

Bending machine 24V

Program instructions

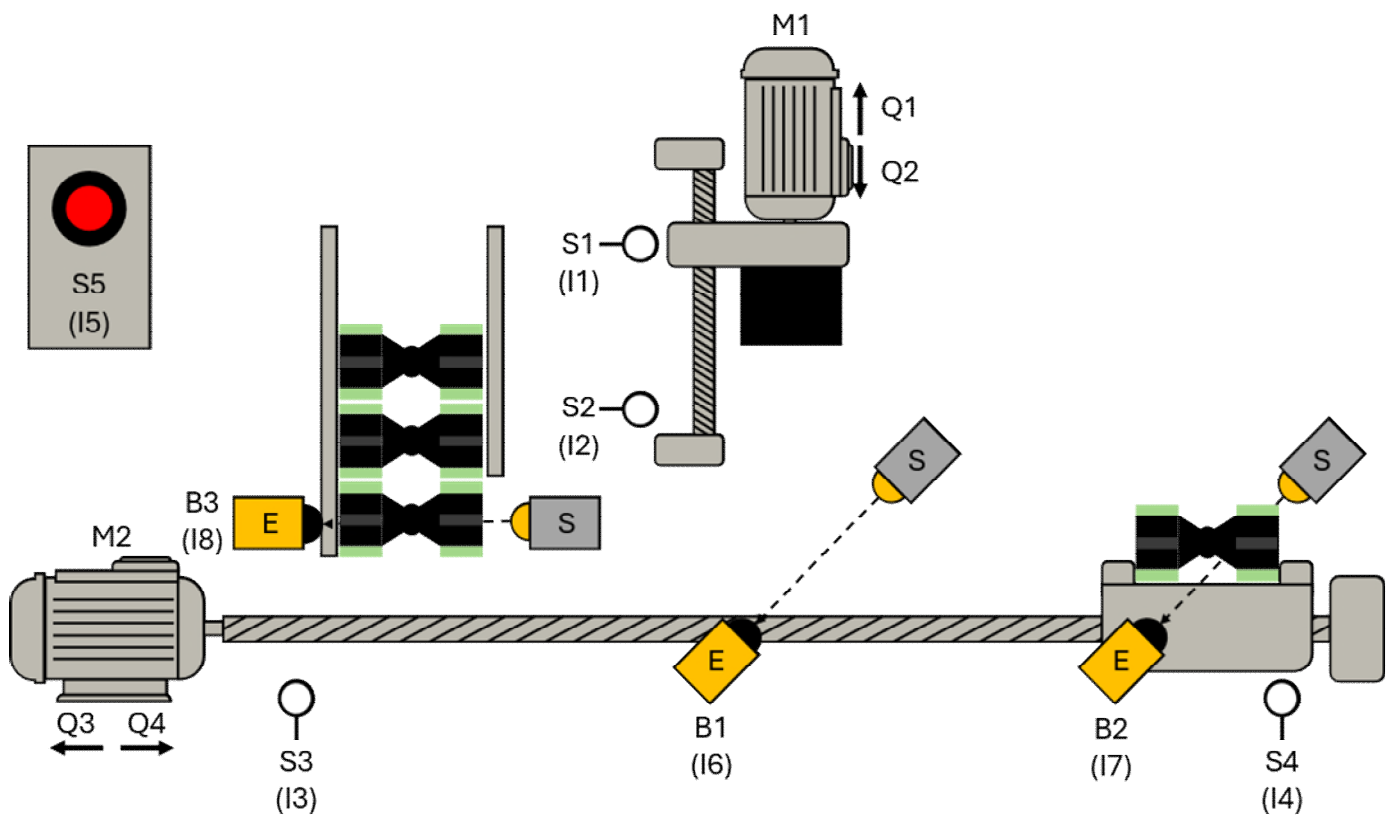


Table of contents

7	Program instructions.....	1
7.1	Flip-flops.....	1
7.1.1	Dominant resetting flip-flop.....	2
7.1.2	Dominant setting flip-flop.....	5
7.1.3	Exercise - Dominance behavior.....	8
7.2	Flanks.....	13
7.2.1	Recognize positive edge R_TRIG.....	14
7.2.2	Recognize negative edge F_TRIG.....	16
7.3	Times.....	18
7.3.1	Switch-on delay TON.....	20
7.3.2	Switch-off delay TOF.....	22
7.3.3	Pulse TP.....	24
7.3.4	Exercise - IEC time functions.....	26
7.4	Counter.....	31
7.4.1	Count forward CTU.....	33
7.4.2	Counting backwards CTD.....	35
7.4.3	Counting forwards and backwards CTUD.....	37
7.4.4	Exercise - IEC meter.....	39
7.5	IF statement [ST / SCL].....	44
7.5.1	IF...THEN - Instruction.....	44
7.5.2	IF...THEN...ELSE statement.....	45
7.5.3	IF...THEN...ELSIF - Instruction.....	46
7.6	CASE structure [ST / SCL].....	48

7 Program instructions

7.1 Flipflops

A flip-flop is a basic digital storage element that can store a binary state (0 or 1). It holds a momentary signal state and can be set or reset by special control commands.

Memory functions play a decisive role in PLC programming, especially in the control of sequential processes and the management of states. They are fundamental for the implementation of logics that go beyond simple on/off controls.

Here are some key aspects of what memory functions are used for:

Condition maintenance:

Memory functions enable a PLC to maintain the status of inputs, internal states or outputs over time. This is particularly important in applications where states must be maintained over a cycle or even over several cycles, such as with start/stop processes of motors or the monitoring of safety functions.

Control of processes:

In production processes where certain steps have to be carried out in a set sequence, memory functions help to save the current step and activate the next step based on the conditions fulfilled.

Debouncing of input signals:

Memory functions can be used to minimize the effects of bouncing or interference on input signals. Storing a stable state of an input prevents short-term fluctuations from leading to unwanted actions.

Interference treatment:

Memory functions can be used to record and save error states as soon as they occur. The fault can be reset by a user acknowledgement, for example.

Process control:

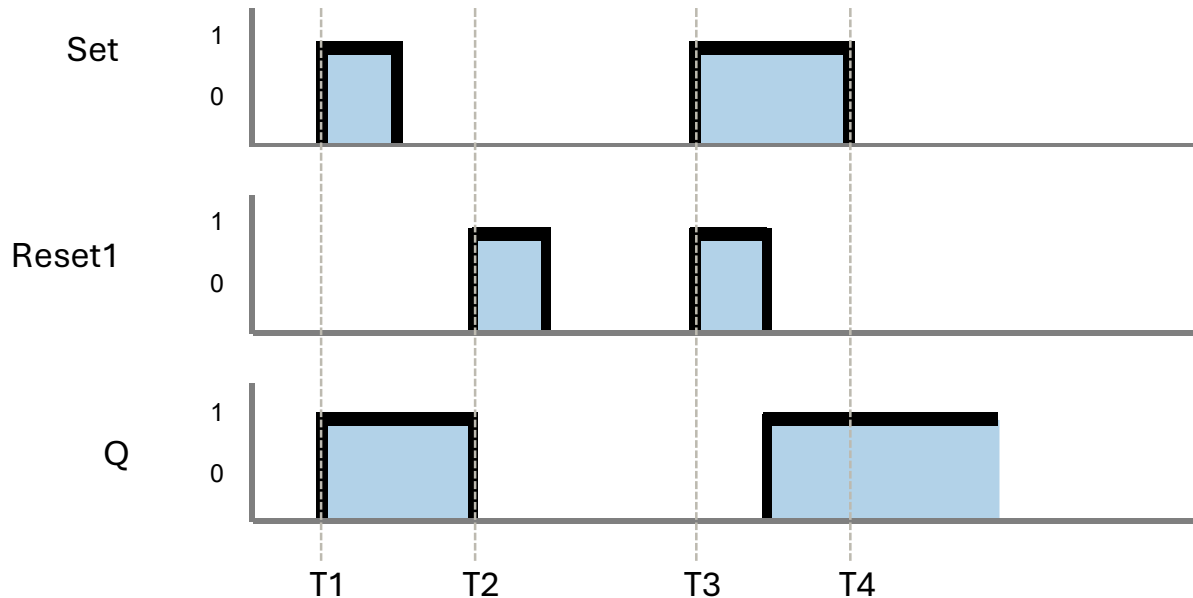
Storage functions are indispensable for processes that do not run continuously, but where actions have to be started or stopped depending on certain events. They enable the storage of events or states that can be retrieved and processed at a later point in time.

A save function is characterized by a set command and a reset command. A data area for saving the signal status must be specified above the memory function.

The memory function can be used with dominant (priority) setting or dominant be executed with resetting memory behavior. The dominant input is identified by a "1".

7.1.1 Dominant resetting flip-flop

Use the instruction to set or reset the bit of a specified operand, depending on the signal status at the Set and Reset1 inputs. The current signal status of the operand is transferred to output Q and can be queried at this output. The dominant behavior is indicated by the "1" at the Reset input.



Picture 1 Pulse diagram - flip-flop dominant resetting

T1

If the signal status at the Set input is "1" and at the Reset1 input is "0", the specified operand is set to "1".

T2

If the signal status at the Set input is "0" and at the Reset1 input is "1", the specified operand is reset to "0".

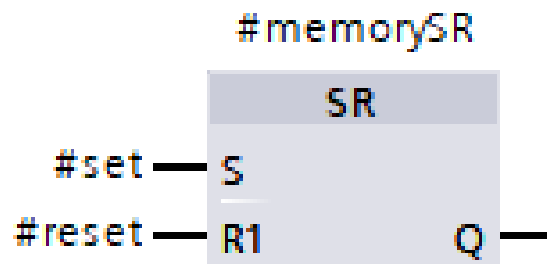
T3

The Reset1 input dominates the Set input. With a signal status of "1" at both inputs Set and Reset1, the signal status of the specified operand is reset to "0".

T4

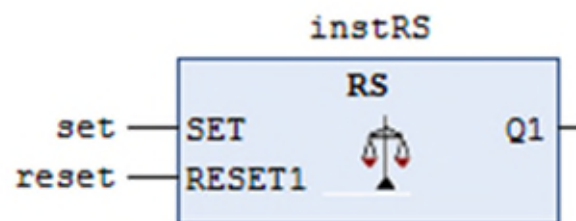
If the signal state is "0" at both inputs Set and Reset1, the instruction is not executed. In this case, the signal status of the operand remains unchanged.

At Siemens, the dominant resetting flip-flop corresponds to the "SR" instruction. A variable of the data type "BOOL" must be connected above the function block call as memory.



Picture 2 Flip-flop dominant resetting - Siemens

In CODESYS and Beckhoff, this is realized by the "RS" instruction. An instance of the data type "RS" must be connected above the function block call, as this is a function block call.



Picture 3 Flip-flop dominant resetting - Beckhoff

In the ST textual programming language, controller manufacturers such as Beckhoff or CODESYS offer the option of calling the RS element as a reset-dominant flip-flop, also as a function block. For this purpose, just as in FBD, an instance of type "RS" must be declared in the function block interface, which is then called in the implementation part.

//Declaration, Interface

VAR

instRS : RS; //Declaration of instance (Multi-instance)

END_VAR

//Instruction, instance call

**instRS(SET := varBoolSet ,
 RESET1 := varBoolReset,
 Q1 => varBoolQ);**

Picture 4 ST instruction - dominant resetting

This option is not available with Siemens controllers. Here, the flip-flop must be programmed using an IF instruction.

```
// IF statement
IF Reset1 THEN
  Q := false;
ELSIF Set THEN
  Q := true;
END_IF;
```

Picture 5 IF statement - dominant resetting

-  The function of the IF instruction is described in detail in this chapter under "7.5 IF instruction [ST/SCL]".

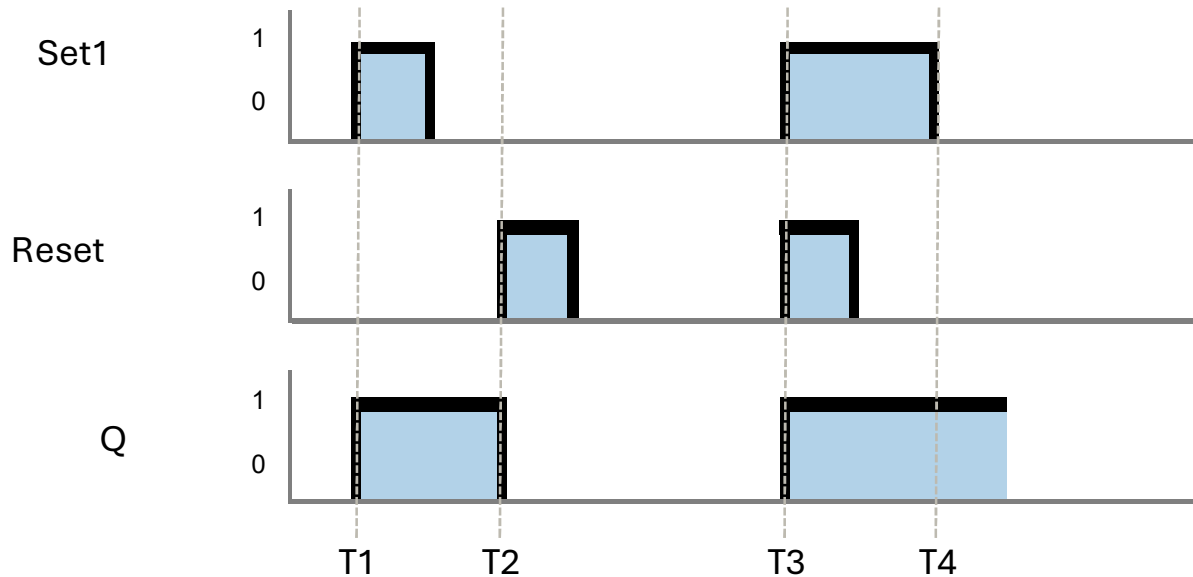
Alternatively, this can also be implemented by logically linking a self-hold.

```
//Logical connection
Q := NOT Reset1 AND (Q OR Set);
```

Picture 6 Logical connection - dominant resetting

7.1.2 Dominant setting flip-flop

Use the Reset or Set instruction to set the bit of a specified operand, depending on the signal status at the Reset and Set1 inputs. The current signal status of the operand is transferred to output Q and can be queried at this output. Dominance behavior is indicated by the "1" at the set input.



Picture 7 Pulse diagram - flip-flop dominant setting

T1

If the signal status at input Set1 is "1" and at input Reset "0", the specified operand is set to "1".

T2

If the signal status at the Reset input is "1" and at the Set1 input is "0", the specified operand is reset to "0".

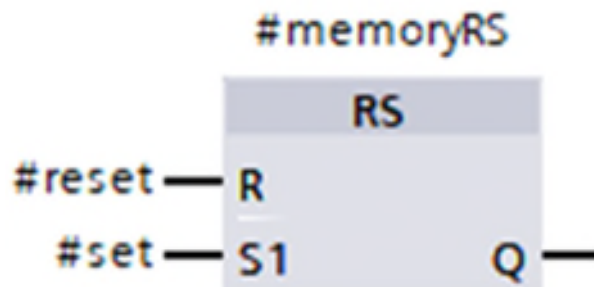
T3

The Set1 input dominates the Reset input. With a signal status of "1" at both inputs, Reset and Set1, the signal status of the specified operand is set to "1".

T4

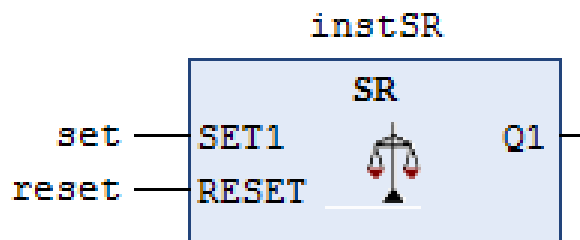
If the signal state is "0" at both inputs Reset and Set1, the instruction is not executed. In this case, the signal status of the operand remains unchanged.

With Siemens, the dominant setting flip-flop corresponds to the "RS" instruction. A variable of the data type "BOOL" must be connected above the function block call as memory.



Picture 8 Flip-flop dominant setting - Siemens

In CODESYS and Beckhoff, this is realized by the "SR" instruction. An instance of the data type "SR" must be connected above the function block call, as this is a function block call.



Picture 9 Flip-flop dominant setting - Beckhoff

In the ST textual programming language, control manufacturers such as Beckhoff or CODESYS offer the option of calling the SR element as a set-dominant flip-flop, also as a function block. For this purpose, just as in FBD, an instance of type "SR" must be declared in the function block interface, which is then called in the implementation part.

//Declaration, Interface

VAR

instSR : SR; **//Declaration of instance (Multi-instance)**

END_VAR

//Instruction, instance call

instSR(SET1 := varBoolSet,
RESET := varBoolReset,
Q1 => varBoolQ);

Picture 10 ST instruction - dominant setting

This option is not available with Siemens controllers. Here, the flip-flop must be programmed using an IF instruction.

```
//IF statement
IF Set1 THEN
  Q := true;
ELSIF Reset THEN
  Q := false;
END_IF;
```

Picture 11 IF statement - dominant setting

-  The function of the IF instruction is described in detail in this chapter under "7.5 IF instruction [ST/SCL]".

Alternatively, this can also be implemented by logically linking a self-hold.

```
//Logical connection
Q := (NOT Reset AND Q) OR Set1;
```

Picture 12 Logical link - dominant setting



7.1.3 Exercise - Dominance behavior

Goal

In this exercise, the dominance behavior of the two flip-flops SR and RS is to be programmed in a test function block and learned in practice.

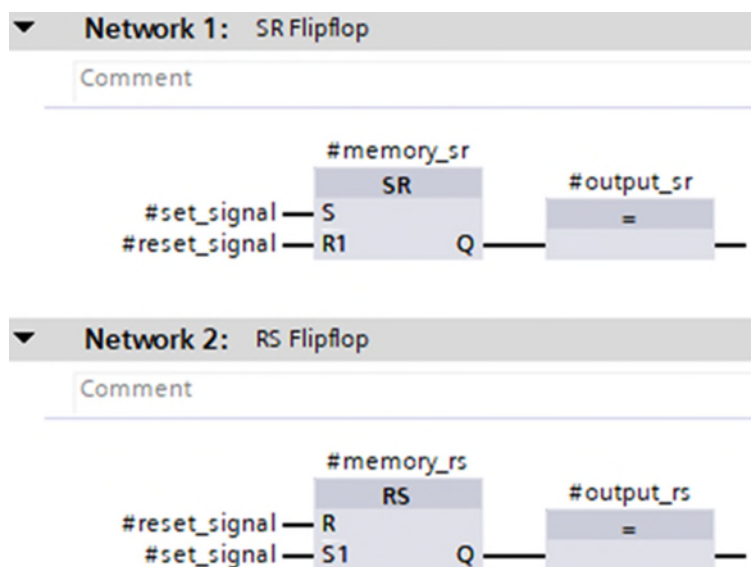
The "set_signal" input parameter is used to control the set input of the two memory elements. The "reset_signal" parameter is responsible for resetting. The status of the SR flip-flop is displayed via the "output_sr" output parameter, that of the RS flip-flop via the "output_rs" output.

Task

- Create a function block with the following function block interface:



	Name	Data type	Comment
1	▼ Input		
2	■ set_signal	Bool	set
3	■ reset_signal	Bool	Reset
4	▼ Output		
5	■ output_sr	Bool	Output SR
6	■ output_rs	Bool	Output RS
7	▼ InOut		
8	■ <Add new>		
9	▼ Static		
10	■ memory_sr	Bool	Internal Memory SR
11	■ memory_rs	Bool	Internal Memory RS
12	▼ Temp		
13	■ <Add new>		
14	▼ Constant		
15	■ <Add new>		

- The following program is to be implemented:



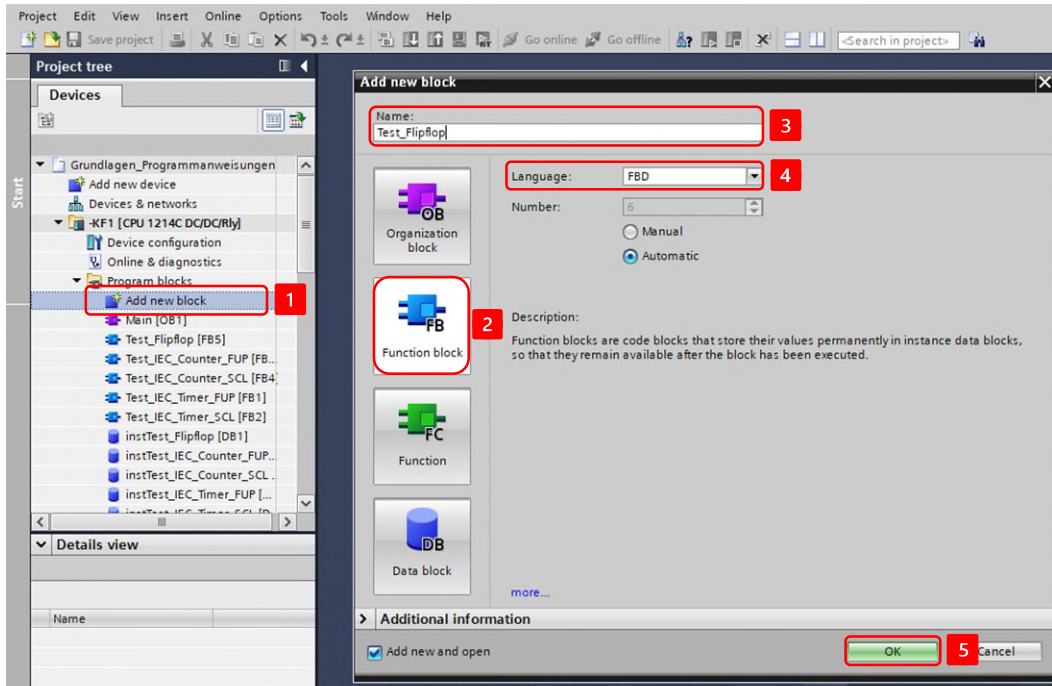
- Call up the created module in "MAIN".
- The input variables can be controlled and the output variables monitored via the instance. Alternatively, if available, these can also be connected to buttons and display elements via the function block interface when called.

If only the "set_signal" or "reset_signal" input is activated, both memory elements behave identically. If "set_signal" and "reset_signal" are set to "TRUE" at the same time, the outputs "output_sr" and "output_rs" differ in their signal status due to the different dominance behavior.

-  Alternatively, the prepared block "Test_Flipflop [FB5]" from the template project "Grundlagen_Programmanweisungen.zap17" can also be used.
-  The "Commissioning (software)" chapter can provide additional assistance in interpreting the program status.

Procedure:

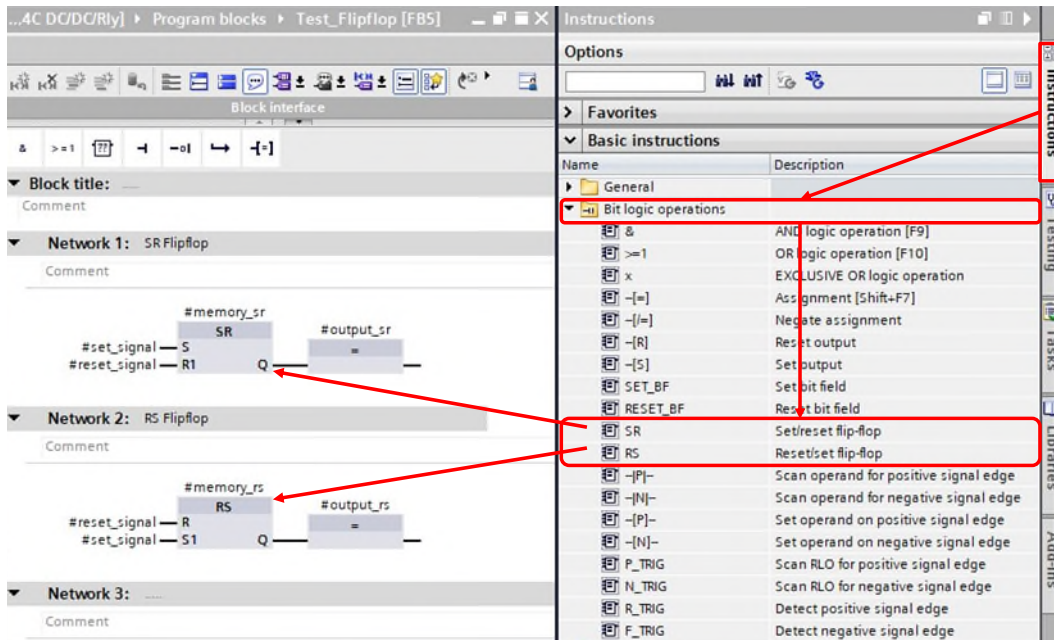
1. create a new function block, select the desired programming language and assign a meaningful name:



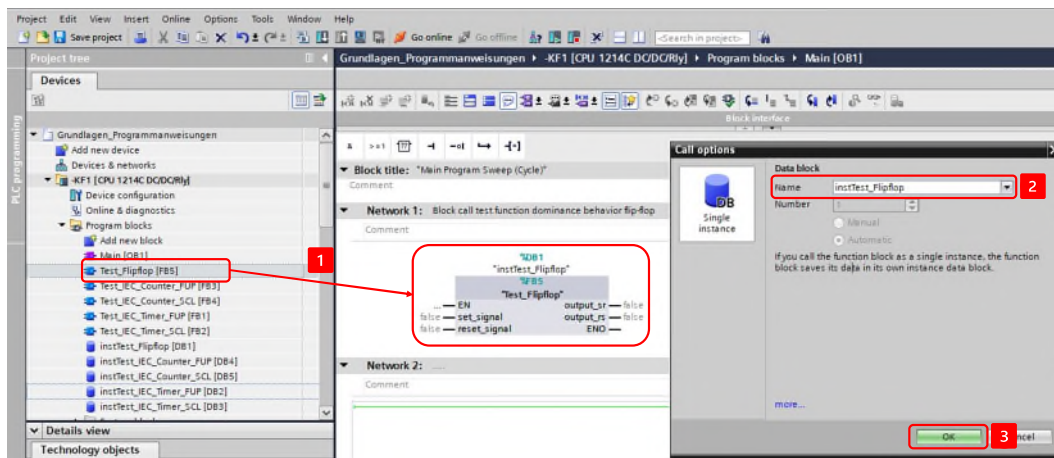
2. declare the following variables in the function block interface:

	Name	Data type	Comment
1	Input		
2	set_signal	Bool	set
3	reset_signal	Bool	Reset
4	Output		
5	output_sr	Bool	Output SR
6	output_rs	Bool	Output RS
7	InOut		
8	<Add new>		
9	Static		
10	memory_sr	Bool	Internal Memory SR
11	memory_rs	Bool	Internal Memory RS
12	Temp		
13	<Add new>		
14	Constant		
15	<Add new>		

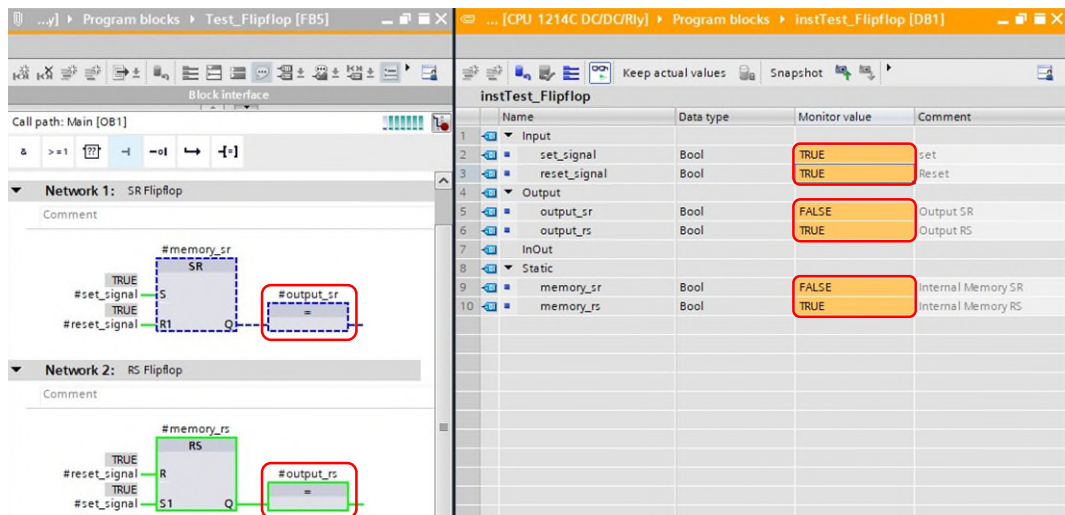
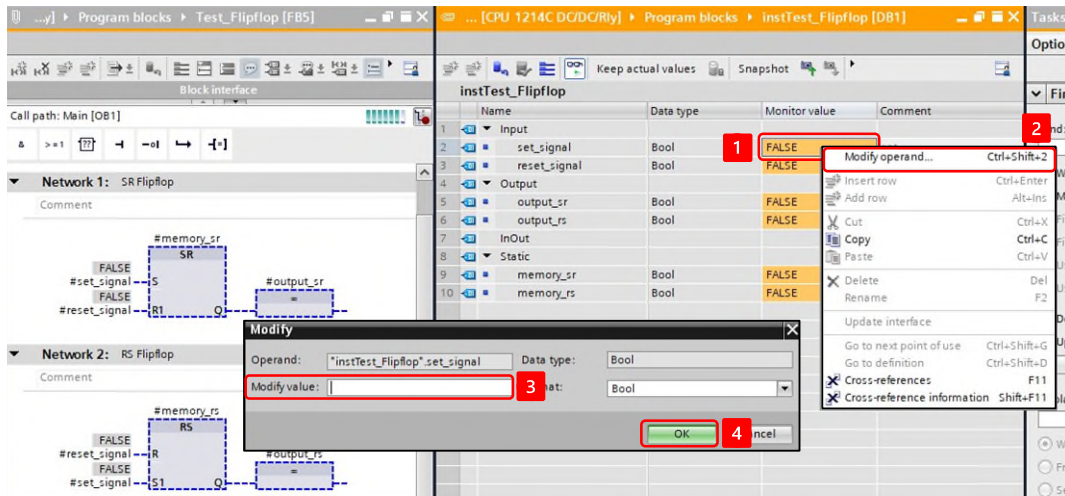
- implement the following program, the storage functions can be found in the TaskCard under "Instructions" → "Simple instructions" → "Bit links". These can be dragged and dropped into the workspace:



- call up the function block in "MAIN" and create an instance.



- Use the instance to set the "set_signal" and "reset_signal" input variables to "TRUE" one after the other, then set both input variables to "TRUE" at the same time and monitor the "output_sr" and "output_rs" output variables:



7.2 Flanks

Edges in PLC programming are important for recognizing certain events that occur when the signal changes. An edge evaluation records whether the state of a binary signal has changed compared to the previous program cycle.

According to IEC 61131, the following edges are available:

- Positive edge (R_TRIG)
- Negative edge (F_TRIG)

Flanks are required, for example, for:

Event detection:

Edges make it possible to recognize specific events that occur exactly when the signal transitions. This is useful for triggering actions exactly when a signal changes, not while it remains in a certain state.

Clock-controlled processes:

In many applications, it is necessary to start processes synchronously with certain signal changes.

Event-controlled actions:

In automation technology, certain actions often have to be started at the exact moment when a switch is actuated or a sensor detects an event. Edge detection makes this possible precisely and prevents malfunctions.

Interrupts and timings:

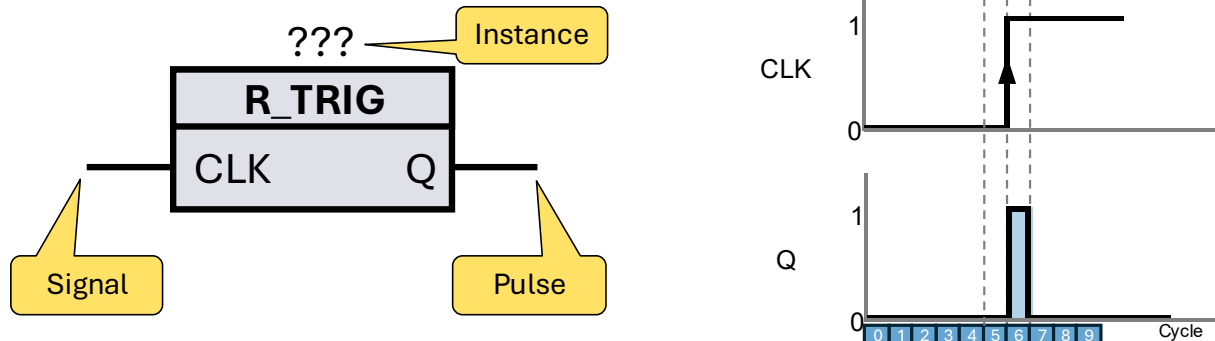
In real-time applications, edges can be used to generate interrupts in order to react immediately to external events. They also enable precise timings and time measurements.

When the signal (edge) of a binary signal changes, the edge evaluation outputs a pulse. This pulse is only present for one cycle.

To detect the signal change (edge), the current state (actual state) is compared with the previous state (past value) of the signal. In each program cycle, the old signal status (historical value) is compared with the current signal status. To do this, the evaluation requires a memory (instance).

7.2.1 Recognize positive edge R_TRIG

With the IEC instruction "R_TRIG", detect positive signal edge, you can detect a status change from "0" to "1" at input CLK.



Picture 13 FBD-instruction R_TRIG and pulse diagram

Current cycle 6

The instruction compares the current value at input CLK (cycle 6) with the status of the previous query (edge memory, cycle 5), which is stored in the instance.

- If CLK is "1" and the edge memory is "0", a positive edge has been detected.
- If the instruction has detected a change of state at input CLK from "0" to "1", output Q is set to "1".
- The edge memory is set to the signal status of the current signal. Edge memory "1".

Current cycle 7

The instruction compares the current value at input CLK (cycle 7) with the status of the previous query (edge memory, cycle 6), which is stored in the instance.

- If CLK is "1" and the edge memory is "1", no edge was detected.
- If the instruction has not detected a signal change at input CLK from "0" to "1", output Q is set to "0".
- The edge memory is set to the signal status of the current signal. Edge memory "1".

If the instruction detects a signal change at input CLK from "0" to "1", a pulse is generated at output Q, i.e. the output carries the value "1" for one cycle. In all other cases, the signal status at the output of the instruction is "0".

Each call of the "Detect positive signal edge" instruction must be assigned an instance of the "R_TRIG" data type, in which the instance data is stored.

Textual conversion Recognize positive signal edge (R_TRIG)

The R_TRIG edge module is instantiated in a similar way to a function module. A separate memory area is required for each call of the "R_TRIG" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
```

```
VAR
```

```
    instRtrig: R_TRIG; // Declaration of instance (Multi-instance)
```

```
END_VAR
```

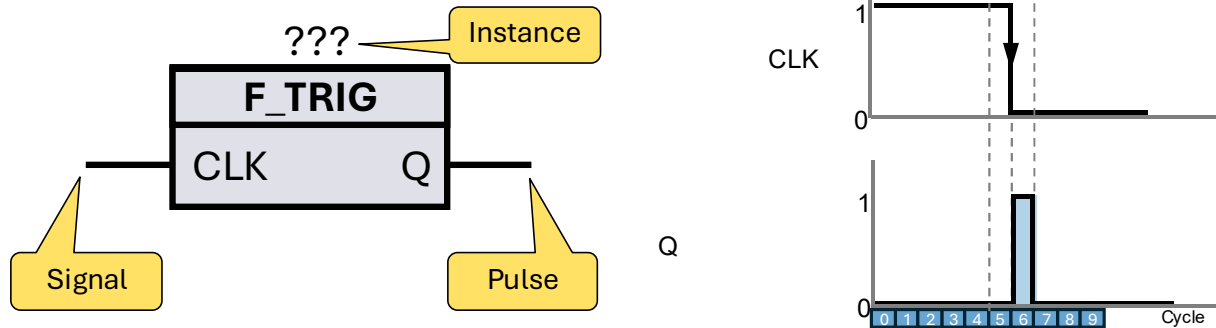
```
//Instruction, instance call
```

```
instRtrig(CLK := varBoolClk,  
          Q => varBoolQ);
```

Picture 14 ST/SCL instruction R_TRIG

7.2.2 Recognize negative edge F_TRIG

You can use the "Detect negative signal edge" instruction to detect a change of state from "1" to "0" at input CLK.



Picture 15 FBD instruction F_TRIG and pulse diagram

Current cycle 6

The instruction compares the current value at input CLK (cycle 6) with the status of the previous query (edge memory, cycle 5), which is stored in the instance.

- If CLK is "0" and the edge memory is "1", a negative edge has been detected.
- If the instruction has detected a signal change at input CLK from "1" to "0", output Q is set to "1".
- The edge memory is set to the signal status of the current signal. Edge memory "0".

Current cycle 7

The instruction compares the current value at input CLK (cycle 7) with the status of the previous query (edge memory, cycle 6), which is stored in the instance.

- If CLK is "0" and the edge memory is "0", no edge was detected.
- If the instruction has not detected a signal change at input CLK from "1" to "0", output Q is set to "0".
- The edge memory is set to the signal status of the current signal. Edge memory "0".

If the instruction detects a change of state at input CLK from "1" to "0", a pulse is generated at output Q, i.e. the output carries the value "1" for one cycle.

In all other cases, the signal status at the output of the instruction is "0".

Each call of the "Detect negative signal edge" instruction must be assigned an instance of the "F_TRIG" data type, in which the instance data is stored.

Textual conversion Recognize negative signal edge (F_TRIG)

The F_TRIG edge module is instantiated in a similar way to a function module. A separate memory area is required for each call of the "F_TRIG" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
```

```
VAR
```

```
  instFtrig: F_TRIG; // Declaration of instance (Multi-instance)
```

```
END_VAR
```

```
//Instruction, instance call
```

```
instFtrig(CLK := varBoolClk,  
          Q => varBoolQ);
```

Picture 16 ST/SCL instruction F_TRIG



Behavior after CPU startup

The IEC 61131 standard describes that the "F_TRIG" instruction sets the "Q" output to TRUE for one cycle if the "CLK" input has the value FALSE when the CPU starts up.

7.3 Times

Time functions can be used to trigger time-controlled actions in the program. They are triggered by a binary signal. The result of the evaluation is also a binary signal. Time functions are instructions that have an instance.

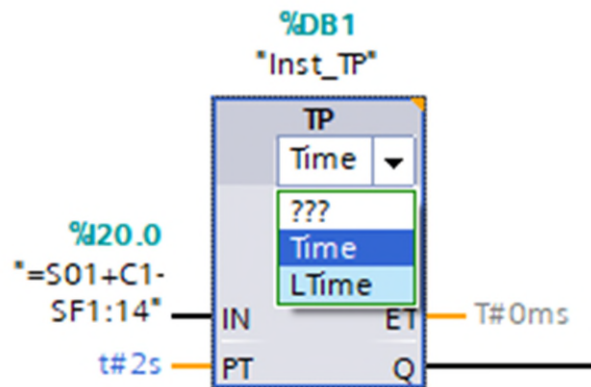
The following time functions are available in accordance with IEC 61131:

- Pulse generation (TP)
- Switch-on delay (TON)
- Switch-off delay (TOF)

The maximum time range that can be displayed depends on the selected data type.

- TIME format (32-bit) maximum time range: ~24 days
- Format LTIME (64-bit) maximum time range: ~106751 days (292 years)

In the TIA Portal, the data type can be set directly on the instruction using the drop-down menu.



Picture 17 Data type of the time function (Siemens)

In Codesys / Beckhoff, separate function blocks are available for 64-bit time operations:

- LTP
- LTON
- LTOF

Examples of fair values:

Syntax	Meaning
T#5s	5 seconds
T#1d3h5m30s500ms	1 day, 3 hours, 5 minutes, 30 seconds and 500 milliseconds
LTIME#50d3h	50 days and 3 hours

Here are some important applications and the benefits of time functions in PLC programming:

Control delays:

Time functions allow delays to be introduced into control processes. For example, the starting of a motor can be delayed after a trigger signal has been triggered.

Sequence control:

If processes have to be executed in certain time intervals, time functions ensure that each step is activated for as long as defined will.

Monitoring of time conditions:

Time functions help to monitor processes by ensuring that certain actions are completed within set time limits.

Pulse generation:

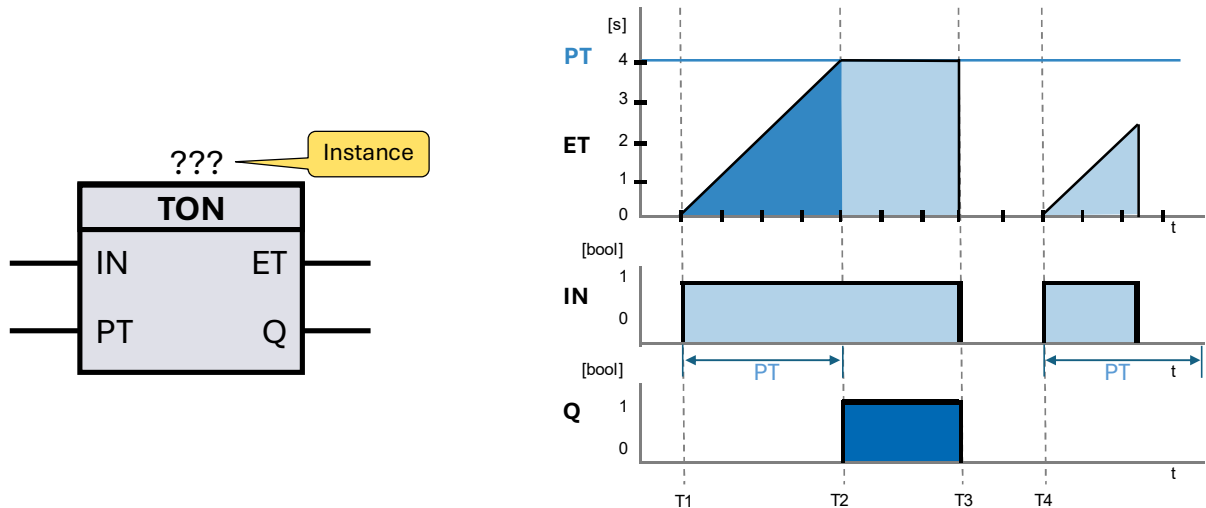
Pulsating signals are required in many applications. Time functions generate these pulses at predefined intervals.

Implement waiting times:

Time functions are useful for setting up waiting times, for example to avoid overloading the machine.

7.3.1 Switch-on delay TON

The "Generate switch-on delay" instruction delays the setting of output Q (Output) by the parameterized time PT (Preset Time). The current time value can be queried at output ET (Elapsed Time). The time value starts at T#0s and ends when the time duration PT is reached. As soon as the signal status at input IN (Input) changes to "0", output ET is reset.



Picture 18 FBD instruction TON and pulse diagram

T1

The instruction starts when the logic operation result (VKE) at input IN changes from "0" to "1" (positive signal edge). From this moment, the programmed time PT starts to run.

T2

As soon as the time period PT has elapsed, output Q is set to signal state "1".

T3

Q remains set as long as the start input is "1". If the signal status at the start input changes from "1" to "0", output Q is reset.

T4

A new positive signal edge at the start input restarts the time function.

Each call of the "Create switch-on delay" instruction must be assigned an instance of the "TON" data type, in which the instance data is stored.

Textual conversion Generate switch-on delay (TON)

The TON switch-on delay is instantiated in a similar way to a function block. A separate memory area is required for each call of the "TON" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
```

```
VAR
```

```
    instTon : TON; // Declaration of instance (Multi-instance)
```

```
END_VAR
```

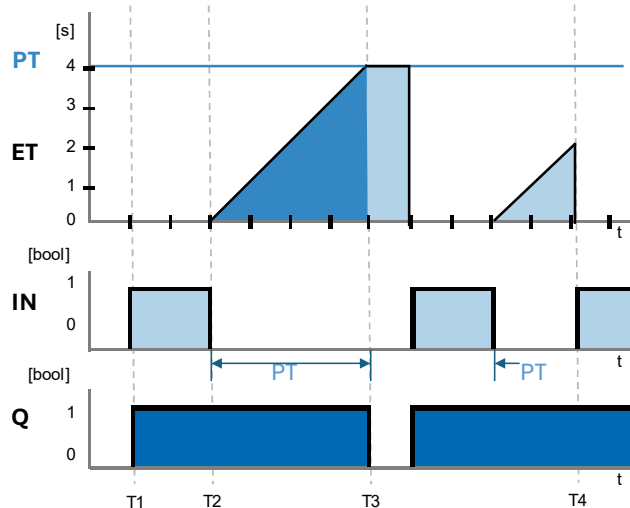
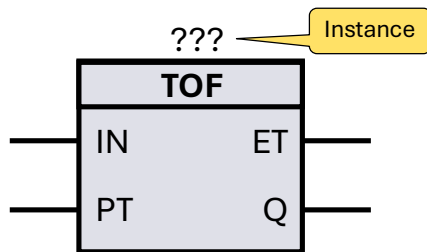
```
//Instruction, instance call
```

```
instTon(IN := varBoolIn,  
        PT := varTimePt,  
        Q => varBoolQ,  
        ET => varTimeEt);
```

Picture 19 ST/SCL instruction TON

7.3.2 Switch-off delay TOF

Use the "Generate switch-off delay" instruction to delay the resetting of output Q (Output) by the programmed time PT (Preset Time). The current time value can be queried at output ET (Elapsed Time). The time value starts at T#0s and ends when the time duration PT is reached. After the time PT has elapsed, the ET output remains at the current value until the IN input changes back to "1". If the input IN changes to "1" before the time PT has elapsed, the output ET is reset to T#0s.



Picture 20 FBD instruction TOF and pulse diagram

T1

Output Q is activated when the logic operation result (VKE) at input IN changes from "0" to "1" (positive edge).

T2

When the input IN changes back to "0" (negative edge), the programmed time PT starts to run. Output Q remains activated as long as the time PT is running.

T3

After PT has elapsed, output Q is reset.

T4

If the input signal IN changes to "1" before the time PT has elapsed, the time is reset and output Q remains activated.

Each call of the "Create switch-off delay" instruction must be assigned an instance of the "TOF" data type, in which the instance data is saved.

Textual conversion Generate switch-off delay (TOF)

The TOF switch-off delay is instantiated in a similar way to a function block. A separate memory area is required for each call of the "TOF" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
```

```
VAR
```

```
  instTof : TOF; // Declaration of instance (Multi-instance)
```

```
END_VAR
```

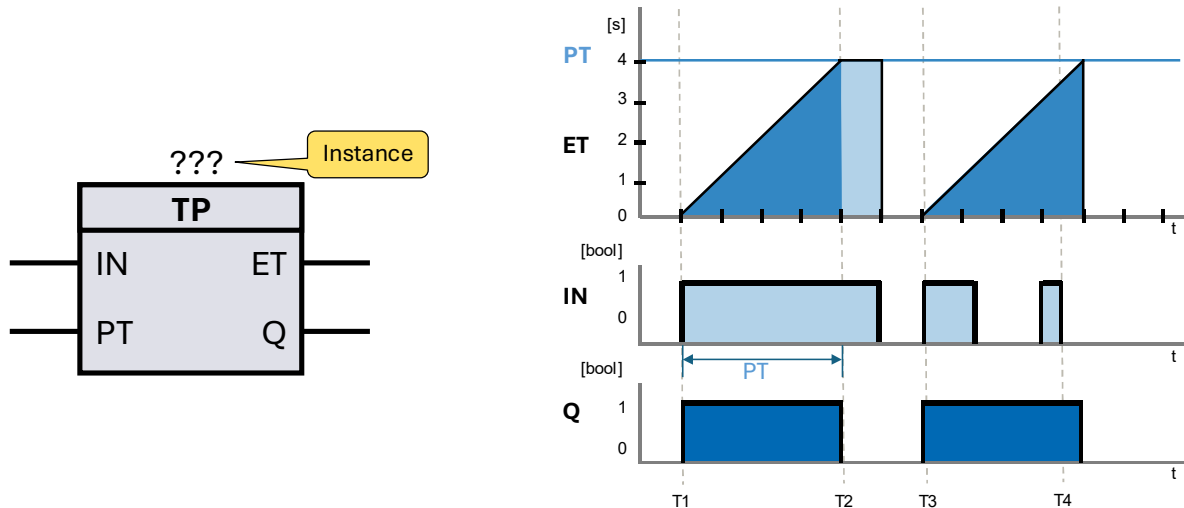
```
//Instruction, instance call
```

```
instTof(IN := varBoolIn,  
        PT := varTimePt,  
        Q => varBoolQ,  
        ET => varTimeEt);
```

Picture 21 ST/SCL instruction TOF

7.3.3 Pulse TP

Use the "Generate pulse" instruction to activate output Q (Output) for a programmed period of time. You can query the current time value at output ET (Elapsed Time). The time value starts at T#0s and ends when the value of PT (Preset Time) is reached. If PT has elapsed and the input signal IN (Input) is "0", the output ET is reset.



Picture 22 FBD instruction TP and pulse diagram

T1

When the input IN changes from "0" to "1", the programmed time PT starts to run and the output Q changes to "1".

T2

After the time PT has elapsed, the output Q changes from "1" to "0", regardless of whether the input signal IN is still "1".

T3

When the time period PT has elapsed and the input IN changes from "0" to "1", the programmed time period PT starts to run again and the output Q changes to "1".

T4

Output Q remains activated for the duration of PT, regardless of how the input signal continues to behave. While PT is running, a new positive edge at input IN has no influence on output Q and the elapsed time PT.

Each call of the "Generate pulse" instruction must be assigned an instance of the "TP" data type in which the instance data is saved.

Textual conversion Generate impulse (TP)

The TP pulse function block is instantiated in a similar way to a function block. A separate memory area is required for each call of the "TP" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

//Declaration, Interface

VAR

instTp : TP; // Declaration of instance (Multi-instance)

END_VAR

//Instruction, instance call

```
instTp(IN := varBoolIn,  
       PT := varTimePt,  
       Q => varBoolQ,  
       ET => varTimeEt);
```

Picture 23 ST/SCL instruction TP



7.3.4 Exercise - IEC time functions

Goal

In this exercise, the 3 IEC time functions (TP, TON, TOF) are programmed in a test function block and familiarized with them in practice.

The time functions can be started via the "input_signal" input parameter.

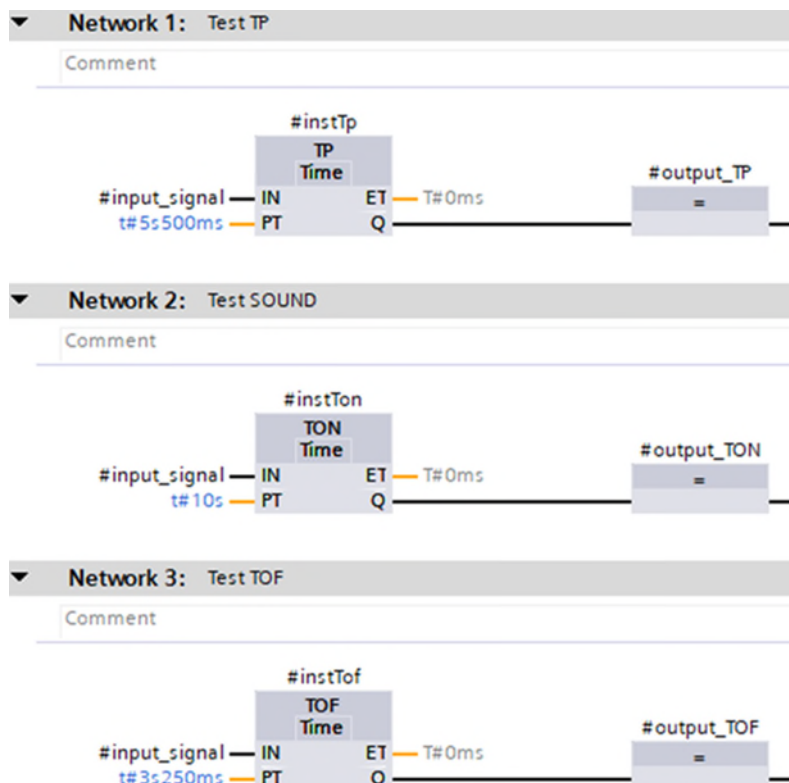
The status is displayed via the output parameters "output_TP", "output_TON", "output_TOF".

Task

- Create a function block with the following function block interface:

	Name	Data type	Comment
1	▼ Input		
2	input_signal	Bool	Test Input
3	▼ Output		
4	output_TP	Bool	Test Output TP
5	output_TON	Bool	Test Output TON
6	output_TOF	Bool	Test Output TOF
7	▼ InOut		
8	<Add new>		
9	▼ Static		
10	instTp	TP_TIME	Instance TP
11	instTon	TON_TIME	Instance TON
12	instTof	TOF_TIME	Instance TOF
13	▼ Temp		

- Depending on the selected programming language, the following program is to be implemented:




```

1 //Test TP
2 #instTp(IN := #input_signal,
3         PT := t#5s500ms,
4         //ET => ,
5         Q => #output_TP);
6
7 //Test TON
8 #instTon(IN := #input_signal,
9          PT := t#10s,
10         //ET => ,
11         Q => #output_TON);
12
13
14 //Test TOF
15 #instTof(IN := #input_signal,
16          PT := t#3s250ms,
17          //ET => ,
18          Q => #output_TOF);

```

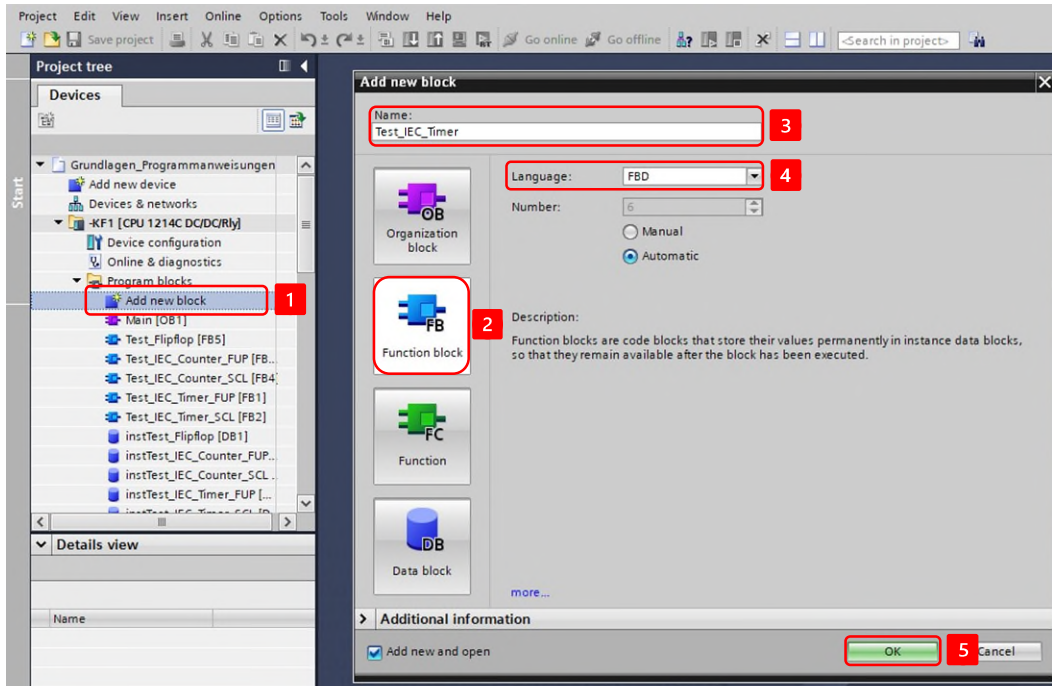
- Call up the created module in "MAIN".
- The input variables can be controlled and the output variables monitored via the instance. Alternatively, if available, these can also be connected to buttons and display elements via the function block interface when called. The passage of time can also be monitored in the individual instances of the time functions.

 Alternatively, the prepared blocks "Test_IEC_Timer_FUP[FB1]" or "Test_IEC_Timer_SCL[FB2]" from the template project "Grundlagen_Programmanweisungen.zap17" can also be used.

 The "Commissioning (software)" chapter can provide additional assistance in interpreting the program status.

Procedure:

1. create a new function block, select the desired programming language and assign a meaningful name:



2. declare the following variables in the function block interface:

	Name	Data type	Comment
1	Input		
2	input_signal	Bool	Test Input
3	Output		
4	output_TP	Bool	Test Output TP
5	output_TON	Bool	Test Output TON
6	output_TOF	Bool	Test Output TOF
7	InOut		
8	<Add new>		
9	Static		
10	instTp	TP_TIME	Instance TP
11	instTon	TON_TIME	Instance TON
12	instTof	TOF_TIME	Instance TOF
13	Temp		

- implement the following program, the time functions can be found in the TaskCard under "Instructions" → "Simple instructions" → "Times" :

The screenshot shows the SIMATIC Manager interface with three networks of IEC timer function blocks. The 'Instructions' task card is open on the right, showing the 'Timer operations' category. Red arrows point from the task card to the corresponding blocks in the networks.

Name	Description
TP	Generate pulse
TON	Generate on-delay
TOF	Generate off-delay
TONR	Time accumulator
-[TP]-	Start pulse timer
-[TON]-	Start on-delay timer
-[TOF]-	Start off-delay timer
-[TONR]-	Time accumulator
-[RT]-	Reset timer
-[PT]-	Load time duration

Network 1: Test TP
 #instTp (TP Time) with inputs #input_signal (IN) and t#5s500ms (PT), and output #output_TP (Q). ET = T#0ms.

Network 2: Test SOUND
 #instTon (TON Time) with inputs #input_signal (IN) and t#10s (PT), and output #output_TON (Q). ET = T#0ms.

Network 3: Test TOF
 #instTof (TOF Time) with inputs #input_signal (IN) and t#3s250ms (PT), and output #output_TOF (Q). ET = T#0ms.

- call up the function module in "MAIN" and create an instance:

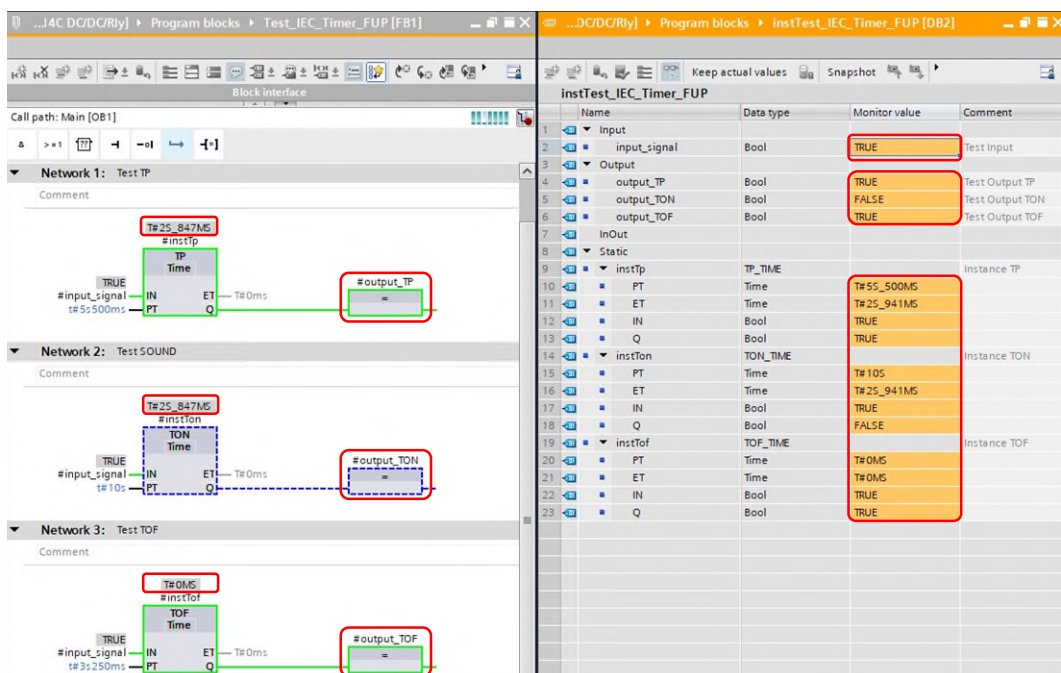
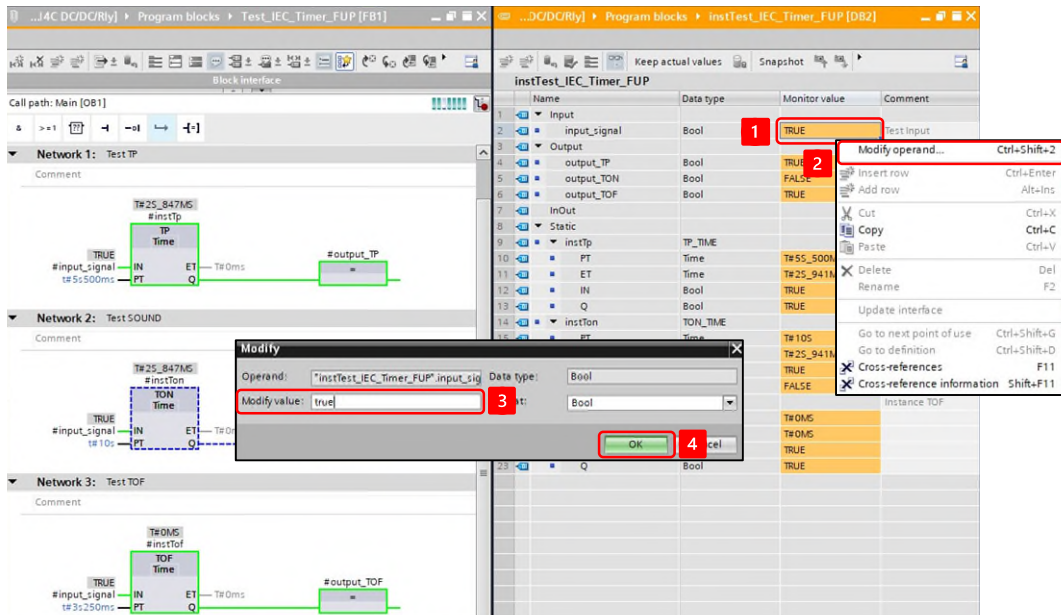
The screenshot shows the SIMATIC Manager interface with the project tree on the left, a ladder logic network in the center, and the 'Call options' dialog box on the right. Red boxes and numbers 1, 2, and 3 highlight specific elements.

Project tree: The 'Test_Timer_FUP [FB1]' block is highlighted with a red box and labeled '1'.

Network 2: A call to the 'Test_Timer_FUP' block is shown with inputs EN, input_signal, and ENO. The block is labeled with 'instTest_IEC_Timer_FUP' and 'FB1'. A red box and label '2' point to the 'Name' field in the 'Call options' dialog.

Call options dialog: The 'Name' field is set to 'instTest_IEC_Timer_FUP' (labeled '2'). The 'Number' field is set to '1'. The 'OK' button is labeled '3'.

5. control the input variable "input_signal" via the instance and observe the output variables "output_TP", "output_TON", "output_TOF", as well as the individual instances of the time functions in the corresponding instance data block:



7.4 Counter

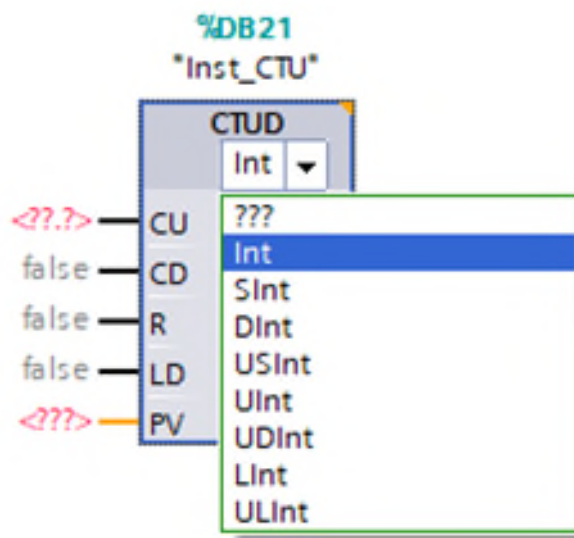
Counting functions can count a numerical value up or down in response to the pulse of a binary signal. These are instructions that require an instance.

The following counting functions are available in accordance with IEC 61131:

- Up counter (CTU)
- Down counter (CTD)
- Up and down counter (CTUD)

The maximum displayable counting range depends on the selected data type.

In the TIA Portal, the data type can be set directly on the instruction using the drop-down menu.



Picture 24 Data type of the counting function (Siemens)

Codesys has its own function blocks for 64-bit counting operations:

- LCTU
- LCTD
- LCTUD

Here are some important applications and the benefits of counting functions in the Automation technology:

Production monitoring:

Counting functions are used to monitor the number of units produced. This is particularly important in the manufacturing industry to ensure that production targets are met.

Inventory and material flow control:

They help to monitor material movements within a production facility. For example, it is possible to count how often a part passes through a particular station, which helps to optimize the material flow and control stock levels.

Security applications:

Counting functions can also be used in safety-critical applications to monitor the number of times a device or machine has been used. This can be important in order to comply with maintenance intervals.

Control of sequential processes:

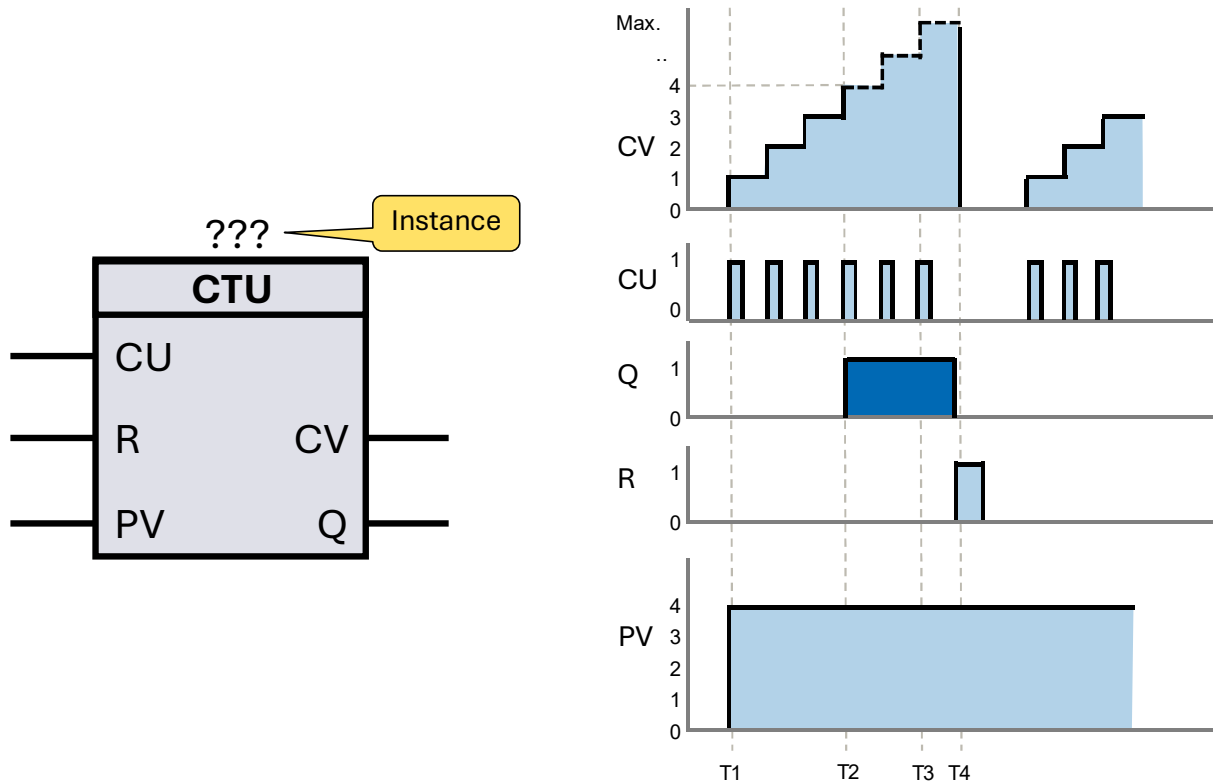
Counting functions provide precise control over the frequency with which certain actions are executed in a program. They are particularly useful in situations where a process has to be repeated several times before the program sequence moves on to the next step.

Batch processing:

In the chemical industry or in food processing, counting functions can be used to monitor the number of batches that have gone through a process, which contributes to quality control and documentation.

7.4.1 Count forward CTU

You can program an up-counter with the IEC-CTU counter. A positive edge at input CU (Count Up) increases the value at output CV (Current Value). If the value specified at input PV (Preset Value) is reached or exceeded, output Q (Output) is switched. The counter value can be set to 0 via input R (Reset).



Picture 25 FBD instruction CTU and pulse diagram

T1

If the signal at input CU changes from "0" to "1" (positive edge), the counter value at output CV is increased by one.

T2

Output Q indicates the status of the counter. The signal status of Q is determined by the parameterized setpoint PV. If the current counter value reaches or exceeds the parameterized setpoint PV, output Q is set to "1". Otherwise, the signal status of Q remains "0". A constant or a variable can be specified in the PV parameter.

T3

The counting process is repeated with each positive edge until the count value reaches the maximum value of the data type defined at output CV. Once this limit value is reached, the count value remains constant, regardless of the signal status at input CU.

T4

If there is a positive edge at input R, the counter value at output CV is reset to "0".

Each call of the "Count up" instruction must be assigned an instance of the "CTU" data type, in which the instance data is stored.

Textual conversion up counter (CTU)

The up-counter CTU is instantiated in a similar way to a function block. A separate memory area is required for each call of the "CTU" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
```

```
VAR
```

```
    instCtu : CTU; //Declaration of instance (Multi-instance)
```

```
END_VAR
```

```
//Instruction, instance call
```

```
instCtu(CU := varBoolCu,
```

```
        R := varBoolR,
```

```
        PV := varIntPv,
```

```
        CV => varIntCv,
```

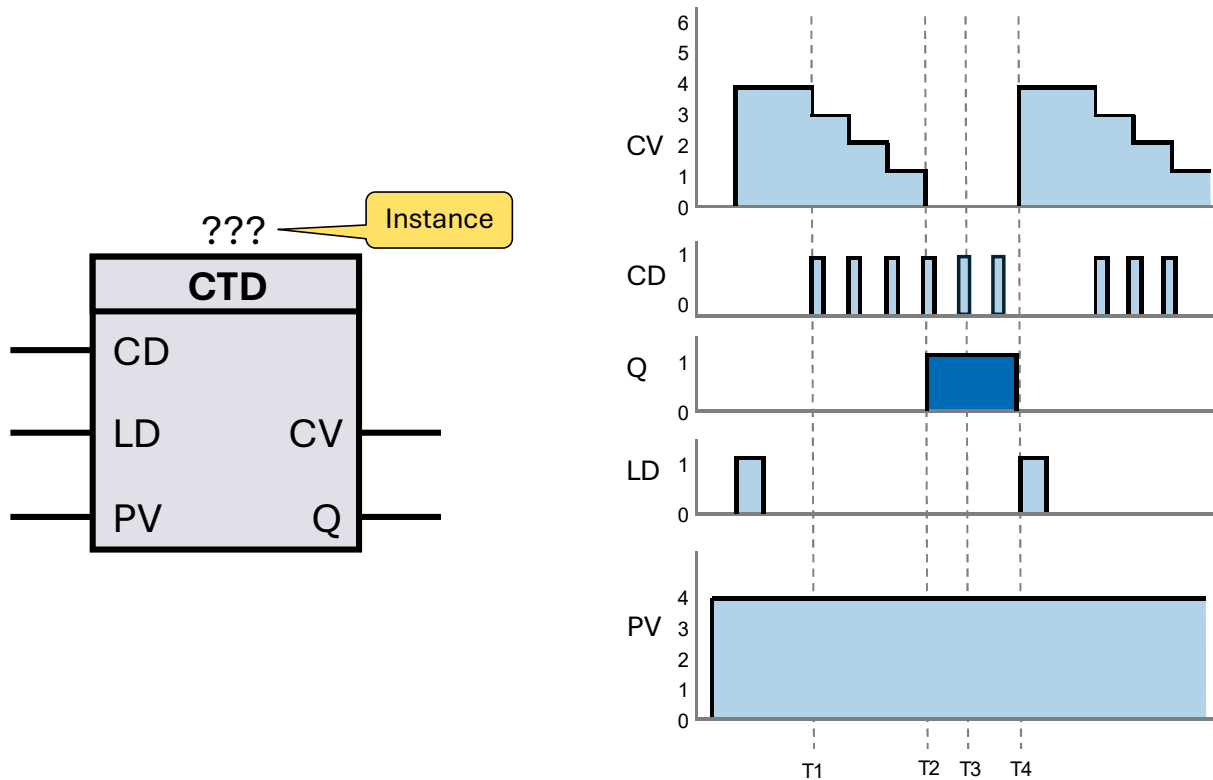
```
        Q => varBoolQ);
```

Picture 26 ST/SCL instruction CTU

7.4.2 Count down CTD

You can program a down counter with the IEC-CTD counter.

With a positive edge at input CD (Count Down), i.e. when the signal status changes from "0" to "1", the current count value at output CV (Current Value) is reduced by one. Output Q (Output) shows the status of the counter. As soon as the counter value is less than or equal to "0", output Q switches to "1". If the signal status at input LD (Load) changes from "0" to "1", the counter value at output CV is loaded to the value of parameter PV (Preset Value).



Picture 27 FBD instruction CTD and pulse diagram

T1

With a positive edge at input CD, the current count value at count value CV is reduced by one.

T2

This process is repeated with each positive edge until the count value CV reaches the lower limit value of the specified data type. If the current count value CV is less than or equal to 0, output Q is set to "1".

T3

If the lower limit value of the specified data type is reached, output Q remains at "1" and input CD has no further influence.

T4

If there is a positive edge at input LD, the value specified at input PV is loaded into the CV.

Each call of the "Count backwards" instruction must be assigned an instance of the "CTD" data type, in which the instance data is stored.

Textual conversion down counter (CTD)

The CTD down counter is instantiated in a similar way to a function block. A separate memory area is required for each call of the "CTD" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
VAR
  instCtd : CTD; //Declaration of instance (Multi-instance)
END_VAR

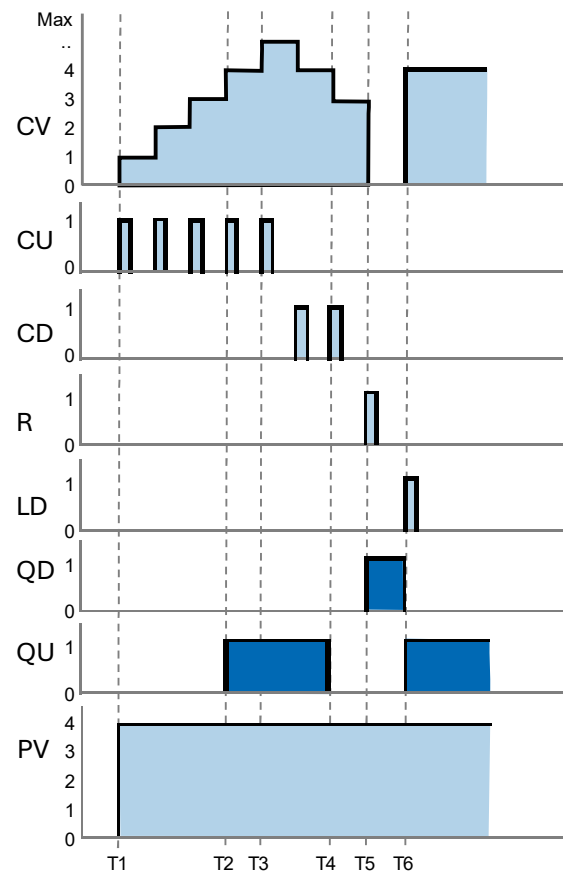
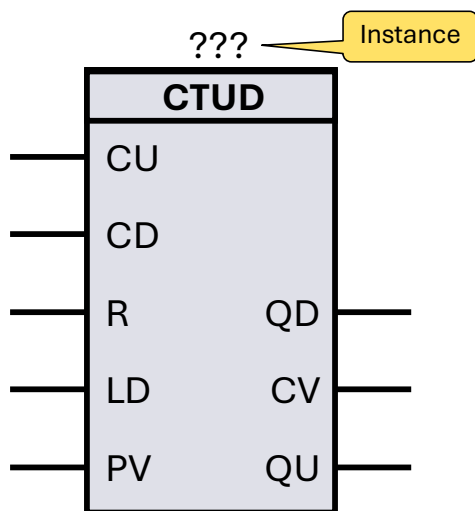
//Instruction, instance call
instCtd(CD := varBoolCd,
        LD := varBoolLd,
        PV := varIntPv,
        CV => varIntCv,
        Q => varBoolQ);
```

Picture 28 ST/SCL instruction CTD

7.4.3 Counting forwards and backwards CTUD

With the "Count up and down" instruction, the count value at output CV (Current Value) is both increased and decreased. A positive edge at input CU (Count Up) increases the count value by one, while a positive signal edge at input CD (Count Down) decreases the count value by one. A positive edge at input R (Reset) resets the CV output to 0. A positive edge at input LD (Load) loads the value specified at input PV (Preset Value) into the counter value CV. The counter has an output QU (Count Up Output), which is set if the current count value CV is greater than or equal to the value specified at input PV. The counter has a QD (Count Down Output), which is set if the current count value CV is less than or equal to 0.

i If a positive signal edge occurs at both inputs in a program cycle, the counter value remains unchanged.



Picture 29 FBD instruction CTUD and pulse diagram

T1

A positive edge at input CU increases the counter value CV by one.

T2

Output QU is set to "1" if the current counter value CV is greater than or equal to the value specified at input PV.

T3

The count value CV can be counted up to the upper or lower limit value of the specified data type.

T4

Each positive edge at input CD reduces the current count value CV by one. If the current count value CV is less than or equal to the value specified at input PV, output QD is reset.

T5

A positive edge at input R sets the current counter value to "0", whereby output QD is set.

T6

A positive edge at input LD sets the current counter value to the value specified at input PV, whereby output QD is set.

Each call of the "Count up and down" instruction must be assigned an instance of the "CTUD" data type, in which the instance data is stored.

Textual conversion of up and down counters (CTUD)

The CTUD up and down counter is instantiated in a similar way to a function block. A separate memory area is required for each call of the "CTUD" function block. The instance data can be created either as a single instance or as a multi-instance (in the function block interface).

Example:

```
//Declaration, Interface
VAR
  instCtud : CTUD; //Declaration of instance (Multi-instance)
END_VAR

//Instruction, instance call
instCtud(CU := varBoolCu,
         CD := varBoolCd,
         R := varBoolR,
         LD := varBoolLd,
         PV := varIntPv,
         QD => varBoolQd,
         CV => varIntCv,
         QU => varBoolQu);
```

Picture 30 ST/SCL instruction CTUD



7.4.4 Exercise - IEC meter

Goal

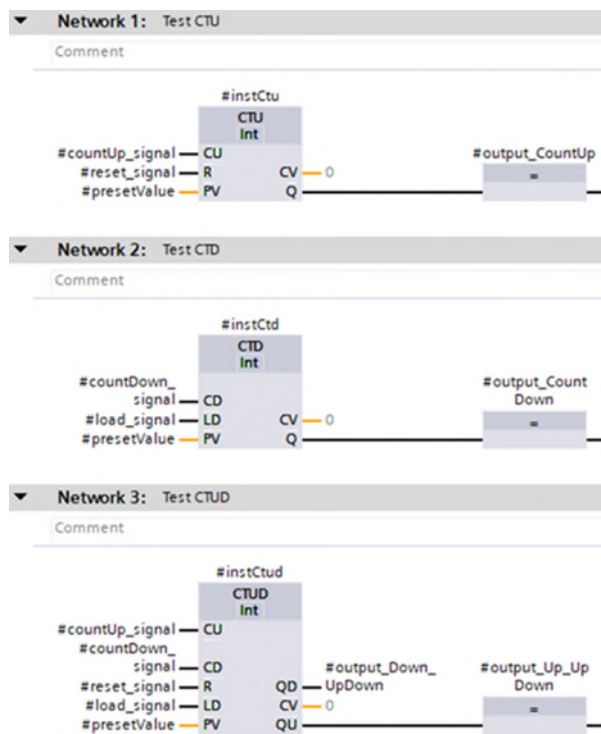
In this exercise, the 3 IEC counters (CTU, CTD, CTUD) are programmed in a test function block and familiarized with them in practice. The "countUp_signal" and "countDown_signal" input parameters can be used to positively or negatively influence the count status and set to defined values via the "reset_signal", "load_signal" and "presetValue" input parameters. The status is displayed via the output parameters "output_CountUp", "output_CountDown", "output_Up_UpDown", "output_Down_UpDown".

Task

- Create a function block with the following function block interface:

Name	Data type	Comment
Input		
countUp_signal	Bool	Test Input Count Up
countDown_signal	Bool	Test Input Countdown
reset_signal	Bool	Test Input Reset Counter
load_signal	Bool	Test Input Load
presetValue	Int	Test Input PV
Output		
output_CountUp	Bool	Test Output Q CTU
output_CountDown	Bool	Test Output QCTD
output_Up_UpDown	Bool	Test Output QU CTUD
output_Down_UpDown	Bool	Test Output QD CTUD
InOut		
<Add new>		
Static		
instCtu	CTU_INT	Instance CTU
instCtd	CTD_INT	Instance CTD
instCtud	CTUD_INT	Instance CTUD
Temp		

- Depending on the selected programming language, the following program is to be implemented:



```

2 //Test CTU
3 #instCtu(CU := #countUp_signal,
4         R := #reset_signal,
5         PV := #presetValue,
6         //CV => ,
7         Q => #output_CountUp);
8
9 //Test CTD
10 #instCtd(CD := #countDown_signal,
11         LD := #load_signal,
12         PV := #presetValue,
13         //CV=> ,
14         Q => #output_CountDown);
15
16 //Test CTUD
17 #instCtud(CU := #countUp_signal,
18         CD := #countDown_signal,
19         R := #reset_signal,
20         LD := #load_signal,
21         PV := #presetValue,
22         //CV=> ,
23         QU => #output_Up_UpDown,
24         QD => #output_Down_UpDown);

```

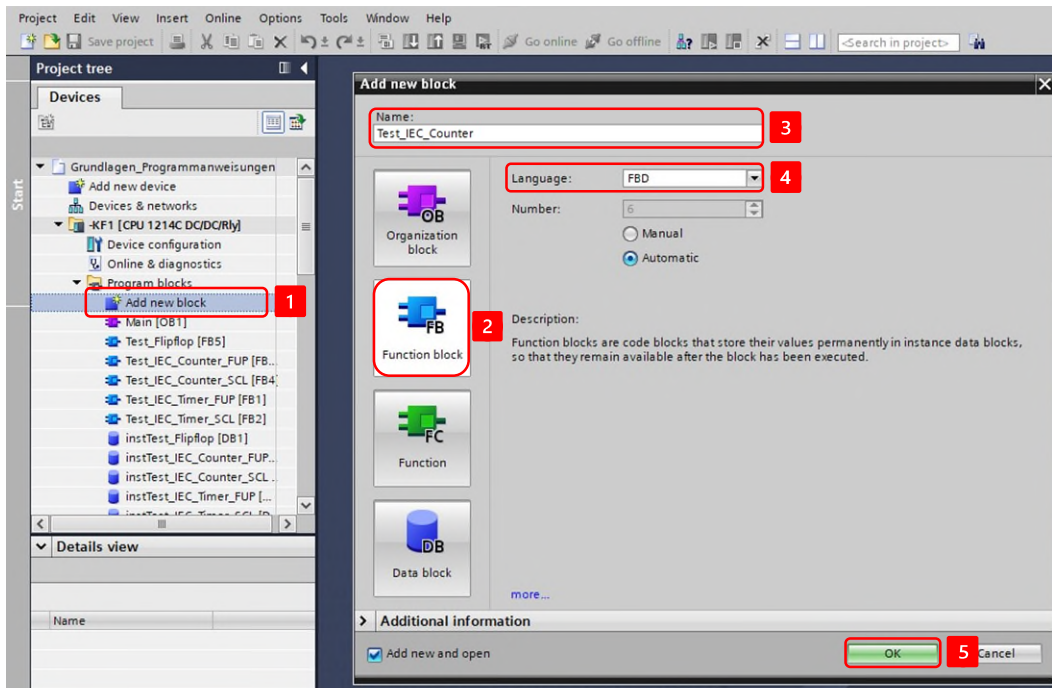
- Call up the created module in "MAIN".
- The input variables can be controlled and the output variables monitored via the instance. Alternatively, if available, these can also be connected to buttons and display elements via the function block interface when called. The count can also be monitored in the individual instances of the counters.

 Alternatively, the prepared blocks "Test_IEC_Counter_FUP[FB3]" or "Test_IEC_Counter_SCL[FB4]" from the template project "Grundlagen_Programmanweisungen.zap17" can also be used.

 The "Commissioning (software)" chapter can provide additional assistance in interpreting the program status.

Procedure:

1. create a new function block, select the desired programming language and assign a meaningful name:



2. declare the following variables in the function block interface:

	Name	Data type	Comment
1	Input		
2	countUp_signal	Bool	Test Input Count Up
3	countDown_signal	Bool	Test Input Countdown
4	reset_signal	Bool	Test Input Reset Counter
5	load_signal	Bool	Test Input Load
6	presetValue	Int	Test Input PV
7	Output		
8	output_CountUp	Bool	Test Output Q CTU
9	output_CountDown	Bool	Test Output QCTD
10	output_Up_UpDown	Bool	Test Output QU CTUD
11	output_Down_UpDown	Bool	Test Output QD CTUD
12	InOut		
13	<Add new>		
14	Static		
15	instCtu	CTU_INT	Instance CTU
16	instCtd	CTD_INT	Instance CTD
17	instCtud	CTUD_INT	Instance CTUD
18	Temp		

- implement the following program, the counting functions can be found in the TaskCard under "Instructions" → "Simple instructions" → "Counter" :

The screenshot shows the SIMATIC Manager interface. On the left, three networks are visible: Network 1: Test CTU, Network 2: Test CTD, and Network 3: Test CTUD. Each network contains a counter block (CTU, CTD, and CTUD respectively) with its associated inputs and outputs. On the right, the 'Instructions' task card is open, showing a tree view of instructions. Red arrows point from the 'CTU', 'CTD', and 'CTUD' entries in the 'Counter operations' folder to their respective blocks in the networks.

- call up the function block in "MAIN" and create an instance:

The screenshot shows the SIMATIC Manager interface with the 'Main [OB1]' network selected. A call to the 'Test_IEC_Counter_FUP' function block is visible in Network 3. A 'Call options' dialog box is open, showing the 'Data block' field set to 'instTest_IEC_Counter_FUP'. Red numbers 1, 2, and 3 are placed on the image to indicate the steps: 1 points to the function block in the project tree, 2 points to the 'Name' field in the dialog, and 3 points to the 'OK' button.

- Use the instance to control the input variables and monitor the output variables and the individual instances of the counting functions in the corresponding instance data block:

The screenshot displays the SIMATIC Manager interface for configuring an IEC Counter function block. The left pane shows three networks (Test CTU, Test CTD, Test CTUD) with their respective ladder logic diagrams. The right pane shows the 'instTest_IEC_Counter_FUP' data block with a table of variables and their monitor values. A 'Modify' dialog box is open, showing the modification of the 'count' variable to 'true'. A context menu is also visible over the data block table.

Name	Data type	Monitor value	Comment
Input			
countUp_signal	Bool	TRUE	Test Input Count Up
countDown_signal	Bool	FALSE	Test Input CountDown
reset_signal	Bool	FALSE	Test Input Reset Counter
load_signal	Bool	FALSE	Test Input Load
presetValue	Int	5	Test Input PV
Output			
output_CountUp	Bool	TRUE	Test Output Q CTU
output_CountDown	Bool	TRUE	Test Output Q CTD
output_Up_UpDown	Bool	FALSE	Test Output QU CTUD
output_Down_UpDown	Bool	FALSE	Test Output QD CTUD
InOut			
Static			
instCtu	CTU_INT		Instance CTU
CU	Bool	TRUE	
CD	Bool	FALSE	
R	Bool	FALSE	
LD	Bool	FALSE	
OU	Bool	TRUE	
OD	Bool	FALSE	
PV	Int	5	
CV	Int	-4	
instCtd	CTD_INT		Instance CTD
CU	Bool	FALSE	
CD	Bool	FALSE	
R	Bool	FALSE	
LD	Bool	FALSE	
OU	Bool	FALSE	
OD	Bool	TRUE	
PV	Int	5	
CV	Int	-4	
instCtud	CTUD_INT		Instance CTUD
CU	Bool	TRUE	
CD	Bool	FALSE	
R	Bool	FALSE	
LD	Bool	FALSE	
OU	Bool	FALSE	
OD	Bool	FALSE	
PV	Int	5	
CV	Int	4	

7.5 IF statement [ST / SCL]

In the ST / SCL programming language, the IF statement is used to control conditional sequences. This statement can be used to control the program flow depending on certain conditions. The use of IF, ELSE and ELSIF statements in SCL is described below.

7.5.1 IF...THEN - Instruction

The IF...THEN statement (Execute conditionally) is used to execute a statement block depending on a condition.

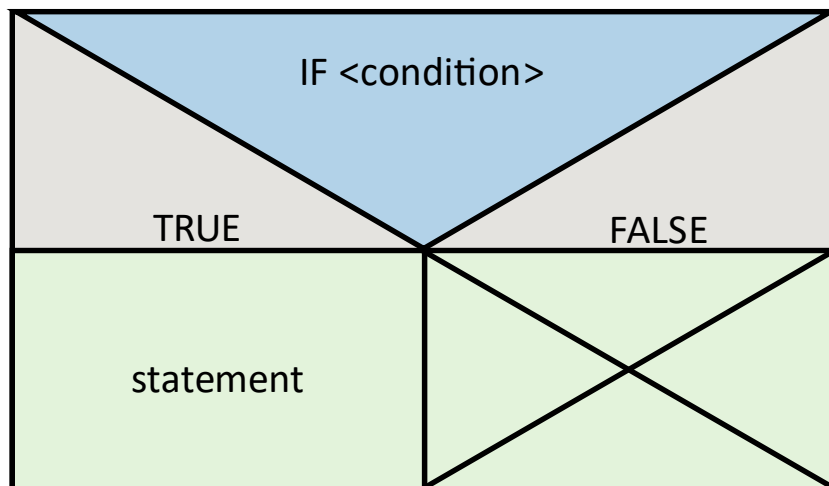
The basic structure of this instruction is as follows:

```
IF (*condition*) THEN
  //statement

END_IF;
```

Picture 31 Code example IF...THEN statement

Structure program



Picture 32 Structure chart IF...THEN statement

Example

```
IF Temperatur > 100 THEN
  Alarm := TRUE;
END_IF;
```

Picture 33 Example IF...THEN statement

In this example, the "Alarm" variable is set to "TRUE" if the "Temperature > 100" condition is met.

7.5.2 IF...THEN...ELSE - statement

The IF...THEN...ELSE statement (conditional branching) is used to simply branch a statement block depending on a condition.

The ELSE statement is used to execute alternative statements if the condition of the IF statement is not fulfilled.

The basic structure of this instruction is as follows:

```

IF (*condition*) THEN
  //statement 1

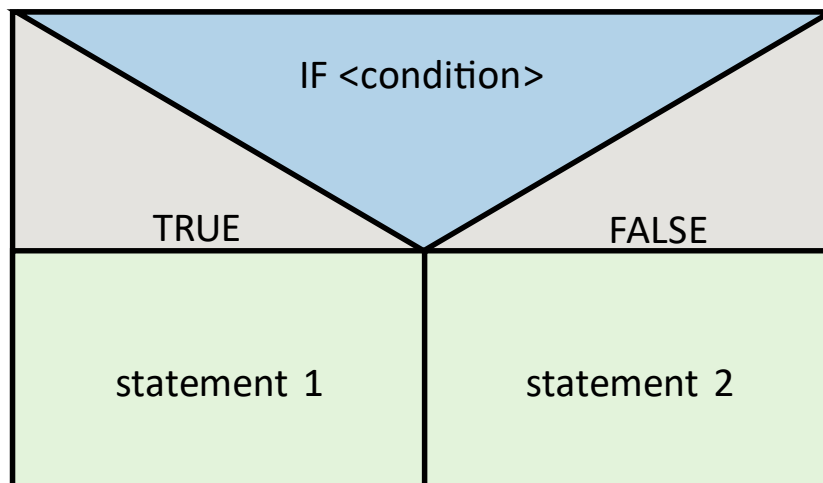
ELSE
  //statement 2

END_IF;

```

Picture 34 Code example IF...THEN...ELSE statement

Struktogramm



Picture 35 Structure chart IF...THEN...ELSE statement

Example

```

IF Temperatur > 100 THEN
  Alarm := TRUE;
ELSE
  Alarm := FALSE;
END_IF;

```

Picture 36 Example IF...THEN...ELSE statement

In this example, the "Alarm" variable is set to "TRUE" if the "Temperature > 100" condition is met. Otherwise, "Alarm" is set to "FALSE".

7.5.3 IF...THEN...ELSIF - Instruction

The IF...THEN...ELSIF statement (multiple conditional branching) is used to branch a statement block depending on several conditions.

The ELSIF statement makes it possible to check several conditions in one IF statement. If the first condition is not fulfilled, the next condition is checked, and so on.

The basic structure of this instruction is as follows:

```

IF (*condition 1*) THEN
  //statement 1

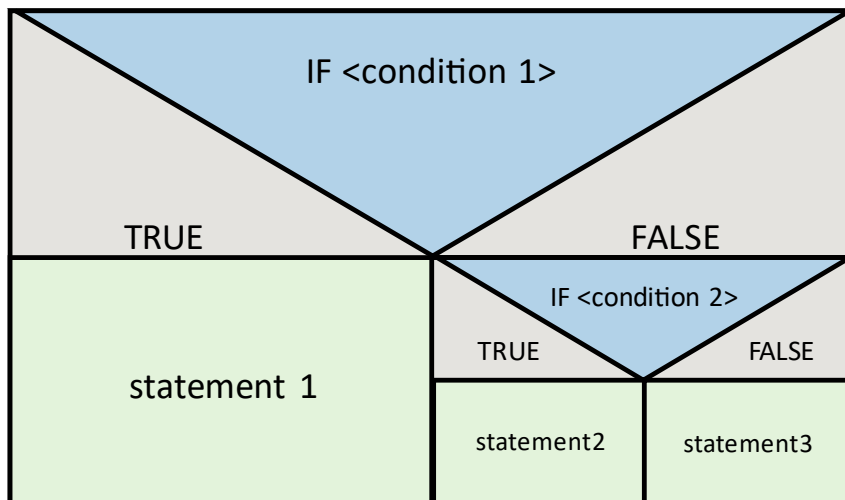
ELSIF (*condition 2*) THEN
  //statement 2

ELSE
  //statement 3

END_IF;
    
```

Picture 37 Code example IF...THEN...ELSIF statement

Structure program



Picture 38 Structure chart IF...THEN...ELSE statement

Example:

```

IF Temperatur > 100 THEN
  Alarm := TRUE;
ELSIF Temperatur > 80 THEN
  Warnung := TRUE;
ELSE
  Alarm := FALSE;
  Warnung := FALSE;
END_IF;
    
```

Picture 39 Example IF...THEN...ELSE statement

In this example:

- The "Alarm" variable is set to "TRUE" if "Temperature > 100".
- The "Warning" variable is set to "TRUE" if the temperature is greater than 80 but less than or equal to 100.
- Otherwise, "Alarm" and "Warning" are set to "FALSE".

7.6 CASE structure [ST / SCL]

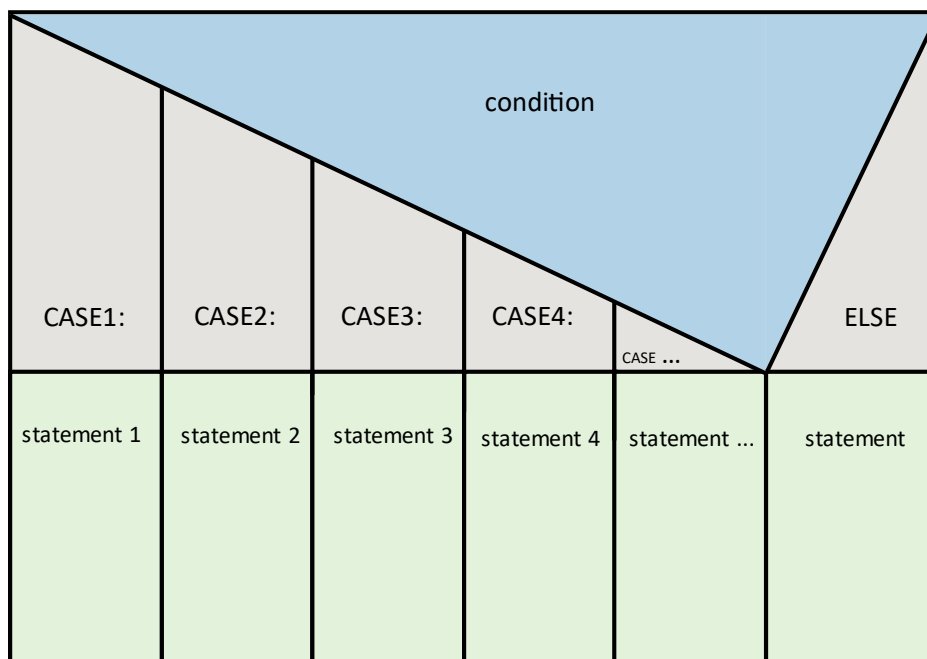
With the CASE statement (multiple branching / case differentiation), you process one of several statement sequences, depending on the value of a numerical expression.

The value of the expression must be an integer. When the instruction is executed, the value of the expression is compared with the values of several constants. If the value of the expression matches the value of a constant, the instructions that are programmed directly after this constant are executed.

The constants can assume the following values:

- An integer (e.g. 3)
- A range of integers (e.g. 5...8)
- An enumeration of integers and ranges (e.g. 11, 17... 25)
- One bit sequence (0001)

Depending on the value of the expression, one of the following alternatives (CASE1 ... CASEN) is selected and the corresponding statement sequence (statement 1 to statement N) is executed. If none of the alternatives apply, the ELSE branch is executed if it is available.



Picture 40 CASE instruction structure diagram

The syntax of a CASE instruction is as follows:

```

CASE (*condition*) OF
1: //CASE 1
   //statement 1

2: //CASE 2
   // statement 2

3: //CASE 3
   // statement 3

4: //CASE 4
   // statement 4

ELSE
   // statement

END_IF;
    
```

Picture 41 Syntax CASE statement

The CASE structure also supports the specification of value ranges and the grouping of multiple values.

```

CASE (*Variable*) OF
1..9: //Variable 1 - 9
     // statement

10, 15, 19: //Variable 10, 15, 19
           // statement

ELSE
     // statement

END_IF;
    
```

Picture 42 CASE statement with multiple selection

In this example, the variable 'Age' is checked and the category is assigned depending on the value range.

```
CASE age OF
0..12: //Child
    Kategorie := 'child';

13..19: //Teenager
    Kategorie := 'teenager';

20..64: //Adult
    Kategorie := 'adult';

65..100: //Senior
    Kategorie := 'senior';

ELSE
    Kategorie := 'invalid';

END_IF;
```

Picture 43 Example CASE

The CASE structure enables a clear and efficient method for selecting between several alternatives based on the value of a variable. It improves the readability and maintainability of the code, especially when many conditions need to be checked, compared to the IF statement.

The option to specify value ranges and multiple values per case offers additional flexibility.