursor on the break point
* Press þ-H

Program stop

* Press A-F6
* Restart the program with þ-F6

Force

The states and values of the inputs (network variables and hardware inputs) can be defined during the debugging. Forced inputs keep the defined state.



- Set the states and values
- Transfer the setttings to the logic module by using *Apply*
  (the window remains open; the settings are only valid as long as the window is open)
- Confirm the settings with *OK*
  (the window is closed; the settings remain valid)

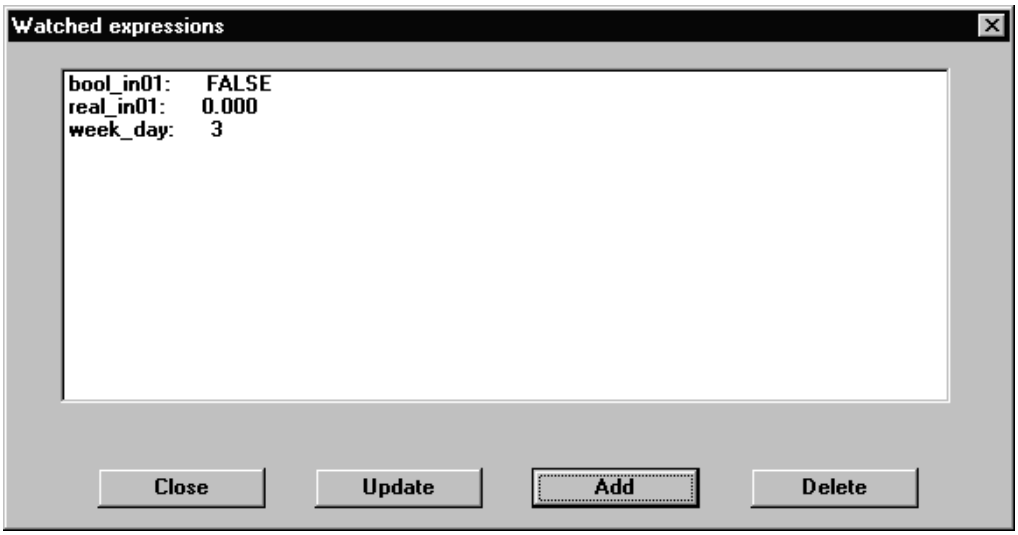
or

- Break off the entry with *Cancel*
  (the window is closed; the status which existes before calling the *Force* command is re-established)

The status line of the ST editor shows whether any inputs are forced.

Monitored expressions

The states and values of the variables and functions can be monitored in a window.



* Insert variables with *Add*

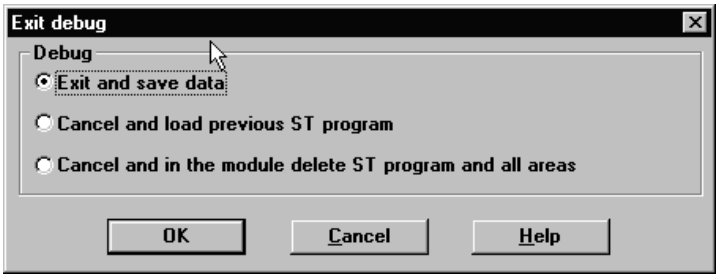
A window is opened for the assignment of the variables

* Mark the desired variable
* Confirm with *OK*
* Display the states and values with *Update*
* End with *Close*

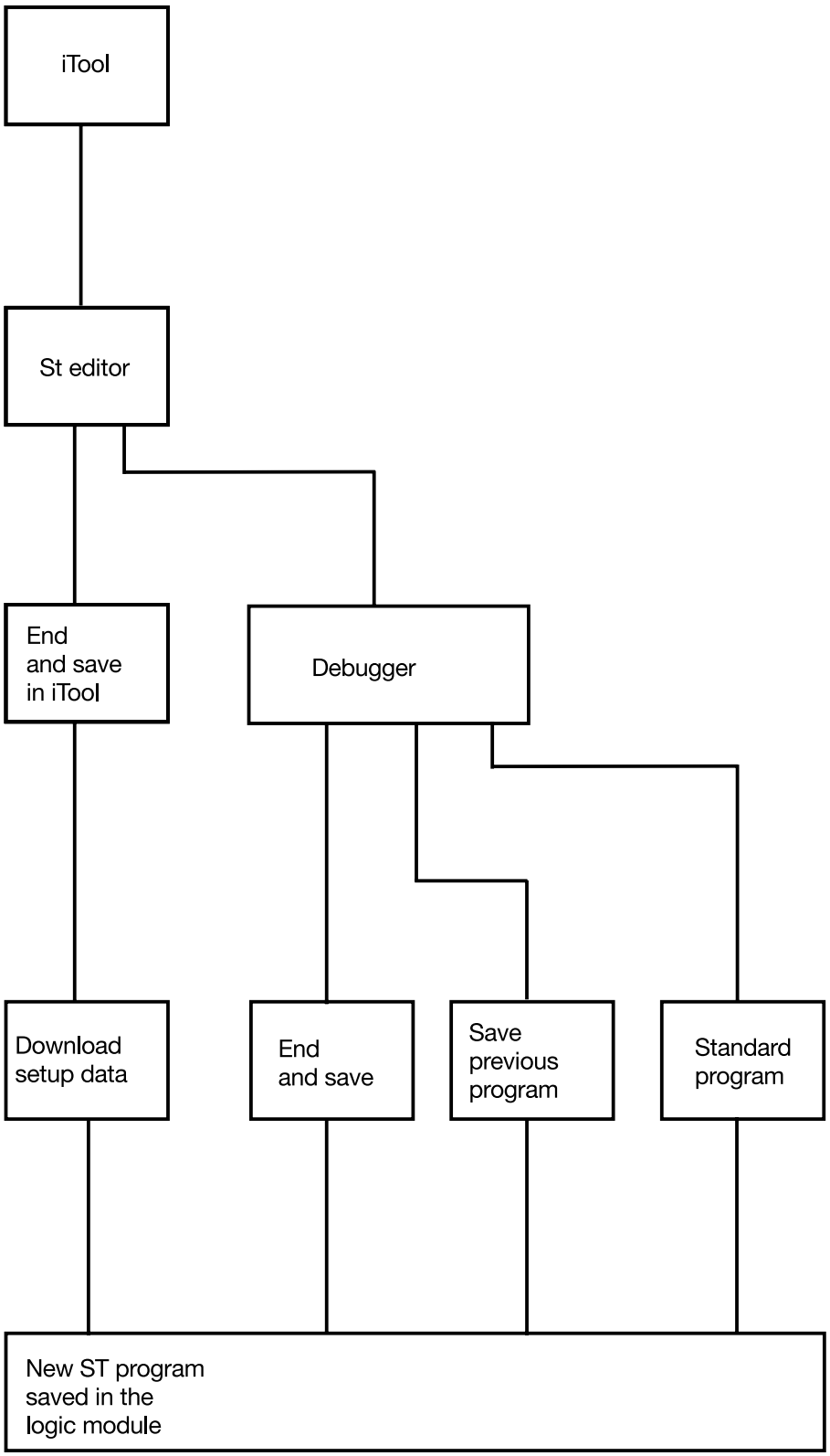
This function cannot be called while the program is running. The number of variables which can be displayed depends on their type (min.7, max. 30 bytes). Variables of “Real” type up to ±100000 are displayed with 3 decimal places (the internal calculation precision is higher), above that with scientific (exponential) notation.

End Debugger

There are three ways of ending the debugger:



* End and save data to logic module.
  The ST program which was processed is stored in the logic module and started
* Cancel and load previous program.
  The previous ST program, which was stored in the logic module before calling the debugger, is reloaded and started
* Cancel and delete all areas in the ST program module.
  The standard (default) program is loaded and started.


iTool
St editor
End
and save
in iTool
Debugger
Download
setup data
End
and save
Save
previous
program
Standard
program
New ST program
saved in the
logic module

6.1 General notes

This chapter describes the programming of the logic module.

The chapter is divided into the following sections:

- Program structure
- Data types
- Operators
- Selection instructions
- Iterations (repeated instructions)
- Functions
- Function blocks
- System variables

The programming of the logic module is carried out according to DIN 1131, Part 3 “Structured Text” (equivalent to IEC 65B(CO)85). The “Tools” which the logic module provides for programming are described in this manual. The decription in this system manual is intended for persons with programming know-how (preferably in Pascal). For detailed information on programming according to a “structured text”, please see DIN 1131, Part 3.

6.2 Program structure

Syntax

The syntax of the programming language “Structured Text” is similar to that of Pascal. The structure of the programs is also similar to Pascal.

Each program consists of two blocks:

- Variable declarations
- Program body (including the initialisation block)

Variables

Variables are labels which stand for specific values. The names of the variables can be freely chosen, observing the following rules:

1. The names can be made up from the characters A — Z, a — z, the numbers 0 — 9 and the understroke “_”
2. German special characters [ä, ö, ü, ß] are not allowed
3. The first character must be a letter
4. The name can be max. 19 characters long
5. Keywords such as UINT or REAL, system variables, function names and function block names must not be used as names of program variables.

Upper/lower case letters

No distinction is made between capital and normal letters.

Example: “NUMBER” and “number” are the same variable.

Assignment

Variables of data type UINT und REAL are assigned as decimal.

Example: a:= 123456; but **not** a:= 0x1E240;

Variables of data type BOOL are assigned the keywords TRUE or FALSE.
Example: a: = TRUE.

If an integer variable is assigned to a boolean variable, the integer variable must first be converted into a boolean variable, by using the function INT_TO_BOOL.

Variable declaration

Each program can use variables which are either

- “not retain”
- “retain”

or

- “constant”

Not retain

Variables which are declared as “not retain” lose their current value if the module is disconnected from the supply voltage. In other words, the current value of a “not retain” variable is **not** buffered. When the supply voltage is reconnected, the value of a “not retain” variable is set to 0 or FALSE (if it is a boolean variable).

Retain

Variables which are declared as “retain” keep their current value if the module is disconnected from the supply voltage. In other words, the current value of a “not retain” variable is buffered (for at least 8 days) and it keeps the same value when the supply voltage is reconnected.

Constant

Variables which are declared as “constant” keep their current value if the module is disconnected from the supply voltage. This means that the value of the “constant” variable is the same when the supply voltage is reconnected as it was before the module was disconnected. The value of a variable of type “constant” cannot be altered while the program is running.

Program body

The body of the program is the program proper. The program is continuously executed from top to bottom. It consists of various instructions:

- Assignments
- Commands for functions and function blocks
- Select instructions
- Repeat instructions

and

- Operations (logical, bitwise, arithmetic, comparison, selection)

Program example

Variable declarations

```
(*Marker,not retain*)
VAR
  dummy_bool : BOOL;
END_VAR

(*Marker, retain*)
VAR RETAIN
  dummy_uint : REAL;
END_VAR

(*Data block*)
VAR CONSTANT
 dummy_real : REAL := 3.14159;
 dummy_dt : DT := dt#1998-01-01-12:00:00;
END_VAR
```

Program body

```
(*Initialisations*)
IF reset_flag THEN
 q_01 := FALSE;
 q_02 := FALSE;
 q_03 := FALSE;
 q_04 := FALSE;
 q_05 := FALSE;
 q_06 := FALSE;
END_IF;

(*GoldCap capacitor discharged OR combination alarm active*)
IF Capacitor_Fault OR Combination_Alarm THEN
 q_01 := FALSE;
 q_02 := FALSE;
 q_03 := FALSE;
 q_04 := FALSE;
 q_05 := FALSE;
 q_06 := FALSE;
ELSE;
(*ST program*)
 q_01 := e_01;
 q_02 := e_02;
 q_03 := e_03;
 q_04 := e_04;
 q_05 := e_05;
 q_06 := e_06;
END_IF;
```

The following must be observed for installations with completed project design:

- If variables are inserted later, then they must always be inserted at the end of the corresponding variables list, just before the keyword “END_VAR”. Failing to do so will cause a shift of the addresses, resulting in faulty operation due to wrong assignment.
- Existing variables must not be deleted.

6.3 Program execution

On starting, the program is run once, completely, and then the body of the program is automatically and continuously repeated.

At each run through the body of the program:

- the hardware inputs are read in (de-bounced)
- the hardware outputs are set, and
- the input network variables are accepted.

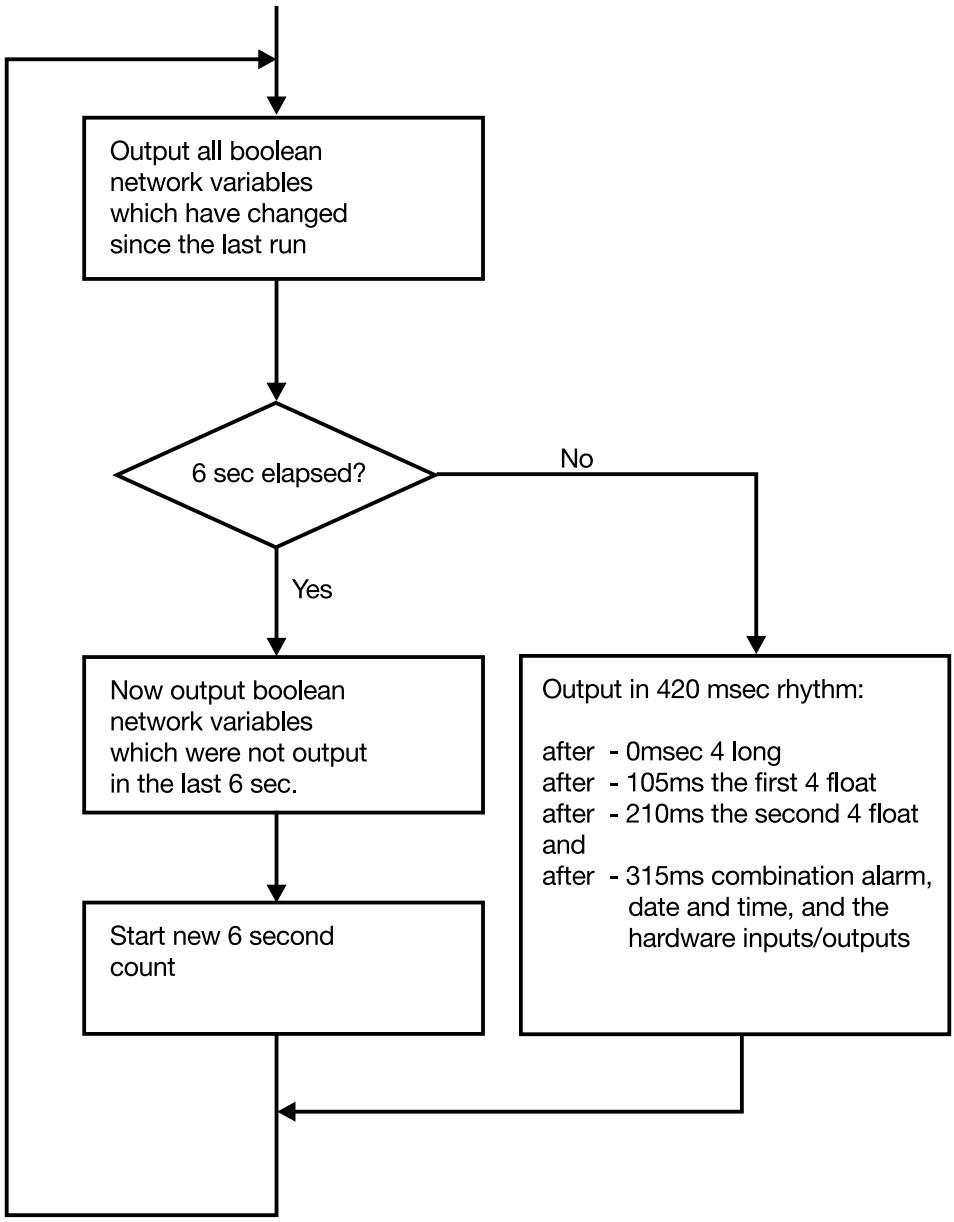
This means that the program body must **not** be included in an endless loop in order to repeat the execution of the program. An endless loop which is enclosed the program body would have the effect that the hardware inputs/outputs would never be read/operated and the input network variables would never be accepted.

Furthermore, this means that, for instance, a hardware output will not be operated at the same moment as it is set in the program

.

6.4 Updating the output network-variables

The output network-variables of the logic module are updated asynchronously (with reference to the execution of the program) according to a method which is shown in the following flow chart. The flow chart is run continuously, with a 105 msec clock cycle.



☞ This asynchronous updating of the output network-variables (with reference to the running of the program) means that the output network-variables are not altered at the same moment as the change of the corresponding variables in the program.

6.5 Data types

The logic module supports four different types of data:

- Boolean value
- Integer number
- Floating point number (float value)
- Date and time

Boolean value

Keyword	Value range
BOOL	TRUE (1) or FALSE (0)

Example of BOOL declaration

```
var                     (* Variable of type BOOL *)
            TuerK : bool;
end_var
var constant            (* Constant in data block *)
            Always : bool := TRUE;
            Never : bool := FALSE;
end_var
```

Example of BOOL assignment

```
TUERK := Always;      (* Caps/lower case permitted *)
TuerK := TuerK OR Always AND Never XOR false;
```

Integer number

Keyword	Value range
UINT	0 — 65535

Example of UINT declaration

```
var                     (* Variable of type UINT *)
            i, j : uint;
            count : uint;
end_var
var retain              (* Remanent marker *)
            OperatingHours : uint;
end_var                 (* Up to 19 characters per variable name *)
```

Example of UINT assignment

```
OperatingHours := OperatingHours + 1;
count := i + j * 14;
```

Floating point number

Keyword	Value range	smallest displayable value
REAL	± 1.0E±38	± 8.43E-37

Example of REAL declaration

```
var                    (* Variable of type REAL *)
            a, b : real;
end_var

var constant
            Skal : real := -13.0E10;
end_var
```

Example of REAL assignment

```
a := b/a - 3.0;
b := 47.8e+10 - a;
```

Date and time

Keywords

DT or DATE_AND_TIME

Example of DT declaration

```
var                    (* Variable of type DT *)
            present : dt;
            start : date_and_time;
end_var
```

Example of DT assignment

```
start := dt#1997-09-12-07:30:10;
present := date_and_time#1997-9-12-13:05:00;
Tuerk := start > present; (* Result Tuerk is BOOL *)
```

During operations there is no range monitoring for the data types.

Exception: - square root function (SQRT)
- reciprocal value (1/x)

6.6 Operators

The following table shows all the operators which are supported by the logic module. The table is arranged in the order of priority of the operators, starting with the highest priority.

Operation	Symbol	Permitted data types	Example
Brackets	(expression)		`a := 3.0*(b-1.0);`
Function	designator (argument list)		`i := min (3, j);`
Negation	-	REAL	`a := -a;`
Complement[1]	NOT	BOOL, UINT	`TuerK := not always;`
Multiplication	*	UINT, REAL	`i := 5 * j;`
Division	/		`a := 5.0 / b;`
Modulo	MOD	UINT	`J := i mod 10;`
Addition	+	REAL, UINT	`i := 5 + j;`
Subtraction	-		`a := b - 5.0e20;`
Comparison	<, >, <=, >=	REAL, UINT, DATE_AND_TIME	`TuerK := 5 <= j;`
Equality	=	REAL, UINT, BOOL, DATE_AND_TIME	`TuerK := 5 = i;`
Inequality	<>		`TuerK := 5 <> j;`
AND	& or AND	UINT[1], BOOL[1]	`TuerK := never & TuerK;` `TuerK := never AND TuerK;`
OR (exclusive)	XOR	UINT[1], BOOL[1]	`TuerK := true XOR TuerK;`
OR (inclusive)	OR	UINT[1], BOOL[1]	`TuerK := false OR TuerK;`

1. Boolean variables are combined logically.
   UINT variables are combined bitwise.

With addition (+) and subtraction (-) with one constant (e.g. 1), make sure that a space is inserted in front of the constant. If this is omitted, the operand will be interpreted as the sign of the constant.

Correct: `i := i - 1;`

Wrong: `i := i -1;`

6.7 Conditional instructions

A conditional instruction performs an instruction or a group of instructions according to a predetermined condition.

IF instruction

The IF instruction determines that a group of instructions will only be performed if the relevant boolean expression returns the value TRUE. If the condition is false, then either no instruction is performed, or the group of instructions which follows the keyword ELSE will be performed (or the keyword ELSIF, if the relevant boolean condition is fulfilled).

Keywords

IF, THEN, ELSIF, ELSE, END_IF

Example of IF instruction

```
VAR                          (*Marker, not retain*)
i : UINT;
END_VAR
IF reset_flag THEN
       q_01 := false;   (* after the reset, first of all
                           the q_01 outputs *)
       q_02 := false;   (* and q_02 are reset and *)
       i := 1;          (* i is set to 1 *)
END_IF;


if i>0 and i<=100 then
       q_01 := true;    (* in runs 1 to 100
                           the q_01 outputs are set *)
       q_02 := false;   (* and q_02 is reset *)
elsif i>100 and i<=200 then
       q_01 := false;   (* in runs 101 to 200
                           the q_01 outputs are reset *)
       q_02 := false;   (* and q_02 is set *)
else
       i := 1;          (* in run 201 ... *)
       q_01 := false;   (* the q_02 outputs are reset *)
                        (* and i is set to 1 *)
       q_02 := false;   (* it all starts again *)
end_if;


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

6.8 Iterations (repeat instructions)

Repeat instructions are used to repeatedly perform instructions and groups of instructions.

There are three types of iteration:

- FOR
- WHILE
- REPEAT UNTIL

EXIT

EXIT is used to stop an iteration before it has reached the end condition.
If EXIT is used within a nested repeat construction, then the innermost loop, within which EXIT is situated, is exited.

During iterations, care must be taken that no endless loops or very lengthy loops are created.

⇨ Section 6.2 “Program structure”

⇨ Section 6.3 “Program execution”

FOR instruction

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

Keywords

FOR, TO, BY, DO, END_FOR

Example of FOR instruction

```
FOR i := 10 TO 100 BY 10 DO (* running var. must be UINT *)
            j := j + i; (* i runs from 10 to 100 in *)
END_FOR;                (* steps of 10 (10,20,30…100) *)

For i := 1 TO 5 DO      (* without entry of “BY x” the run. *)
            a := a + b; (* var. is incr. by 1 (1,2,3,4,5) *)
            a := a * 3.0;

END_FOR;
```

WHILE instruction

The WHILE instruction has the effect, that a group of instructions is repeated until the relevant boolean expression becomes FALSE. If the boolean expression is false at the start, the group of instructions is not performed at all.

Keywords

WHILE, END_WHILE

Example of WHILE instruction

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

REPEAT instruction

The REPEAT instruction has the effect, that a group of instructions up to the keyword UNTIL is repeated (at least once) until the relevant boolean expression becomes TRUE.

Keywords

REPEAT, UNTIL, END_REPEAT

Example of REPEAT instruction

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

6.9 Functions

This section describes the functions which are supported by the logic module.

The logic module supports the following functions:

- Type conversion
- Arithmetic functions
- Bit sequence functions
- Logic functions
- Selection and comparison
- Elements of data type DATE_AND_TIME

6.9.1 Type conversion

The logic module supports the following type conversions:

- INT_TO_REAL
- INT_TO_BOOL
- REAL_TO_INT

INT_TO_REAL converts an INTEGER into a REAL value.

Example

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

INT_TO_BOOL converts an INTEGER into a BOOLEAN variable.

Example

```
a := INT_TO_BOOL(0); (* a = FALSE *)
b := INT_TO_BOOL(1); (* b = TRUE *)
c := INT_TO_BOOL(20); (* c = TRUE *)
```

REAL_TO_INT converts a REAL value into an INTEGER value.

Example

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

6.9.2 Arithmetic functions

The logic module supports the following arithmetic functions:

- Addition
- Subtraction
- Multiplication
- Division
- Negation of REAL numbers
- Modulo
- Square root

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

Example

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

Subtraction The result is formed by subtracting the second argument from the first.

Example

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

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

Example

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

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

Example

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

Comment The result of the division of integer numbers (data type UINT) is an integer number with the decimal places cut off (7/3 = 2).

Negation REAL numbers can be negated.

Example

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

Modulo The arguments of the modulo function (MOD) must be integer numbers (data type UINT). The result of the modulo function is the same as processing the following instruction:

```
IF (IN2 = 0)  THEN OUT := 0;
              ELSE OUT := IN1 - (IN1/IN2) * IN2;
END_IF
```

Comment The result of dividing the integer numbers (data type UINT) in the ELSE branch of the IF conditional instruction above is an integer with the decimal places cut off.

Example

```
IN1 := 17;
IN2 := 3;
OUT := IN1 MOD IN2;     (* OUT := 2 *)
```

SQRT (IN) calculates the square root of the REAL number IN. The result is a REAL number.

Example

```
OUT := SQRT(9.0); (* OUT := 3.0 *)
```

Comment If the argument (IN) is less than 0, the function returns the value 0. In addition, the boolean variable SQRT_FAULT := TRUE is set

⇨ Section 6.11 “System variables”

6.9.3 Bit-sequence and logic functions

Bit-sequence functions shift the row of bits in the argument to the left or right, filling up with zeros, or perform a cyclic rotation.

The logic module supports the following bit-sequence functions:

- SHL
- SHR
- ROL
- ROR

The logic module supports the following logic functions:

- AND
- OR
- XOR
- NOT

SHL (IN, n) Shifts the bit sequence of the argument IN to the left by n bits. The following spaces are filled from the right by zeros.

Example

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

SHR (IN, n) Shifts the bit sequence of the argument IN to the right by n bits. The preceding spaces are filled from the left by zeros.

Example

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

ROL (IN, n) Performs a cyclic rotation of the bit sequence IN to the left by n bits.

Example

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

ROR (IN, n) Performs a cyclic rotation of the bit sequence IN to the right by n bits.

Example

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

AND Boolean variables are added logically, UINT variables are added bitwise.

If there are more than two parameters, they are worked through in pairs from left to right.

Example logic combination:

```
X := TRUE;
Y := FALSE;
Z := X AND Y; (* z = FALSE *)
```

Bitwise combination

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

OR

Boolean variables are combined logically, UINT variables are combined bitwise.

If there are more than two parameters, they are worked through in pairs from left to right.

Example

logical combination:

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

bitwise combination

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

XOR

Boolean variables are combined logically, UINT variables are combined bitwise.

If there are more than two parameters, they are worked through in pairs from left to right.

Example

logical combination:

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

bitwise combination

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

NOT

Boolean variables are combined logically, UINT variables are combined bitwise.

Example

```
IN := 1;
OUT := NOT IN;   (* OUT = 0 *)
IN := 43690;     (* IN = 1010 1010 1010 1010 *)
OUT := NOT IN;   (* OUT = 21845; OUT = 0101 0101 0101 0101 *)
```

6.9.4 Selection and comparison

The logic module supports the following selection and comparison functions:

- Selection:
  - MIN
  - MAX
  - LIMIT
- Comparison:
  - GT (>)
  - GE (>=)
  - EQ (=)
  - LE (<=)
  - LT (<)
  - NE (<>)

Selection functions

MIN Returns the smallest argument as a result.

Example
```
OUT := MIN(8, 12);              (* OUT := 8 *)
```

MAX Returns the largest argument as a result.

Example
```
OUT := MAX(8, 12);              (* OUT := 12 *)
```

LIMIT Tests if a value (IN) is within a certain range (MIN, MAX).
If the value is within the range, the value itself will be returned. If the value is below the range, the lower (minimum) limit will be returned. If the value is above the range, the upper (maximum) value is returned.

Example
```
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 *)
```

Comparison functions

Data types The following data types are supported by the particular functions:

Comparison	Supported data types
EQ, NE	all data types
GT, GE, LT, LE	all data types except BOOL

GT (>) Compares arguments to see if one is larger than the other. The result is a boolean value.

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

GE (>=) Compares arguments to see if one is the same as or larger than the other. The result is a boolean value.

Example
```
OUT := IN1 >= IN2;    (* OUT := TRUE,
                          when IN1 is larger than or equal to IN2 *)
```

EQ (=) Compares arguments to see if they are equal. The result is a boolean value.

Example
```
OUT := IN1 = IN2;
```

LE (<=) Compares arguments to see if one is the same as or smaller than the other. The result is a boolean value.

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

LT (<) Compares arguments to see if one is smaller than the other. The result is a boolean value.

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

NE (<>) Compares arguments to see if they are unequal. The result is a boolean value.

Example
```
OUT := IN1 <> IN2;    (* OUT := TRUE,
                            when IN1 is not equal to IN2 *)
```

6.9.5 Elements of data type DATE_AND_TIME (DT)

The following elements of data type DT can be individually requested:

- Year
- Month
- Day
- Hour
- Minute
- Second

The return value is always of data type UINT.

GET_YEAR (DT) Returns the year (with century).

Example
```
OUT:= GET_YEAR (dt#1998-03-20-06:45:12);
```
OUT = 1998

GET_MONTH (DT) Returns the month.

Example
```
OUT:= GET_MONTH (dt#1998-03-20-06:45:12);
```
OUT = 3

GET_DAY (DT) Returns the day of the month as a number.

Example
```
OUT:= GET_DAY (dt#1998-03-20-06:45:12);
```
OUT = 20

⇨ Section 6.11 “System variables”, System variable Week_Day

GET_HOUR (DT) Returns the hour.

Example
```
OUT:= GET_HOUR (dt#1998-03-20-17:45:12);
```
OUT = 17

GET_MINUTE (DT) Returns the minute.

Example
```
OUT:= GET_MINUTE (dt#1998-03-20-06:45:12);
```
OUT = 45

GET_SECOND (DT) Returns the second.

Example
```
OUT:= GET_SECOND (dt#1998-03-20-06:45:12);
```
OUT = 12

Example of data type DATE_AND_TIME

```
VAR                                    (* Marker, not retain *)
       year      : UINT;
       month     : UINT;
       day       : UINT;
       hour      : UINT;
       minute    : UINT;
       second    : UINT;
       weekday    : UINT;
END_VAR


weekday := week_day;
year := get_year(rtc.cdt);
month := get_month(rtc.cdt);
day := get_day(rtc.cdt);
hour := get_hour(rtc.cdt);
minute := get_minute(rtc.cdt);
second :=get_second(rtc.cdt),


(* from Mo-Th 13:00:00 to 13:00:09 pulls in relay 6 *)
if (weekday=0 OR weekday=1 OR weekday=2 OR weekday=3)
    AND hour=13 AND minute=0 AND second<10 then
       q_06 := TRUE;        (* weekday=0 means Monday *)
                            (* weekday=6 means Sunday *)
else
       q_06 := FALSE;
end_if;
```

6.10 Function blocks

Instance

A function block is a unit of the program organisation which can return one or several values when it is run. A function block can exist in several instances (copies). Each one of these instances possesses a corresponding designator. The instances of a function block are independent of each other.

For instance, if a function block called TON exists in 8 instances, then the individual instances are called as TON1, TON2, ... to TON8.

The logic module provides the following function blocks:

Designator	Instances	Comment
CTUD	32	Software up/down counter
RTC	1	Real-time clock
TP	8	Pulse generator
TON	8	Switch-on delay
TOF	8	Switch-off delay
R_TRIG	8	Rising edge recognition (0 -> 1)
F_TRIG	8	Falling edge recognition (1 -> 0)
SR	8	Bistable function block (biased to set)
RS	8	Bistable function block (biased to reset)
C_HW	2	Hardware counter

Parameter

One or more parameters (inputs) are passed to the function blocks. The parameters are placed in round brackets after the designation of the instance. The sequence of the parameters is fixed. Every call of a function block is terminated by a “ ; ”.

Example

TON1(TRUE,5,1);

Inputs/outputs

All the states of the inputs and outputs of all instances of the function blocks always remain constant until the next call of the particular instance of the function block. They can be requested and be freely used in operations.

Access to outputs

Outputs (structure elements) of function blocks can be accessed in the same way as normally used for the Pascal programming language.

Example

Request the output ET of the function block TON (instance 1):

```
OUT := TON1.ET;
```

6.10.1 Software up/down counter

The counter provides the following functions:

- Up counting
- Down counting
- Reset the count value to 0
- Set an upper count value
- Request value if the upper count value is exceeded
- Request if count value = 0
- Request count value

Call

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

Inputs

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

Parameter	Data type	Description
<instance>		01 … 32 instances 01 to 09 written as two figures
R	BOOL	set count value to 0
LD	BOOL	TRUE = set upper count value
CU	BOOL	TRUE = count up
CD	BOOL	TRUE = count down
PV	UINT	upper count value

Outputs

Parameter	Data type	Request/description
CV	UINT	`OUT := CTUD <Instance>.CV` OUT = count value
QU	BOOL	`OUT := CTUD <Instance>.QU` OUT = TRUE if count > upper count value
QD	BOOL	`OUT := CTUD <Instance>.QD` OUT = TRUE, if count value <= 0

Comment

The first 16 instances (01 — 16) of these function blocks are “retain”, i.e. the values of the inputs/outputs are held for up to 8 days if the logic module is disconnected from the supply voltage.

Example

```
VAR
            count_value : uint;
END_VAR
VAR RETAIN                          (* variable remains
                                       if supply is interrupted *)
            count_value: uint;      (* for counter *)
END_VAR
if reset_flag then
            q_01 := FALSE;
            q_01 := FALSE;
            q_03 := FALSE;

if capacitor_fault then              (* if data buffering
                                       not OK *)
            count_high := 0;         (* then defined
                                       reset *)
            end_if;
end_if;

count_value := 20;                   (* assignment *)
                                     (* count value 20 is
                                       signalled by output Q *)
                                     (* reset counter *)
CTUD16 (FALSE, FALSE, e_08, FALSE, count_value);
if e_08 then
            count_high := 0;         (* reset
                                       count variable *)
end_if;

R_TRIG7(e_07);                      (* increment counter by 1
                                       on each activation *)
            q_06 := R_TRIG7.Q;      (* edge triggering signalled
                                       by output 6 *)

if R_TRIG7.Q then
            CTUD16 (TRUE, FALSE, FALSE, FALSE, count_value);
            count_high := CTUD16.CV;
end_if;
            q_04 := CTUD16.QU;      (* output 4 ON, when
                                       counter reaches 20 *)
(* additional evaluation *)
if          count_high = 5 then q_01 := TRUE;
            elsif count_high = 10 then q_02 := TRUE;
            elsif count_high = 15 then q_03 := TRUE;
            else q_01 := FALSE;
end_if;
```

6.10.2 Real-time clock

The logic module has a real-time clock which provides the following functions:

- set real-time clock
- read real-time clock

Call

RTC (EN, PDT);

Inputs

Parameter	Data type	Description
EN	BOOL	rising edge on EN loads preset date and time (PDT)
PDT	DT	date and time which are loaded into the real-time clock when EN = 0 -> 1

Outputs

Parameter	Data type	Request/description
Q	BOOL	`OUT := RTC.Q;` OUT = copy of EN
CDT	DT	`OUT := RTC.CDT;` OUT = present date and time

Comment

Date and time are only valid when Q = TRUE.

As described above, a 0/1 edge is necessary on input EN to set the real-time clock. In practice this can be implemented by calling the RTC function block twice.

```
RTC (FALSE, date and time);
RTC (TRUE, date and time);
```

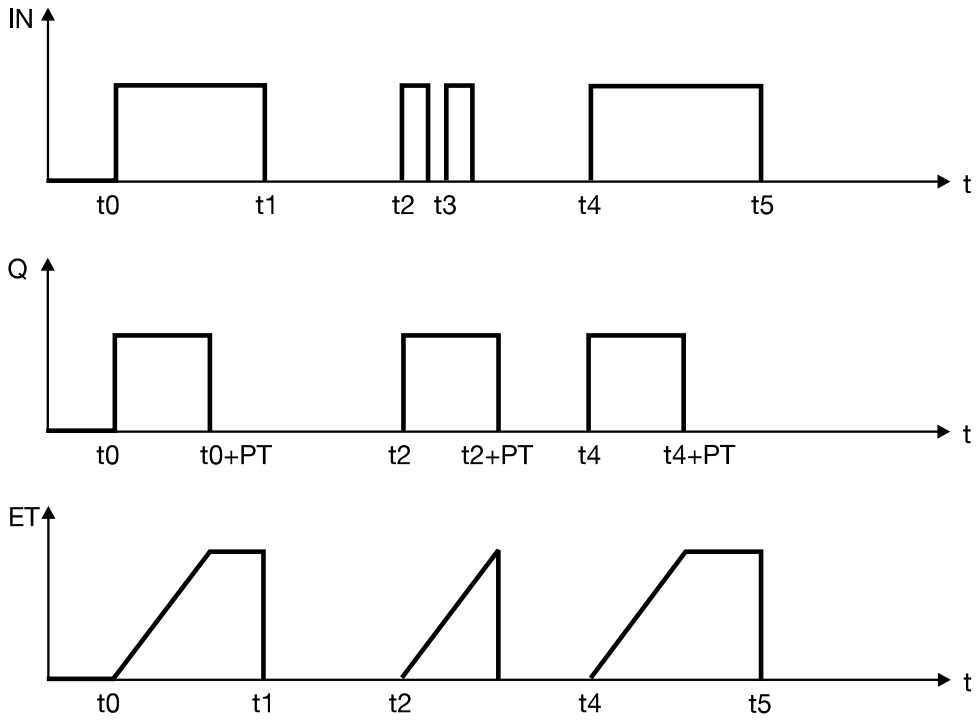
If the supply voltage is disconnected, the real-time clock is buffered for up to 8 days.

Example

```
VAR
                (* marker, not retain *)
                ramp : UINT;
END_VAR

VAR RETAIN
                (* marker, retain *)
                j : UINT;
END_VAR

VAR CONSTANT    (* data block *)
     start_holiday: dt := dt#1997-12-15-7:0:0;
     stop_time   : dt := dt#1997-12-15-7:0:30;
END_VAR

IF reset_flag THEN
                (* initialisation block *)
     RTC(FALSE, start_holiday);
                (* rising edge
                in RTC function block: *)
     RTC(TRUE, start_holiday);
                (* accept date and time *)
     q_01 := FALSE;
END_IF;

IF NOT q_01 THEN(* time measurement still active ?
                (until q_01=TRUE) *)
     RTC(TRUE, start_holiday);
                (* cyclic call of RTC function block *)
                (* date and time are only continued *)
                (* by the RTC HW-block, since no *)
                (* edge transition 0 -> 1 in first parameter *)
                (* of SW function block occurs *)
     IF RTC.Q THEN
                (* as per Standard 1131: time and date only *)
                (* valid if function block output Q = TRUE *)
          IF RTC.CDT > stop_time THEN (* 30sec elapsed? *)
                q_01 := TRUE (* yes: HW output 1 indicates
                                end of time measurement *)
          END_IF;
     END_IF;
END_IF;
```

6.10.3 Pulse generator

The method of operation of the pulse generator is illustrated in the following diagram:



Call

TP <Instance> (IN, PT, TB);

Inputs

Parameter	Data type	Description
<Instance>		1 … 8
IN	BOOL	IN = TRUE sets TP.Q for the time PT to TRUE
PT	UINT	time (unit Time_Base (TB)), for which Q = TRUE when IN = TRUE
TB	UINT	unit of PT: 1 = msec 2 = sec 3 = min

Output

Parameter	Data type	Request/description
Q	BOOL	`OUT := TP < Instance >.Q` OUT = TRUE for the time PT when IN := 0 -> 1
ET	UINT	`OUT := TP < Instance >.ET` OUT = time elapsed since start of the active pulse phase

Comment

The parameter Time_Base (TB) is an extension of DIN 1131, Part 3. This extension does not signify any restriction compared with the DIN. The time entry is, according to DIN “implementation-dependent”.

The maximum error in the function block is 210 msec.

Values less than 105 msec for PT do not make sense.

⇨ Section 6.3 “Program execution”

Example

```
VAR
     (* marker, not retain *)
     pulse : UINT;
     ramp : UINT;
END_VAR

IF reset_flag THEN    (* initialisation block *)
     q_01 := FALSE;
     pulse := 3;
END_IF;

IF e_07 AND e_08 THEN (* When HW inputs 7 and 8, then: *)
     TP7(e_01, pulse,3);(* pulse gen. 7 with 3 min. duration *)
     q_01 := TP7.Q;    (* HW output 1 indicates outputs
                          of the function block *)
     ramp := TP7.ET    (* time elapsed in min. since *)
                       (* start of active pulse phase *)
END_IF;
```

Retain instances

The first four instances of this function block are “retain”, i.e. the values of the inputs/outputs are held for up to 8 days after the supply voltage has been disconnected from the logic module. The DIN does not require this “retain” function.

After reconnecting the supply voltage, the values can be read, but the functionality of the function block (continuing to run automatically) is only available after an initialisation of the function block.

The following aspects must be observed for the initialisation of the function block:

- the reduced entry for PT
- the time which has elapsed during the disconnection of the logic module from the supply voltage

The following two program examples show a possible initialisation of the function block after reconnecting the supply voltage:

The examples start on the following page, for clarity!

Example

```
(* Implementation of a “retain” pulse generator, i.e. a pulse
   generator which, after a mains supply interruption,
   continues to run for the remainder of the set period.
   The duration of the supply interruption is NOT included *)

VAR           (* marker, not retain *)
PulseDuration: UINT;
END_VAR

VAR  RETAIN    (* marker, retain *)
     RestTime: UINT;
END_VAR

VAR  CONSTANT                (* data block *)
     millisecond: UINT := 1; (* timebase in msec *)
     second    : UINT := 2;  (* timebase in seconds *)
     minute    : UINT := 3;  (* timebase in minutes *)
     TotalTime : UINT := 60; (* duration of pulse phase *)
END_VAR

IF reset_flag THEN      (* initialisation block: new start *)
     IF RestTime > 0 THEN    (* pulse phase continued
                                after PowerOff? *)
                PulseDuration := RestTime;
                             (* yes: with remaining time *)
                TP2 (FALSE, TP2.ET, second);
                             (* cancel last pulse phase *)[1]
                TP2 (TRUE, PulseDuration, second);
                             (* High level for pulse gen. *)
     ELSE PulseDuration := TotalTime;(* no: next pulse phase
                                        starts fresh *)
     END_IF;
END_IF;

TP2 (e_01, PulseDuration, second);(* e_01: key for start of
                                     pulse phase *)
IF TP2.Q THEN                (* pulse phase *)
           RestTime := PulseDuration - TP2.ET;(* remaining time
                                                 of pulse phase *)
           q_01 := TRUE;     (* q_01: signals pulse phase *)
ELSE RestTime := 0;      (* no pulse phase *)
           q_01 := FALSE;    (* q_01: signals no pulse phase *)
           PulseDuration := TotalTime;(* next pulse phase
                                         starts fresh *)
END_IF;
```

1. To continue a pulse phase of a pulse generator which has already started, after a supply interruption, it is mandatory to cancel the unfinished pulse phase in the initialisation block after reconnecting the supply voltage.
   This is done by applying a LOW level (FALSE) to input IN and entering the present time value (TPx.ET) at the PT input of the pulse generator.

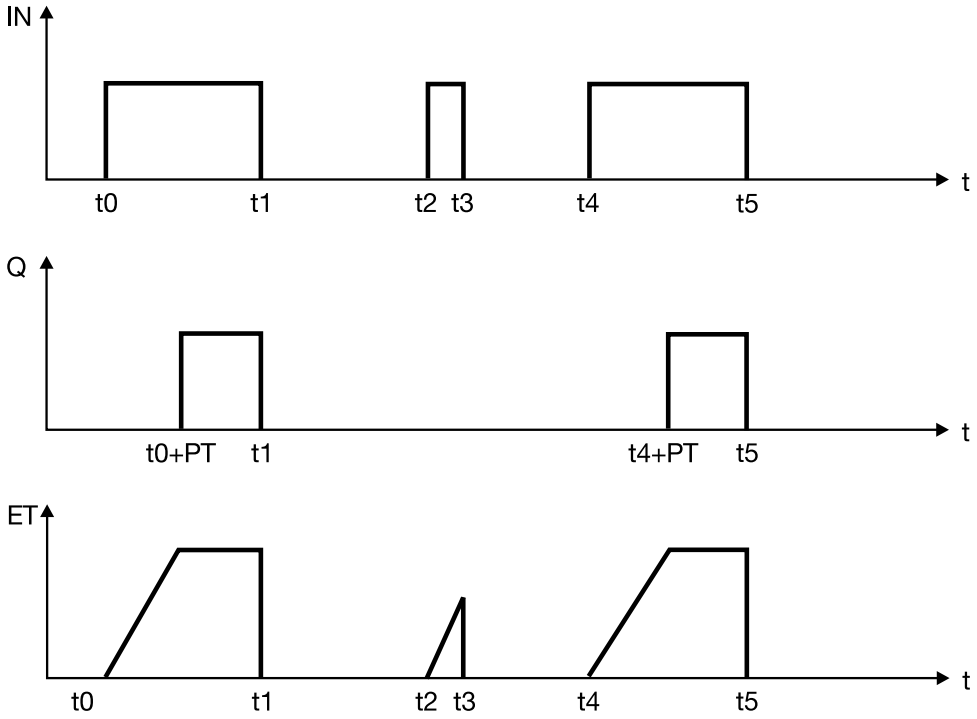
Example

```
(* Implementation of a “retain” pulse generator, i.e. a pulse
   generator which, after a mains supply interruption,
   continues to run for the remainder of the set period. In
   addition, the duration of the supply disconnection is taken
   into account. For clarity, only a power-off time of up to
   1 hour has been considered. *)

VAR    (* marker, nicht retain *)
  PulseDuration: UINT;
  PowerOffDuration : UINT;
END_VAR

VAR RETAIN  (* marker, retain *)
  RestTime: UINT;
  TimePowerOff : DT;
  TimePowerOn : DT;
END_VAR

VAR CONSTANT(* data block *)
  millisecond: UINT :=   1;    (* timebase msec *)
  second: UINT :=   2;         (* timebase seconds *)
  minute: UINT :=   3;         (* timebase minutes *)
  TotalTime: UINT := 120;      (* duration of pulse phase *)
END_VAR

IF reset_flag THEN   (* initialisation block: new start *)
  IF RestTime > 0 THEN(* continue pulse phase after PowerOff? *)
       TP2 (FALSE, TP2.ET, second);(* cancel last pulse ph.? *)
       TimePowerOn := RTC.CDT;(* time of PowerOn *)
       (* following 5 lines: PowerOff time in seconds *)
       IF GET_MINUTE (TimePowerOn) >= GET_MINUTE (TimePowerOff)
THEN
            PowerOffDuration := (GET_MINUTE (TimePowerOn) -
                                 GET_MINUTE (TimePowerOff)) * 60;
       ELSE
            PowerOffDuration := ((GET_MINUTE (TimePowerOn)+60)
                                - GET_MINUTE (TimePowerOff))* 60;
       END_IF;
       PowerOffDuration := PowerOffDuration + GET_SECOND
                           (TimepowerOn)
                           -GET_SECOND (TimePowerOff);
       IF PowerOffDuration >= RestTime THEN
       (* PowerOff duration is too long, do not ... *)
            PulseDuration := 0;(* ... continue pulse gen. *)
       ELSE
            PulseDuration := RestTime - PowerOffDuration;
            (* Pulse gen. less PowerOff duration ... *)
            TP2 (TRUE, PulseDuration, second);
            (* ... continue *)
       END_IF;
  ELSE
       PulseDuration := TotalTime;(* next pulse phase
                                     starts fresh *)
  END_IF;
END_IF;
```

```
TP2 (e_01, PulseDuration, second);
(* e_01: key to start a pulse phase *)
IF TP2.Q THEN(* pulse phase *)
   RestTime := PulseDuration - TP2.ET;
   (* remaining time of the pulse phase *)
   q_01 := TRUE;(* q_01: signals pulse phase *)
   TimePowerOff := RTC.CDT;(* note time, in case PowerOff
                              follows *)
ELSE
   RestTime := 0;(* no pulse phase *)
   q_01 := FALSE;(* q_01: signals no pulse phase *)
   PulseDuration := TotalTime;(* next pulse phase starts fresh *)
END_IF;
```

6.10.4 Switch-on delay

The method of operation of the switch-on delay is shown in the following diagram:



Call

TON <Instance> (IN, PT, TB);

Inputs

Parameter	Data type	Description
<Instance>		1 … 8
IN	BOOL	IN = TRUE sets Q = TRUE, **after** the elapse of time PT
PT	UINT	time delay (unit Time_Base (TB)),
TB	UINT	unit of PT: 1 = msec 2 = sec 3 = min

Outputs

Parameter	Data type	Request/description
Q	BOOL	`OUT := TON <Instance>.Q` OUT = TRUE, when IN = TRUE and PT has elapsed
ET	UINT	`OUT := TON <Instance>.ET` ET = time elapsed since the active pulse phase

Comment

The parameter Time_Base (TB) is an extension of DIN 1131, Part 3 (IEC 65B(CO)85). This extension does not signify any restriction compared with the DIN. The time entry is, according to DIN “implementation-dependent”.

The maximum error in the function block is 210msec.

Values less then 105msec for PT do not make sense.

⇨Section 6.3 “Program execution”

Beispiel

```
VAR                    (* marker, not retain *)
     ramp : UINT;
END_VAR

VAR CONSTANT           (* data block *)
     delay : UINT := 8;
END_VAR

IF reset_flag THEN     (* initialisation block *)
     q_01 := FALSE;
END_IF;

IF e_07 AND e_08 THEN (* when HW inputs 7 and 8, then: *)
TON5 (e_01,delay,2); (* switch-on delay of output 1
                        is 8 seconds *)
q_01 := TON5.Q;        (* HW output 1 indicates output of
                        the function block, *)
                        (* i.e. the switch-on signal,
                        delayed by 8 seconds *)
ramp := TON5.ET;       (* time elapsed in seconds since
                        start of active pulse phase.*)
END_IF;
```

Retain instances

The first four instances of this function block are “retain”, i.e. the values of the inputs/outputs are held for up to 8 days after the supply voltage has been disconnected from the logic module. The DIN does not require this “retain” function.

After reconnecting the supply voltage, the values can be read, but the functionality of the function block (continuing to run automatically) is only available after an initialisation of the function block.

The following aspects must be observed for the initialisation of the function block:

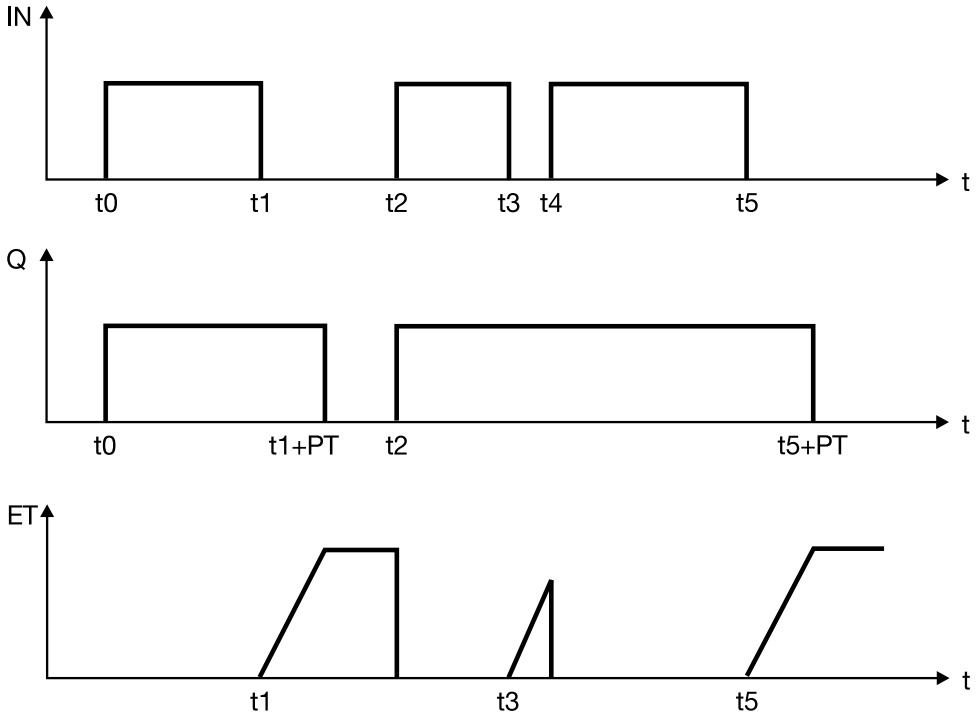
- the reduced entry for PT
- the time which has elapsed during the disconnection of the logic module from the supply voltage

The two program examples at the end of Section 6.10.3 “Pulse generator” show the principle of a possible initialisation of the function block after reconnecting the supply voltage.

.

6.10.5 Switch-off delay

The method of operation of the switch-off delay is shown in the following diagram:



Call

TOF <Instance> (IN, PT, TB);

Inputs

Parameter	Data type	Description
<Instance>		1 … 8
IN	BOOL	IN = TRUE sets Q = TRUE IN = FALSE => Q is set to FALSE after the delay time PT
PT	UINT	time delay (unit Time_Base (TB))
TB	UINT	unit of PT: 1 = msec 2 = sec 3 = min

Outputs

Parameter	Data type	Request/description
Q	BOOL	`OUT := TOF <Instance>.Q`
ET	UINT	`OUT := TOF <Instance>.ET` ET = time elapsed since the active pulse phase

Comment

The parameter Time_Base (TB) is an extension of DIN 1131, Part 3. This extension does not signify any restriction compared with the DIN. The time entry is, according to DIN “implementation-dependent”.

The maximum error in the function block is 210 msec.

Values less then 105 msec for PT do not make sense.

Section 6.3 “Program execution”

Beispiel

```
VAR                    (* marker, not retain *)
     ramp : UINT;
VAR CONSTANT           (* data block *)
     delay : UINT := 8;
END_VAR

IF reset_flag THEN     (* initialisation block *)
     q_01 := FALSE;
END_IF;

IF e_07 AND e_08 THEN (* When HW inputs 7 and 8, then: *)
TOF5 (e_01, delay,2);
                       (* switch-off delay of output 1
                       of 8 seconds *)
q_01 := TOF5.Q;        (* HW output 1 indicates output of
                       the function block, *)
                       (* i.e. the switch-off signal
                       which is delayed by 8 seconds *)
ramp := TOF5.ET;       (* time elapsed in seconds since
                       start of the active pulse phase. *)
END_IF;
```

Retain instances

The first four instances of this function block are “retain”, i.e. the values of the inputs/outputs are held for up to 8 days after the supply voltage has been disconnected from the logic module. The DIN does not require this “retain” function.

After reconnecting the supply voltage, the values can be read, but the functionality of the function block (continuing to run automatically) is only available after an initialisation of the function block.

The following aspects must be observed for the initialisation of the function block:

- the reduced entry for PT
- the time which has elapsed during the disconnection of the logic module from the supply voltage

The two program examples at the end of Section 6.10.3 “Pulse generator” show the principle of a possible initialisation of the function block after reconnecting the supply voltage.

6.10.6 Edge recognition (rising edge)

This function block recognises a rising edge (0 -> 1 transition).

Call

R_TRIG <Instance> (IN);

Input

Parameter	Data type	Description
<Instance>		1 … 8
IN	BOOL	if 0 -> 1 transition at IN, Q = TRUE

Outputs

Parameter	Data type	Request
Q	BOOL	`OUT := R_TRIG <Instance>.Q`

Comment

The operation of the function block corresponds to the following program code:

```
VAR CLK : BOOL; END_VAR
VAR Q : BOOL; END_VAR
VAR M : BOOL := 0; END_VAR
Q := CLK AND NOT M;
M := CLK;
```

The Q output holds its boolean value from one call to the next. It follows the transition of the input signals (IN) from “0” nach “1” and returns to “0” at the next call.

Example

```
VAR                     (* marker, not retain *)
     i : UINT;
     f : REAL;
END_VAR

IF (reset_flag) THEN
     q_01  := FALSE;
END IF;

R_TRIG7 (e_07);         (* recognition of a rising edge
                           on HW input 7 *)
q_01 := r_trig7.q;      (* HW output 1 indicates output
                           of the function block *)
```

6.10.7 Edge recognition (falling edge)

This function block recognises a falling edge (1 -> 0 transition).

Call

F_TRIG <Instance> (IN);

Input

Parameter	Data type	Description
<Instance>		1 … 8
IN	BOOL	If 1 -> 0 transition at IN, Q = TRUE

Outputs

Parameter	Data type	Request
Q	BOOL	`OUT := F_TRIG <Instance>.Q`

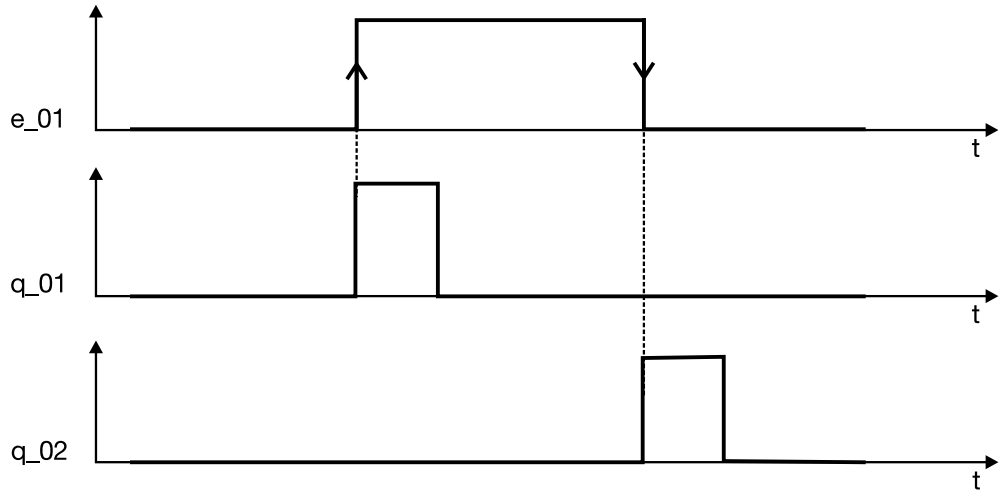
Comment

The operation of the function block corresponds to the following program code:

```
VAR CLK : BOOL; END_VAR
VAR Q : BOOL; END_VAR
VAR M : BOOL := 1; END_VAR
Q := NOT CLK AND NOT M;
M := NOT CLK;
```

The Q output holds its boolean value from one call to the next. It follows the transition of the input signals (IN) from “1” to “0” and returns to “0” at the next call.

Example




```
VAR
     i : UINT;
END_VAR

IF RESET_flag then
     q_01 := FALSE;
     q_01 := FALSE;
end_if;

R_TRIG7 (e_01);          (* with a positive edge on input 1,
                            output 1 appears briefly *)
     q_01 := R_TRIG7.Q;  (* output is just a short pulse *)

F_TRIG8 e_01);           (* with a negative edge on input 1,
                            output 2 appears briefly *)
     q_02 := F_TRIG8.Q; (* output is just a short pulse *)
```

6.10.8 Bistable function block SR

Bistable function (biased to set) with hold.

Call

SR <Instance> (S1, R);

Inputs

Parameter	Data type	Description
<Instance>		1 … 8
S1	BOOL	S1 = TRUE sets Q1 = TRUE
R	BOOL	R = TRUE sets Q1 = FALSE, when S1 = FALSE

Outputs

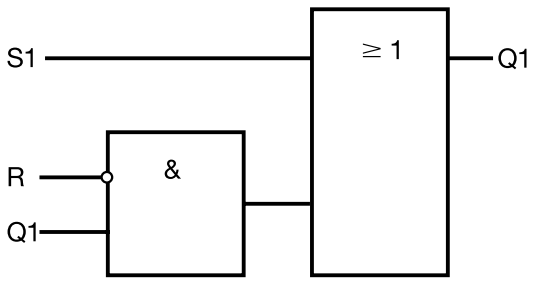
Parameter	Data type	Request
Q1	BOOL	OUT := SR <Instance>.Q1

Comment

After being set by S1, the Q1 output remains set (TRUE) until it is reset by R.

If S1 and R are operated (TRUE) simultaneously, the Q1 output will still be set (TRUE).

Logic function



Example

```
SR3 (e_07, e_08);      (* bistable function SR (instance 3) *)
                          biased to set *)
                       (* e_07 = set, e_08 = reset *)

q_01 := SR3.Q1;        (* HW output 1 indicates the output
                          of the function block *)
```

6.10.9 Bistable function block RS

Bistable function (biased to reset) with hold.

Call

RS <Instance> (S, R1);

Input

Parameter	Data type	Description
<Instance>		1 … 8
R1	BOOL	R1 = TRUE sets Q1 = FALSE
S	BOOL	S = TRUE sets Q1 = TRUE, when R1 = FALSE

Outputs

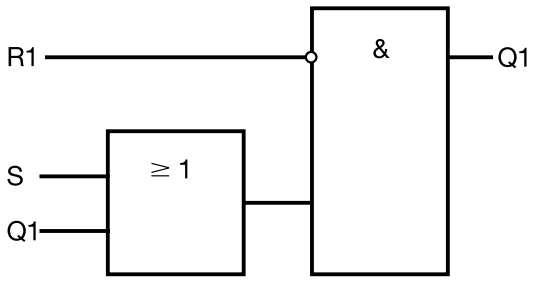
Parameter	Data type	Request
Q1	BOOL	`OUT := RS <Instance>.Q1`

Comment

After being reset, the Q1 output remains reset (FALSE) until it is set by S.

If S1 and R are operated (TRUE) simultaneously, the Q1 output will still be reset (FALSE).

Logic function



Example

```
RS3 (e_07, e_08);       (* bistable function RS (instance 3) *)
                        (* biased to reset *)
                        (* e_07 = reset, e_08 = set *)

q_01 := RS3.Q1;         (* HW output 1 indicates output
                        (* of the function block *)
```

6.10.10 Hardware counters

Two hardware counters

The logic module includes two hardware counters. The hardware counters are JUMO-specific function blocks, which are not derived from DIN 1131, Part 3.

Counter types

The first hardware counter offers three different count functions (through a multiplexer), the second hardware counter is always a total-count counter.

The functions of the first counter are:

- Total count (up counter)
- Period count (time measurement)
- On-time (time measurement)

The first parameter of the call to the first counter determines the count function.

If this parameter changes from one call to the next, the current count value is cancelled.

Total count

This count function establishes the number of pulses at an input (up counter).

Period count

This count function measures the time between two 0 → 1 transitions.

On-time

This count function measures the time during which a logic signal is high.

Maximum counting frequency

The maximum counting frequency for both counters is 10 kHz, the maximum count value is 65536.

Inputs

The inputs for both hardware counters are fixed.

Counter	Count function	Logic input
1	Total count	6
1	Period count	7
1	On-time	8
2	Total count	5

Call counter 1 (total count)

C_HW1 (1);

Input

Parameter	Description
1	Total-count counter

Output

Parameter	Data type	Description
CV	UINT	`OUT := C_HW1.CV`

Comment

The pulses counted are on logic input 6 of the logic module.

The first call starts the counter. At each subsequent call the count value is read out and the count procedure starts again.

Example

```
VAR                    (* marker, not retain *)
     HW_countstate1: UINT;
     HW_countstate2: UINT;
     count_active : bool;
END_VAR

VAR RETAIN             (* marker, retain *)
     j : UINT;
END_VAR

IF e_01 THEN
     count_active := FALSE;
     HW_countstate1 := 0;
     C_HW1 (1);
END_IF;

IF not count_active THEN
     IF e_02 THEN
          count_active := TRUE;
          C_HW1 (1);
          HW_countstate1 := C_HW1.CV;
     END_IF;
END_IF;
```

Call counter 1 (period count)

C_HW1 (2, IN);

Inputs

Parameter	Data type	Description
2		period-count counter
IN	UINT	time raster 0 … 7

The parameter IN determines which time raster is used for the counter

IN	Time raster	Largest measurable value with max. count 65536 (time raster * 65536)
0	200 ns	13.11 ms
1	400 ns	26.21 ms
2	800 ns	52.43 ms
3	1.6 µs	104.86 ms
4	3.2 µs	209.72 ms
5	6.4 µs	419.43 ms
6	12.8 µs	838.86 ms
7	25.6 µs	1.678 s

Output

Parameter	Data type	Request
CV	UINT	`OUT := C_HW1.CV;`

Comment

The time is measured between two 0 → 1 transitions at logic input 7.

The first call to the function block starts the function. The counting process starts with the next 0 → 1 transition, the following 0 → 1 transition stops the count.

The next call to the function block, made with the same parameters, initialises the counter for the next counting process. The result of the previous counting process is transferred to C_HW1.CV and can be requested.

Successive calls to the counter, with different parameters, mean that the counter will be initialised again and the count result (value) will be cancelled.

Example

```
VAR                    (* marker, not retain *)
     HW_countstate1: UINT;
     HW_countstate2: UINT;
     count_active : bool;
END_VAR

VAR RETAIN             (* marker, retain *)
     j : UINT;
END_VAR

IF e_07 THEN
     C_HW1 (2,7);
     HW-countstate1 := C_HW1.CV;
END_IF;
```

Call counter 1 (On-time)

C_HW1 (3, IN);

Inputs

Parameter	Data type	Description
3		On-time counter
IN	UINT	time raster 0 … 7

The parameter IN determines which time raster is used for the counter:

IN	Time raster	Largest measurable value with max. count 65536 (time raster * 65536)
0	200 ns	13.11 ms
1	400 ns	26.21 ms
2	800 ns	52.43 ms
3	1.6 µs	104.86 ms
4	3.2 µs	209.72 ms
5	6.4 µs	419.43 ms
6	12.8 µs	838.86 ms
7	25.6 µs	1.678 s

Output

Parameter	Data type	Request
CV	UINT	`OUT := C_HW1.CV;`

Comment

The time is measured between a 0 → 1 transition and a 1 → 0 transition at logic input 8.

The first call to the function block starts the function. The counting process starts with the next 0 → 1 transition, the following 1 → 0 transition stops the count.

The next call to the function block, made with the same parameters, initialises the counter for the next counting process. The result of the previous counting process is transferred to C_HW1.CV and can be requested.

Successive calls to the counter, with different parameters, mean that the counter will be initialised again and the count result (value) will be cancelled.

Example

```
VAR                    (* marker, not retain *)
     HW_countstate1: UINT;
     HW_countstate2: UINT;
     count_active : bool;
END_VAR

VAR RETAIN             (* marker, retain *)
     j : UINT;
END_VAR

IF e_08 THEN
     C_HW1 (3,7);
     HW_countstate1 := C_HW1.CV;
END_IF;
```

Call counter 2 (total count)

C_HW2 ( );

The second hardware counter does not require any parameters, since it is always a total-count counter.

Output

Parameter	Data type	Description
CV	UINT	`OUT := C_HW2.CV`

Comment

The pulses which are counted are on logic input 5 of the logic module.

The first call starts the counter. Each subsequent call reads out the count state (value) and restarts the counting process.

Example

```
VAR                    (* marker, not retain *)
     HW_countstate1: UINT;
     HW_countstate2: UINT;
     count_active : bool;
END_VAR

VAR RETAIN             (* marker, retain *)
     j : UINT;
END_VAR

IF e_01 THEN
     count_active := FALSE;
     HW_countstate2 := 0;
     C_HW2 ();
END_IF;

IF not count_active THEN
     IF e_02 THEN
          count_active := TRUE;
          C_HW2 ();
          HW_countstate2 := C_HW21.CV;
     END_IF;
END_IF;
```

6.11 System variables

The following system variables are available for use in the ST program:

- Reset_Flag
- Combination_Alarm
- Capacitor_Fault
- Divide_BY0_Fault
- SQRT_Fault
- Week_Day
- Bool_In 01…08
- Bool_Out 01…08
- Long_In 01…04
- Long_Out 01…04
- Real_In 01…08
- Real_Out 01…08
- e_01…08
- q_01…06

Reset_Flag

Data type	Description
BOOL	TRUE on fresh start of a program

Comment The Reset_Flag is used to initialise variables/signals.

Combination_ Alarm

Data type	Description
BOOL	TRUE when combination alarm occurs, i.e. if not all input network-variables are written to

Capacitor_Fault

Data type	Description
BOOL	TRUE when storage capacitor is discharged

Comment If Combination_Alarm is active (TRUE), all the buffered data are reset.

Divide_BY0_ Fault

Data type	Description
BOOL	TRUE on division by 0

Comment The state of the variable refers to the last division operation or the last modulo operation.
If a division by 0 occurs, the function which causes the error returns the value 0 as a result.

SQRT_Fault

Data type	Description
BOOL	TRUE, if a negative argument is passed to the square root function

Comment

If the square root function (⇨ Page 38) is called with a negative argument, the system variable is set. The value 0 is returned as a result for the function.

Week_Day

Data type	Description
UINT	0 = Monday, 1 = Tuesday, …, 6 = Sunday

Bool_In01 … 08

Data type	Description
BOOL	These network variables can be combined with any boolean network variables from other modules

Bool_Out01 … 08

Data type	Description
BOOL	These network variables can be combined with any boolean network variables from other modules

Long_In01 … 04

Data type	Description
UINT	These network variables can be combined with any long (UINT) network variables from other modules

Long_Out01 … 04

Data type	Description
UINT	These network variables can be combined with any long (UINT) network variables from other modules

Real_In01 … 08

Data type	Description
REAL	These network variables can be combined with any real network variables from other modules

Real_Out01 … 08

Data type	Description
REAL	These network variables can be combined with any real network variables from other modules

e_01 … 08

Data type	Description
BOOL	These variables represent the states of the logic hardware inputs

Comment

The variables e_01 … e_08 can only be read.

Exception: in the debugger the inputs can be forced to certain states for test purposes (Forcing, ⇨ Page 21).

q_01 … 06

Data type	Description
BOOL	These variables represent the states of the logic hardware outputs

Comment

The variables q01 … q06 can be read and written.

Example hardware inputs/outputs

```
(* marker, not retain *)
VAR
     dummy_bool: BOOL;
END_VAR
```

```
(* marker, retain *)
VAR RETAIN
     dummy_uint: REAL;
END_VAR
```

```
(* data block *)
VAR CONSTANT
     dummy_real : REAL := 3.14159;
     dummy_dt : DT := dt#1998-01-01-12:00:00;
END_VAR
```

```
(* initialisation; all outputs to FALSE after reset*)
IF reset_flag THEN
q_01 := FALSE;
q_02 := FALSE;
q_03 := FALSE;
q_04 := FALSE;
q_05 := FALSE;
q_06 := FALSE;
END_IF;
```

```
(* storage capacitor discharged OR combination alarm active *)
IF Capacitor_Fault OR Combination_Alarm THEN
q_01 := FALSE;
q_02 := FALSE;
q_03 := FALSE;
q_04 := FALSE;
q_05 := FALSE;
q_06 := FALSE;
ELSE;
(* ST program; passing-on function of the hardware inputs
               to the outputs *)
q_01 := e_01;   (* logic hardware output 1 =
                 logic hardware input 1 *)
q_02 := e_02;   (* logic hardware output 2 =
                 logic hardware input 2 *)
q_03 := e_03;   (* logic hardware output 3 =
                 logic hardware input 3 *)
q_04 := e_04;   (* logic hardware output 4 =
                 logic hardware input 4 *)
q_05 := e_05;   (* logic hardware output 5 =
                 logic hardware input 5 *)
q_06 := e_06;   (* logic hardware output 6 =
                 logic hardware input 6 *)
END_IF;
```

7.1 Behaviour after a power interruption

- After 10sec the full functionality of the module is achieved, during these 10 sec the relays remain de-energised.
- All input network-variables are set to their default values, until the bound network-variables have been updated.

7.2 Behaviour on faulty communication

If the bound network-variables are no longer being regularly updated, then these variables are set to their default values and a combination alarm is output.

7 Special module conditions

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

M. K. Juchheim GmbH & Co

36035 Fulda, Germany
Phone (0661) 6003 - 0
Fax (0661) 6003 - 607
Telex 49 701 juf d
email JUMO_de@e-mail.com

UK
Jumo Instrument Co. Ltd.
Temple Bank, Riverway
Harlow, Essex CM20 2TT
Phone (01279) 63 55 33
Fax (01279) 63 52 62

USA
Jumo Process Control Inc.
735 Fox Chase
Coatesville, PA 19320
Phone 610 - 380 - 8002
800 - 554 JUMO
Fax 610 - 380 - 8009

MEASUREMENT AND CONTROL

JUMO mTRON

Logic module

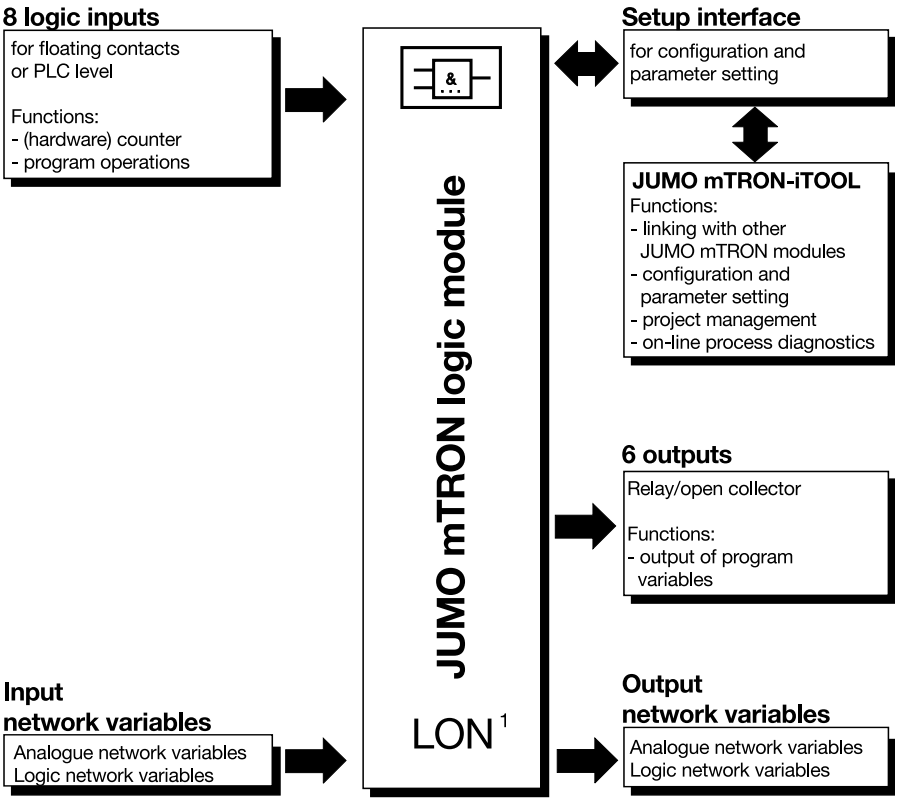
for stock items see price list

Brief description

The unit is a module of the JUMO mTRON control and automation system. The plastic housing measures 91 mm x 85.5 mm x 73.5 mm (W x H x D) and is mounted on a standard rail.
The logic module processes programs which are created according to IEC 1131 Part 3 “Structured text”. It permits logic, arithmetic, bit sequence, comparison and selection operations. A library contains standardised function blocks for timed operations, up/down counters, edge recognition and bistable functions. The module features eight logic inputs (floating contact or PLC level) and six relay or open-collector outputs. A network connection is available for the exchange of data. A screened twisted pair is used as a transmission line. There is a setup interface for module parameter setting and configuration from a PC under the JUMO mTRON-iTOOL project design software. The electrical connections are made through plug-in connectors with screw terminals.

Type 704030/0-...

Block structure



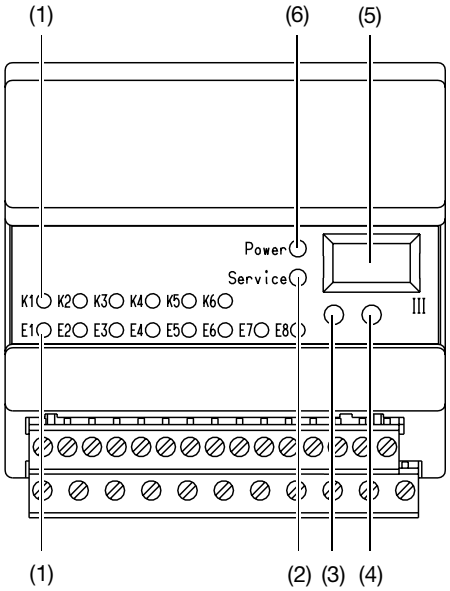
1. LON® = **L**ocal **O**perating **N**etwork
Registered trademark of the ECHELON Corporation

Features

- **8 logic inputs**
- **6 switching outputs**
- **Real-time clock**
- **Network inputs**
  8 logic, 4 long, 8 real
- **Network outputs**
  8 logic, 4 long, 8 real
  combined alarm
  switching status of inputs/outputs
  date/time
- **2 hardware counters**
  for counting pulses and time measurements via the logic inputs
- **Programming through “Structured text” to DIN 1131**
- **Function blocks to DIN 1131**
- **Debugger**
  for program testing
  (via JUMO mTRON iTOOL)
- **Setup interface**
  For configuration and parameter setting, the module is linked to a PC via a PC interface
- **Plug & Play function**
  Problem-free replacement of modules without re-configuration

8.99/00336365

Displays and controls



(1)	**Status LED, yellow** for the outputs K1 to K6 and the logic inputs E1 to E8, lights up if the output is active /contact is closed or voltage on the logic input	(4)	**Installation key** the module reports to the JUMO mTRON-iTOOL project design software or the operating unit
(2)	**Service LED, red** - lights up on operating fault - blinks when the mechanical connection to the module from the JUMO mTRON-iTOOL or the operating unit is checked by a test signal (“wink”)	(5)	**Setup interface** for the PC interface line which connects the module to the PC
(3)	**Switch** for the termination resistance of the LON network	(6)	**Power LED, green** lights up when the supply is switched on

Technical data

Hardware inputs

Logic inputs

Activation:
– floating contacts
– PLC level

Functions:
- (hardware) counters
- program operations

Hardware outputs

Switching outputs

Function:
- output of program variables

Relay outputs
Type: (n.o.) make
Nominal voltage: 250V
Nominal current: 3A
Rating: 3A, 250V AC,
resistive load
Life: $5 \cdot 10^5$ operations
with resistive load
Contact material: AgCdO (hard gold plated)
Minimum load: 5V 10mA DC

Open-collector outputs
Rating: 50V 0.5A max.
short-circuit proof

Input network variables

Analogue network variables

- 8 variables “real” type
- 4 variables “long” type

Logic network variables

- 8 variables “bool” type

Output network variables

Analogue network variables

Output cycle: 420msec

- 8 variables “real” type

Logic network variables

Output cycle: event-controlled
every 105 msec, but at least every 6sec

- 8 variables “bool” type

Further network variables

Output cycle: 420 msec

- 4 variables “long” type
- date and time
- combined alarm
- switching status of the inputs
- switching status of the outputs

General data

Environmental conditions to EN 61 010

Operating and ambient temperature:
0 — 55°C
Permitted storage temperature:
-40 to +70°C
Relative humidity: rH 80% max.
Pollution degree 2
Overvoltage category 2

Housing

Material: plastic, self-extinguishing
Flammability Class: UL 94 V0
Protection: IP20 (to EN 60529)
Mounting: on a standard rail

Supply

110 — 240V AC +10/–15%, 48 — 63Hz,
or 20 — 53V AC/DC, 48 — 63Hz
Power consumption: 5VA max.

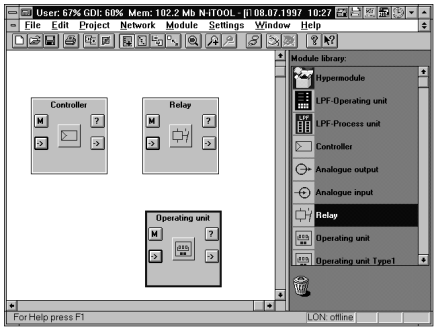
Network (LON interface)

Transceiver: free topology FTT-10A
Topology: ring, star, line or
mixed structure
Baud rate: 78 kbaud
Max. lead length
(depending on lead type):
line: 2700m
star: 500m
ring: 500m
mixed: 500m
Max. number of modules: 64

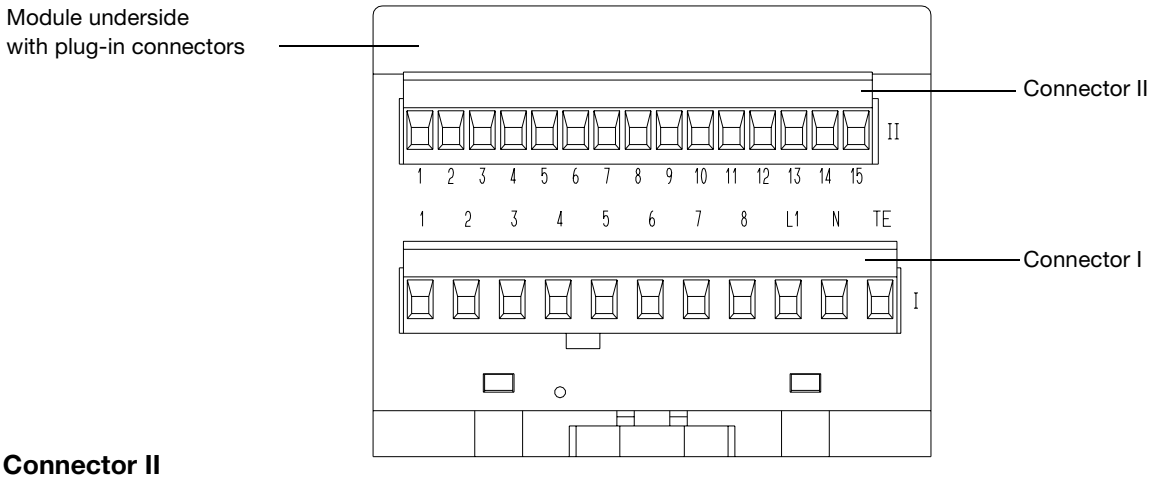
Operation and project design

Operation, parameter setting and configuration of JUMO mTRON modules can be carried out from the JUMO mTRON operating unit.

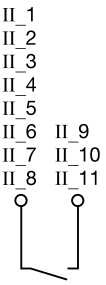
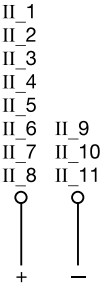
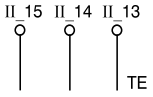
The JUMO mTRON-iTOOL project design software permits convenient design and start-up of a JUMO mTRON system. The projects can be archived and documented. Individual modules are linked via LON by assigning network-variable (NV) names.



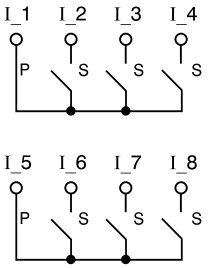
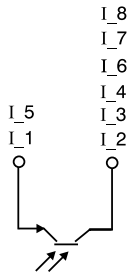
Connection diagram



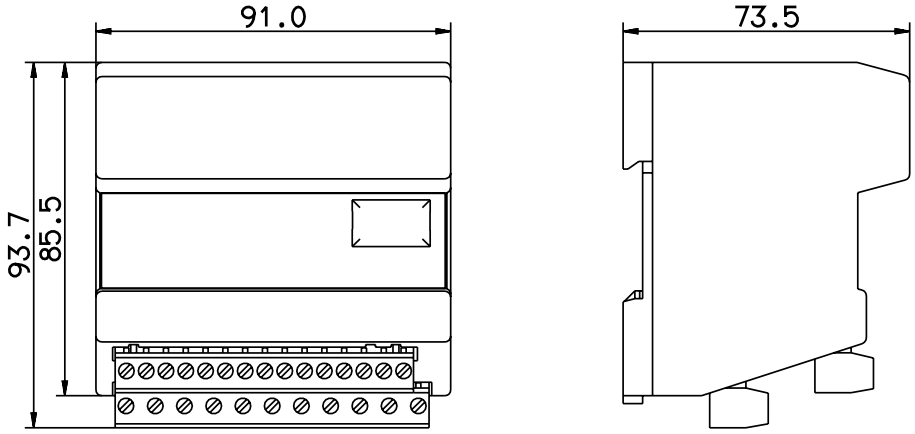
Connector II

Connection for	Terminals								Diagram
Logic inputs	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**	**E7**	**E8**	
Floating contact	II_1 II_9	II_2 II_9	II_3 II_9	II_4 II_9	II_5 II_9	II_6 II_9	II_7 II_9	II_8 II_9	
Voltage -35V to 4.5V -> low 13V to 35V -> high	II_1 + II_9 —	II_2 + II_9 —	II_3 + II_9 —	II_4 + II_9 —	II_5 + II_9 —	II_6 + II_9 —	II_7 + II_9 —	II_8 + II_9 —	
	The terminals II_9, II_10 and II_11 are linked internally.								
LON interface	II_13 = TE						screen		
	II_14 = Net_A II_15 = Net_B						any polarity		
Technical earth	II_13 II_TE								

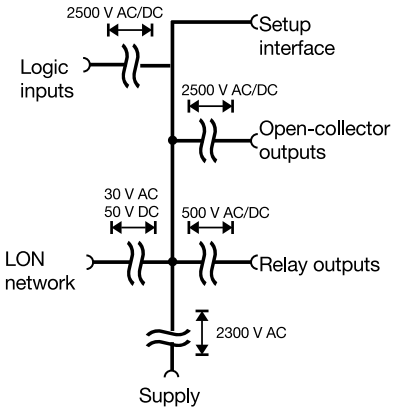
Connector I

Connection for	Terminals						Notes	Diagram
Outputs	K1	K2	K3	K4	K5	K6		
Relay output 3A, 250VAC, resistive load	I_1 I_2	I_1 I_3	I_1 I_4	I_5 I_6	I_5 I_7	I_5 I_8	P = common S = n.o. (make)	
Open-collector output 50V 0.5A max.	I_1 I_2 +	I_1 I_3 +	I_1 I_4 +	I_5 I_6 +	I_5 I_7 +	I_5 I_8 +		
	I_1 and I_5 are not linked internally!							
Supply as label	AC			DC				
	I_L1 line I_N neutral I_TE technical earth			I_L1 } any I_N } polarity I_TE technical earth				

Dimensions



Isolation



Ordering details

	(1)		(2)		(3)
704030/0-	. . .	-	. . .	-	. .

(1) Inputs .....

Inputs	Code
8 logic inputs, volt-free from the system	**178**
8 voltage inputs 0/24V	**188**

(2) Outputs .....

Outputs	Code
6 logic outputs (relay, n.o. make)	**156**
6 open-collector outputs (transistor) (available from October ´98)	**176**

(3) Supply .....

Type	Code
110 — 240V AC +10/–15%, 48 — 63Hz	**23**
20 — 53V AC/DC, 48 — 63Hz	**22**

Standard accessory

1 Installation Instructions 70.4030

Accessories

PC interface
with TTL/RS232C converter
for connecting the module to a PC; length 2m.
Sales No. 70/00301315

Project design software
JUMO mTRON-iTOOL
Using the JUMO mTRON- iTOOL project design software, the modules can be designed graphically on the PC. The user is able to link modules of the JUMO mTRON family and to configure the application-specific parameters.

JUMO mTRON System Manual
Documentation of configuration, parameter setting and installation of the modules.
Sales No. 70/00334336

JUMO mTRON modules

Controller module
Data Sheet 70.4010

Relay module
Data Sheet 70.4015

Analogue input module
Data Sheet 70.4020

Analogue output module
Data Sheet 70.4025

Logic module
Data Sheet 70.4030

Operating unit
Data Sheet 70.4035

Communication module
Data Sheet 70.4040

JUMO mTRON-iTOOL
project design software
Data Sheet 70.4090