INGENIEURBÜRO FÜR
TECHNOLOGIETRANSFER
DIPL.-ING. B. P. SCHULZ-HEISE

Training Manual

STEP® 5 with *S5 for Windows*®

Basic Training

TTI Ingenieurbüro für
Technologie Transfer
Dipl. Ing. B. Peter Schulz-Heise

Stadtring 207
64720 Michelstadt, Germany

Tel.:	**06061 3382**	**Home page:**	**TTIntl.com**
Fax:	**06061 71162**	**E-Mail:**	**PSH@TTIntl.com**

Table of Contents

Table of Contents 1

1 Basic S5 Programming 1-1

1.1 Methods of Representation 1-1

Ladder Diagram 1-1
Ladder diagram representation on the monitor 1-7
Symbolic Programming 1-8
Statement List (STL) 1-9
Structure of a statement 1-9
Function Block Call (STL presentation) 1-10
Control System Flowchart 1-13
Calling a Function Block 1-16

1.2 Structure of the Application Program 1-17

Blocks 1-17
Organization blocks (OBs) 1-18
Program blocks (PBs) 1-18
Function blocks (FBs, FXs) 1-18
Sequence blocks (SBs) 1-18
Data blocks (DBs, DXs) 1-19

1.3 Segment 1-19

1.4 PLC Program Structures 1-20

1.5 Linear Programs 1-20

1.6 Partitioned Program 1-21

1.7 Structured programs 1-22

1.8 Example of a program structure 1-24

1.9 Cyclic Program Processing 1-26

The Cyclic PLC Program Execution 1-28

1.10 CPU Start-up 1-29

RESTART OB's 1-29
Cold Restart Routine 1-29
Restart Characteristics and Cyclic Operation 1-31

1.11 Cyclic Program Processing 1-32

The Cyclic PLC Program Execution 1-34

1.12 Organization Blocks for Interrupt-Driven Program Execution 1-35

Non-Cyclic Program Execution 1-35
Cyclic Interrupt Organization Blocks (OB10 to OB18) 1-37
Interrupt Driven Program Scanning 1-38
Overview of the System Interrupt OBs default settings 1-38

Error Handling Organization Blocks 1-38

2 Statement List Instructions Structure 2-1

2.1 STEP 5 Operands 2-4

2.2 Operands, Addressing Overview 2-7

Operands Addressing 2-7
Bit Variables (Bit Operands) 2-9
Byte Variable (Byte Operands) 2-11
Word Variable (Word Operands) 2-13
High Byte and Low Byte in a Word 2-14
Double Word Variable 2-15
Byte Order in a Double Word Variable 2-17
Overlapping of Variables 2-19

2.3 Symbolic Programming 2-21

Symbolic Table Format 2-21

2.4 Block Calls 2-25

Unconditional Call 2-25
Practice Exercise 2–1; Unconditional Call (JU) 2-28
Conditional Call 2-29
Practice Exercise 2–2; Conditional Call (JC) 2-32
Calling Organization Blocks 2-33
Calling Program Blocks 2-33
Calling Sequence Blocks 2-34
Calling Function Blocks 2-34

2.5 Block End (BE) 2-35

Block End Unconditional (BEU) 2-37
Practice Exercise 2–3; Conditional Call, BEU 2-38
Block End Conditional (BEC) 2-39
Practice Exercise 2–4; Conditional Call, BEC 2-40

3 Bit Logic Instructions 3-1

Binary Logical Instructions 3-1
Combinations of the Logical Instructions 3-2
Processing the Result of a Logic Operation 3-2
First Scan instruction 3-4
Practice Exercise 3–1; Result of the Logic Operation, Status 3-6
RLO delimiting 3-7
RLO delimiting Instructions 3-7

3.1 Basic Rules of Boolean Algebra 3-8

Conversion AND / OR 3-8
Conversion OR / AND 3-8
Example of a Logical Connection 3-10
A AND Function 3-11
Practice Exercise 3–2; Logical AND 3-13
Practice Exercise 3–3; Logical OR 3-16
NAND Function 3-18

NOR Function 3-19
Practice Exercise 3–4; Conveyer Belt, Package Height 3-20
AND before OR 3-22
Practice Exercise 3–5; AND before OR 3-24
OR before AND 3-25
Practice Exercise 3–6; OR before AND 3-27
Practice Exercise 3–7; Normally Open (NO), Normally Closed (NC) 3-28
Converting a relay logic into a PLC Program 3-29
Using the LAD Editor 3-31
Practice Exercise 3–8; Motor right/left 3-35

3.2 Number Systems 3-36

Decimal system 3-36
Binary Numbers 3-38
Hexadecimal Numbers 3-39
The link between binary numbers and hexadecimal numbers 3-39
BCD numbers 3-41
The link between binary, BCD, and hexadecimal numbers 3-41
Practice Exercise 3–9; Seven Segment Display 3-44

3.3 Setting / Resetting Bit Addresses 3-47

S – Set instruction 3-47
R – Reset instruction 3-49
RS Flip Flop 3-50
SR Flip Flop 3-51
Practice Exercise 3–10; Latch 3-52

3.4 Edge Detection 3-53

Positive Edge Detection 3-54
Negative Edge Detection 3-56
Practice Exercise 3–11; Motor ON/OFF, Edge Detection with Latch 3-58

4 Timing Functions (Timer) and Counters 4-1

4.1 Timing Functions (Timer) 4-1

Timer signals overview 4-1
Area in Memory 4-2
Enable Timer – FR (Free) 4-5
Pulse Timer (SP) 4-7
Extended Pulse Timer (SE) 4-10
On-Delay Timer (SD) 4-12
Retentive On-Delay Timer (SS) 4-15
Off-Delay Timer (SF) 4-17
Selecting the right Timer 4-19
Practice Exercise 4–1; Flashing Light 4-20
Practice Exercise 4–2; Traffic Light 4-21
Picture Block; Editor 4-23
Picture Block; Status Display 4-24

4.2 Counter Instructions 4-25

Enable Counter FR (Free) 4-25
Set Counter S (Preset Counter) 4-28

Load Current Counter Value (L) into ACCU 1 in Binary Form 4-28
Load Current Counter Value (LC) into ACCU 1 in BCD Form 4-30
Counter Up (CU) 4-31
Counter Down (CD) 4-32
Practice Exercise 4–3; Counter 4-34

5 Function Blocks (FB; FX) and Data Blocks (DB; DX)..... 5-1

5.1 Programming Function Blocks 5-1

Function Blocks Without Block Parameters 5-2
Function Blocks With Block Parameters 5-2
Block Parameters 5-3
Parameter type 5-5
Data type 5-6
Block Parameters (Formal Operands) defined in a FB 5-8
Calling a Function Block with Parameters (graphic presentation) 5-8
Calling a Function Block with Parameters (STL) 5-10

5.2 Data Blocks 5-11

Calling Data Blocks 5-11
Opening another Data Block in a called Block 5-14
Creating a Data Block (DB, DX) 5-15
Changing the Data Word Format 5-16
Creating a Data Block (DB, DX) automatically 5-17
Function Block (FB) with Data Block (DB) 5-19
Practice Exercise 5–1; Hysteresis, Function Block with Data Block 5-20

1 Basic S5 Programming

1.1 Methods of Representation

In STEP 5, a task definition can be formulated using three different methods of representation:

- Ladder diagram (LAD)
- Statement list (STL)
- Control system flowchart (CSF)

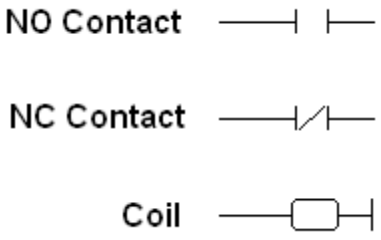
The three methods of representation are discussed briefly in the following subsections.

Ladder Diagram

In ladder diagrams, the control task is defined using symbols similar to those used in circuit diagrams. Programs can be entered, modified and documented as ladder diagrams. This method of representation also enables the output of dynamic status displays during on-line testing.

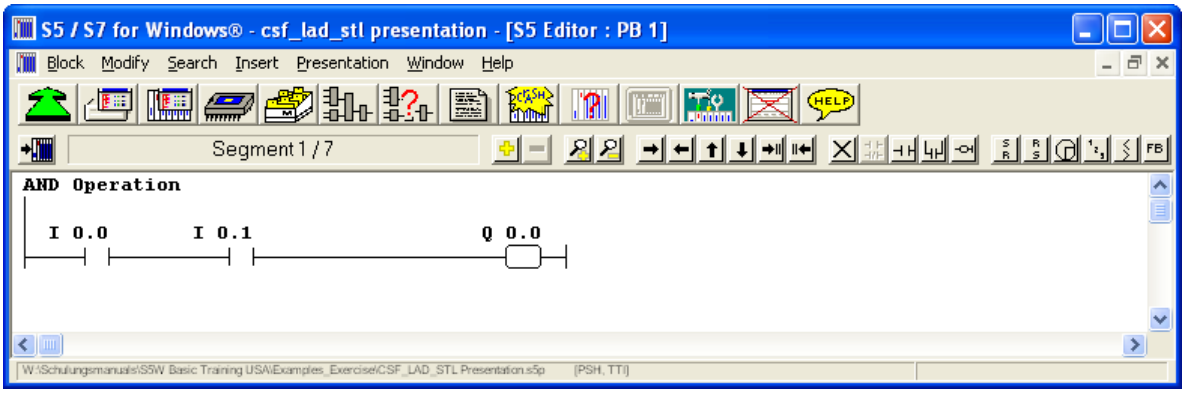
Symbols used in ladder diagrams

The symbols used in ladder diagrams are similar to those used in circuit diagrams. They are represented on the screen by unbroken lines and in printouts by characters from the printer's standard character font. Brackets (as per the standard "American" conventions) are used as symbols for NO and NC contacts, and parentheses as coil symbol for a contactor or relay:



As in circuit diagrams, the "contacts" can be interconnected both in series and in parallel. The symbol for the coil is located at the end of the "rung"

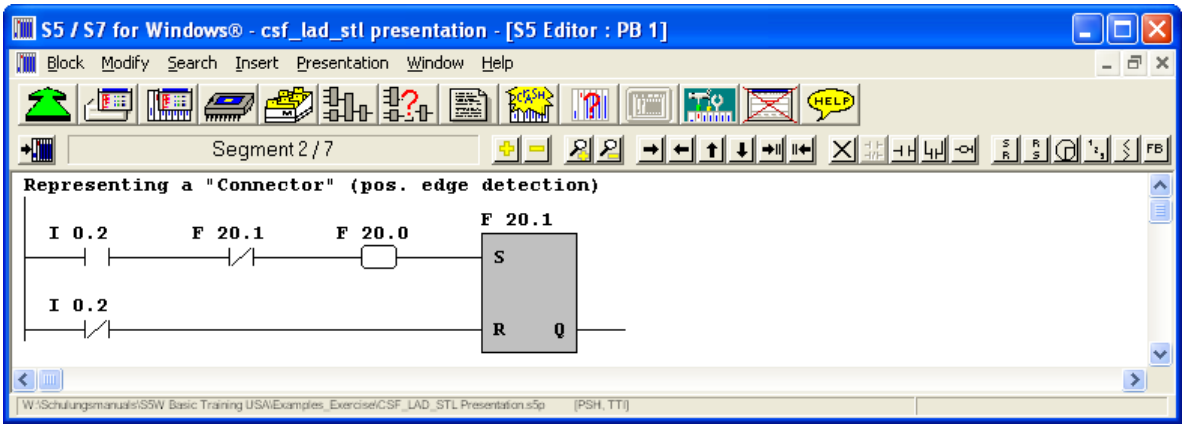
Example: Representing a series connection



The binary logic operations represented in a ladder diagram are interconnecting structures of, NO and NC contacts in series and parallel circuits. The coil symbol for a result assignment terminates the rung. It is thus possible to represent individual set/reset operations and conditional block calls (except for those relating to function blocks – these are automatically displayed in Statement List presentation).

A special case is the “connector”, which represents a result assignment within a logic operation and is identified by the symbol like a coil.

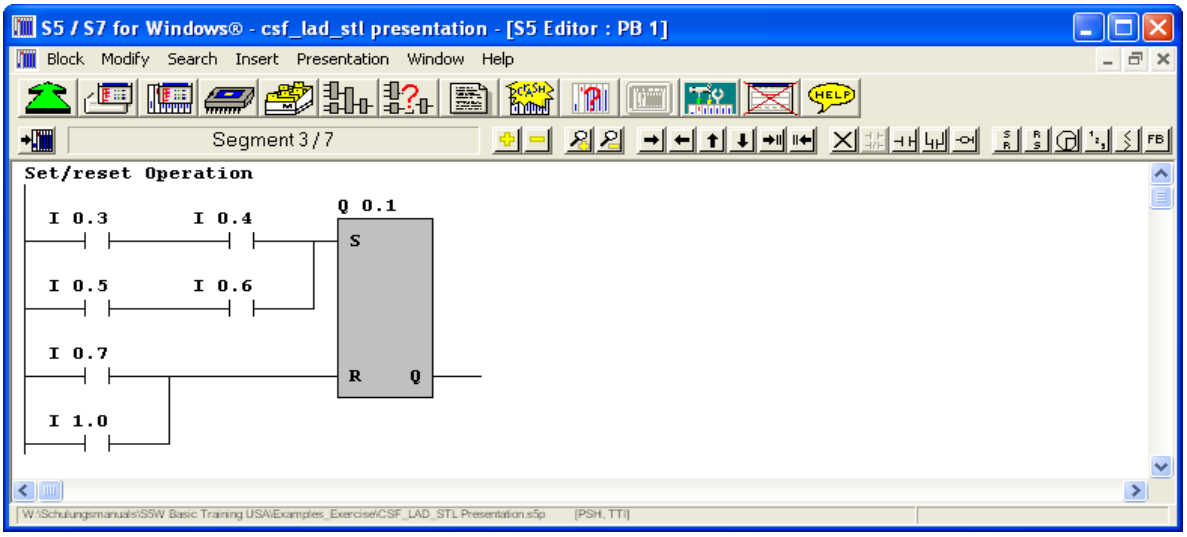
Example: Representing a “connector” in a ladder diagram



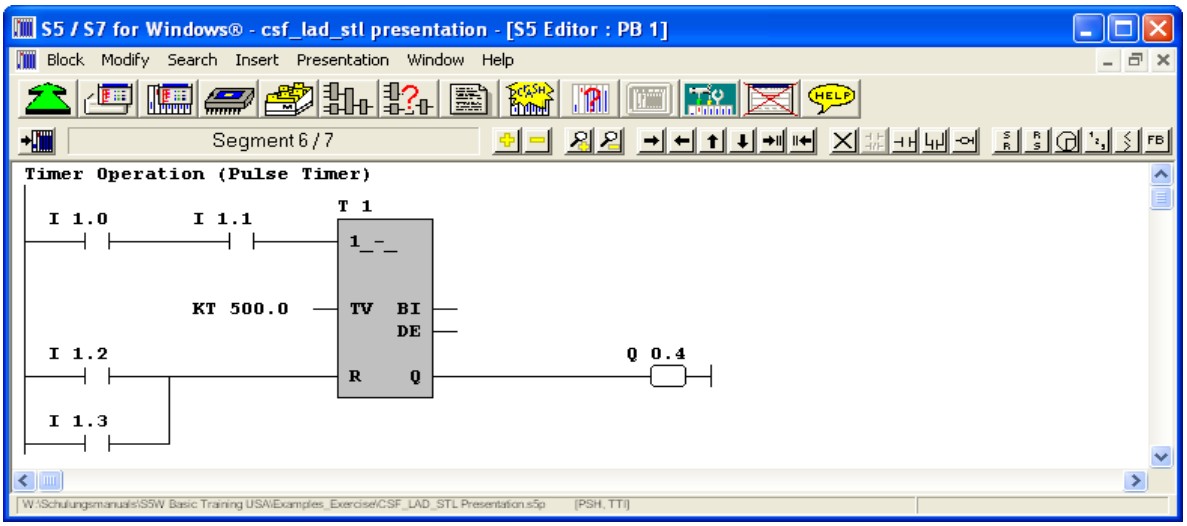
Complex operations are represented as boxes. A box contains the symbol identifying the relevant operation. “Rungs” lead to the function symbol’s inputs from the left, and “rungs” can be connected to the function symbol’s outputs at the right. “Complex” operations include set/reset, timer, and counters and compare operations.

Examples of “complex” operations in ladder diagrams

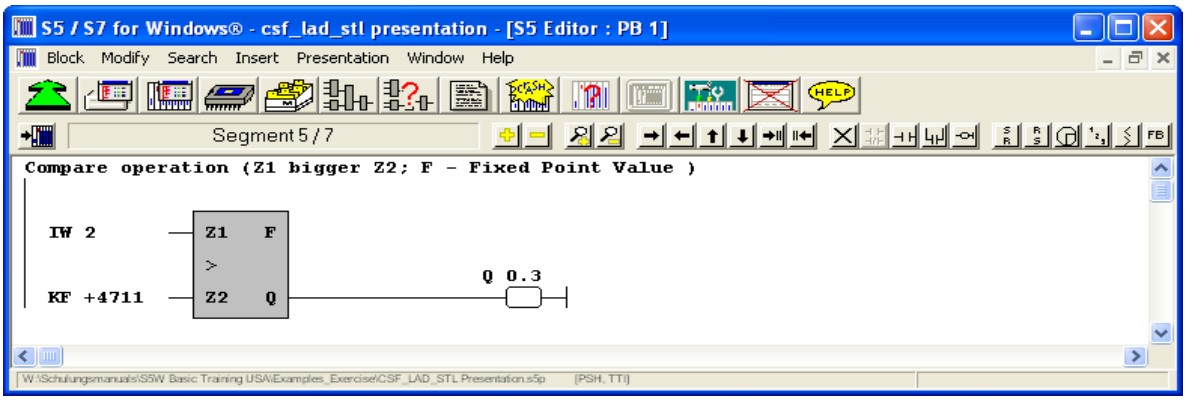
Set/reset operation:



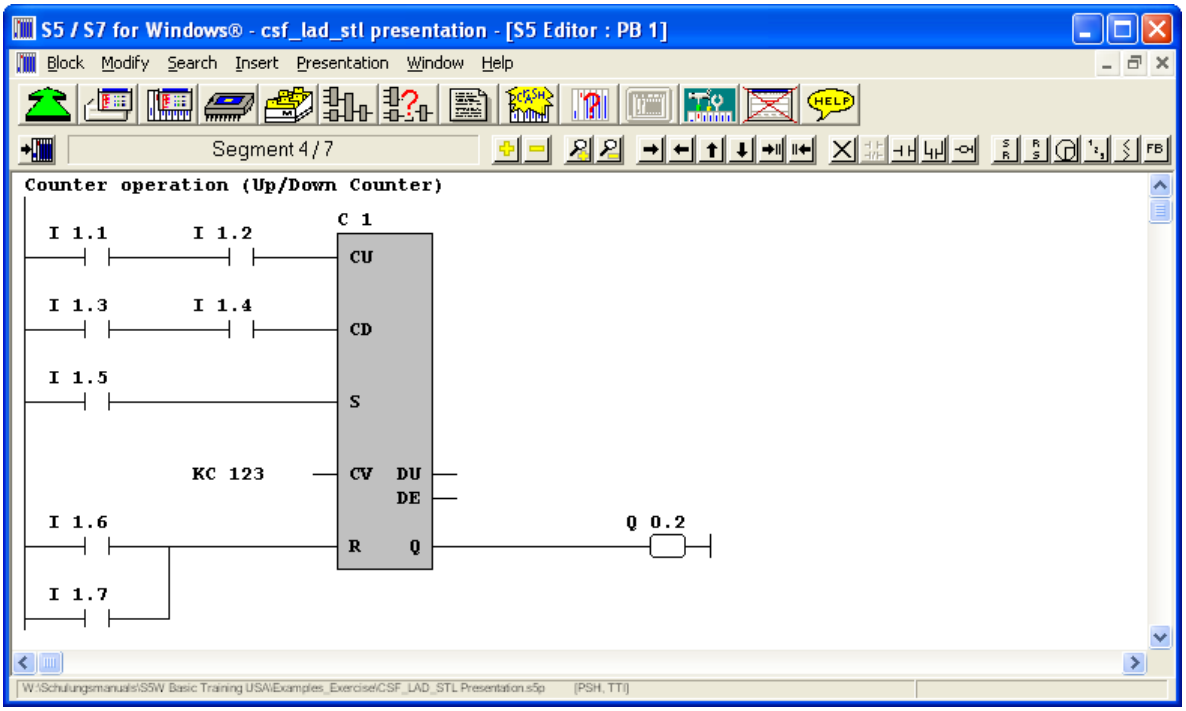
Timer operation:



Compare operation:

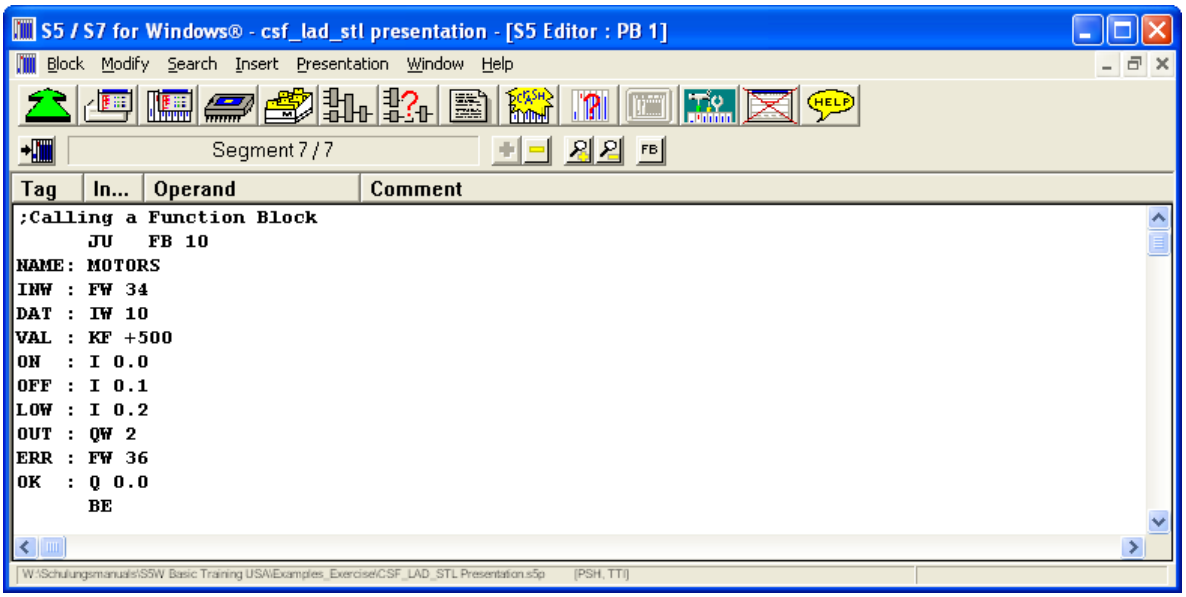


Counter operation:

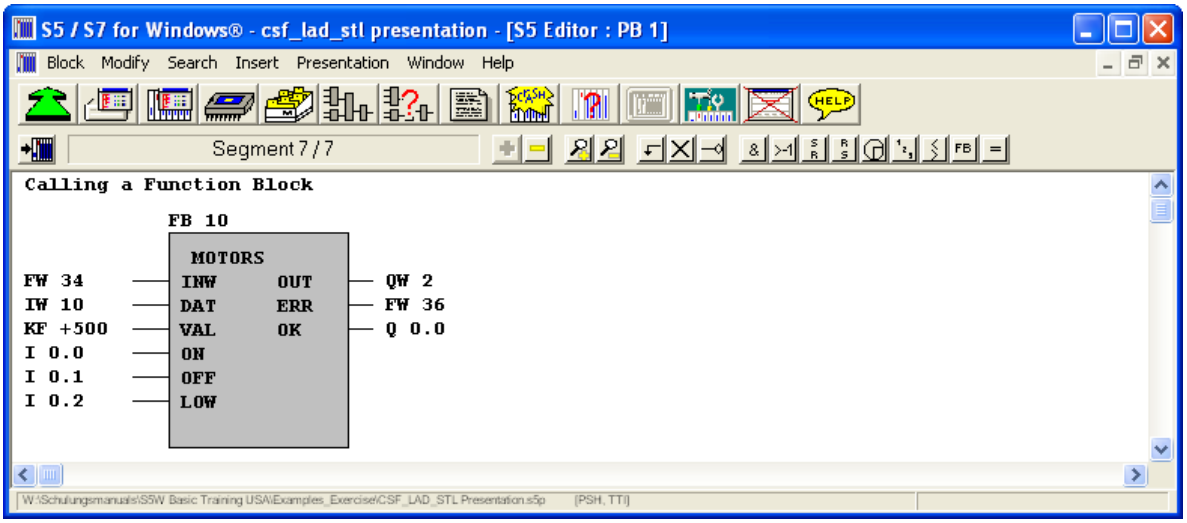


A function block call can also be called using LAD. *S5 for Windows*® automatically switches into STL mode. If you prefer a graphical presentation you must switch to CSF presentation. A function block call must be programmed in a separate segment.

The number of the function block is shown within the Block call instruction, the function block name and the names of the block parameters (the function block's "inputs" and "outputs") are listed below.



Function Block call, graphical presentation

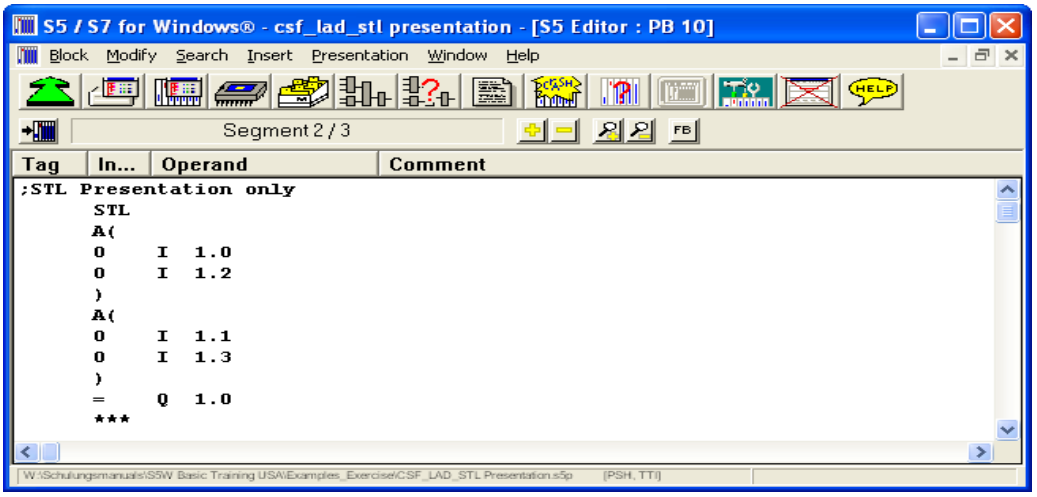


The number of the function block is specified above the box, the function block name and the names of the block parameters (the function block's "inputs" and "outputs") in the box.

A function block call must be programmed in a separate segment.

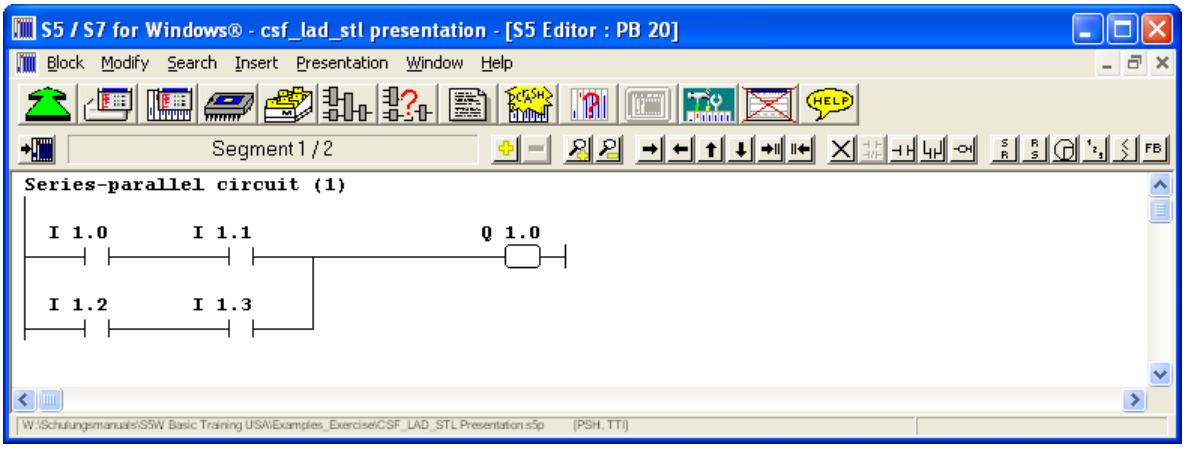
Even when ladder diagram has been selected as the representation method, it is still possible to enter basic STEP 5 operations which cannot be represented in graphic form. To do so, you can switch to STL presentation any time. This segment can subsequently be entered as statement list. The system reverts to ladder diagram mode at the beginning of the next segment.

It is also possible to prevent *S5 for Windows*® from switching into LAD mode (if LAD is selected in the "Preference" settings). By entering "STL" at the beginning of a segment the segment will only be displayed in STL.

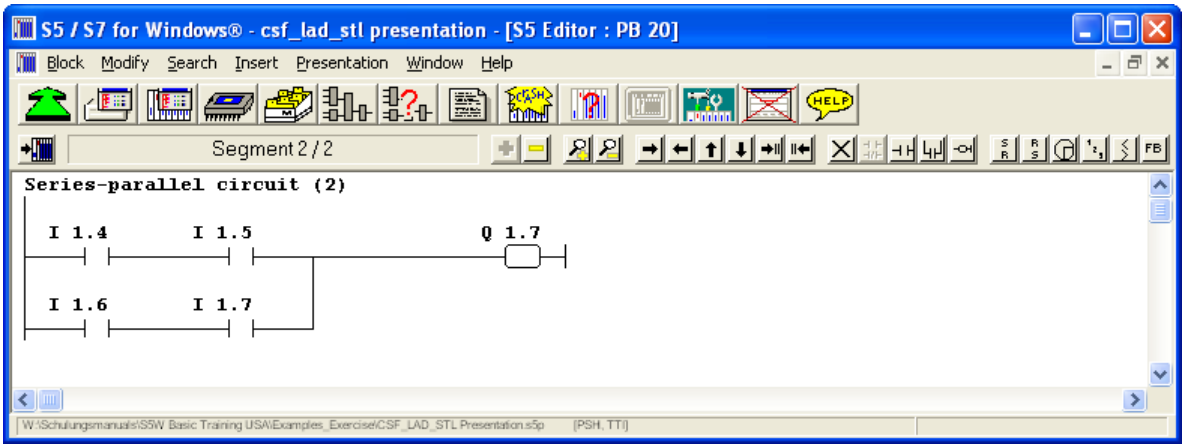


Example for program representation

SEGMENT 1 Series-parallel circuit (1)



SEGMENT 2 Series-parallel circuit (2)



The "rungs" are represented in segments. The segments are numbered automatically. The first line is reserved to hold the segment header. The segment header may be up to 60 characters (selected in the “Preference” settings, miscellaneous tab).

To enter a segment commentary of arbitrary length the “Comments display Window must be opened (Presentation Menu, Display Comments).

A new segment must be created for each “rung”. A segment may contain only one “rung”.

A block may comprise no more than 255 segments and no more than 4091 statements.

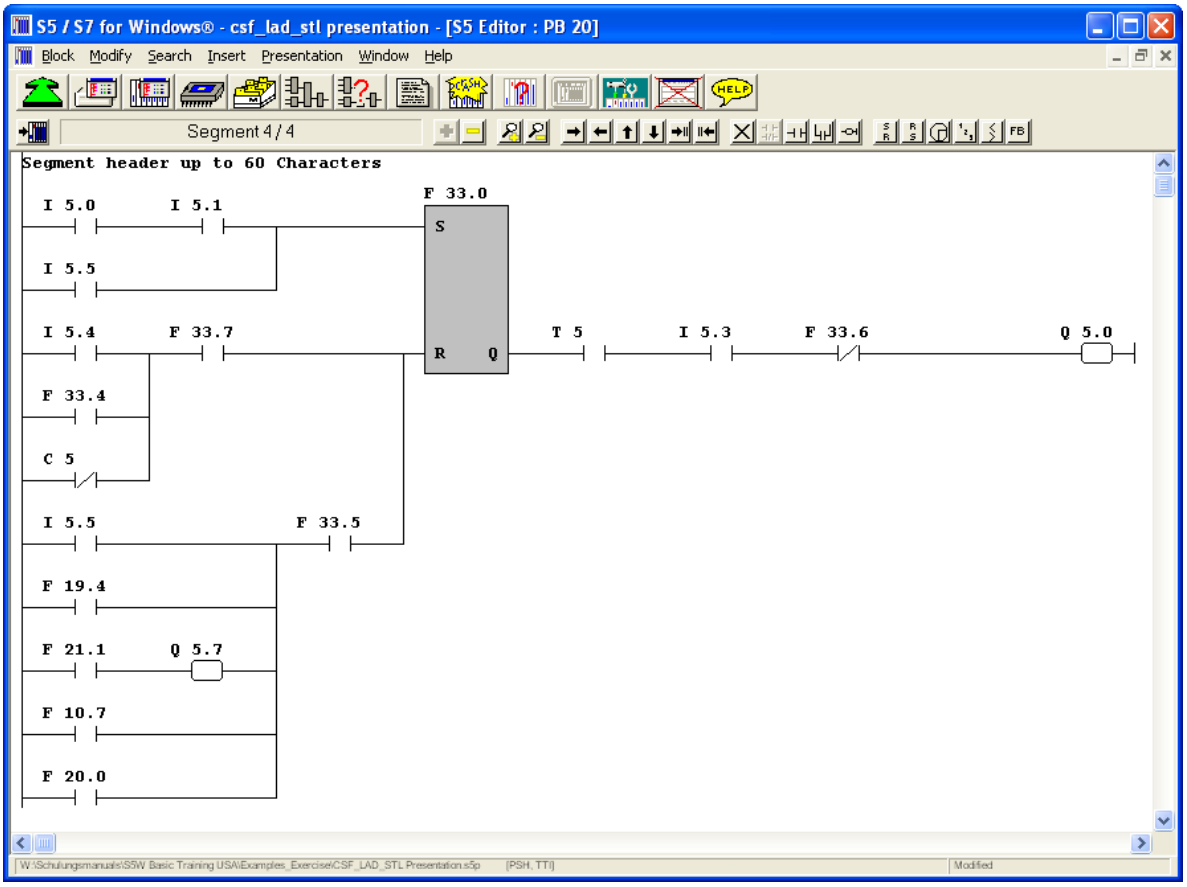
Ladder diagram representation on the monitor

The screen space for ladder diagrams is subdivided into up to eight (8) columns and up to thirty (30) lines. The logic operation is shown in the first seven columns, the outputs in the eighth column.

The contacts for a logic operation are drawn at the left screen margin (“rung”) or at a branch.

Each field contains a contact symbol and the associated operand identifier. The vertical links between the contacts (branches) are shown at the boundaries between the fields. Several fields may be required for “complex” function symbols.

Segment Representation on the monitor:

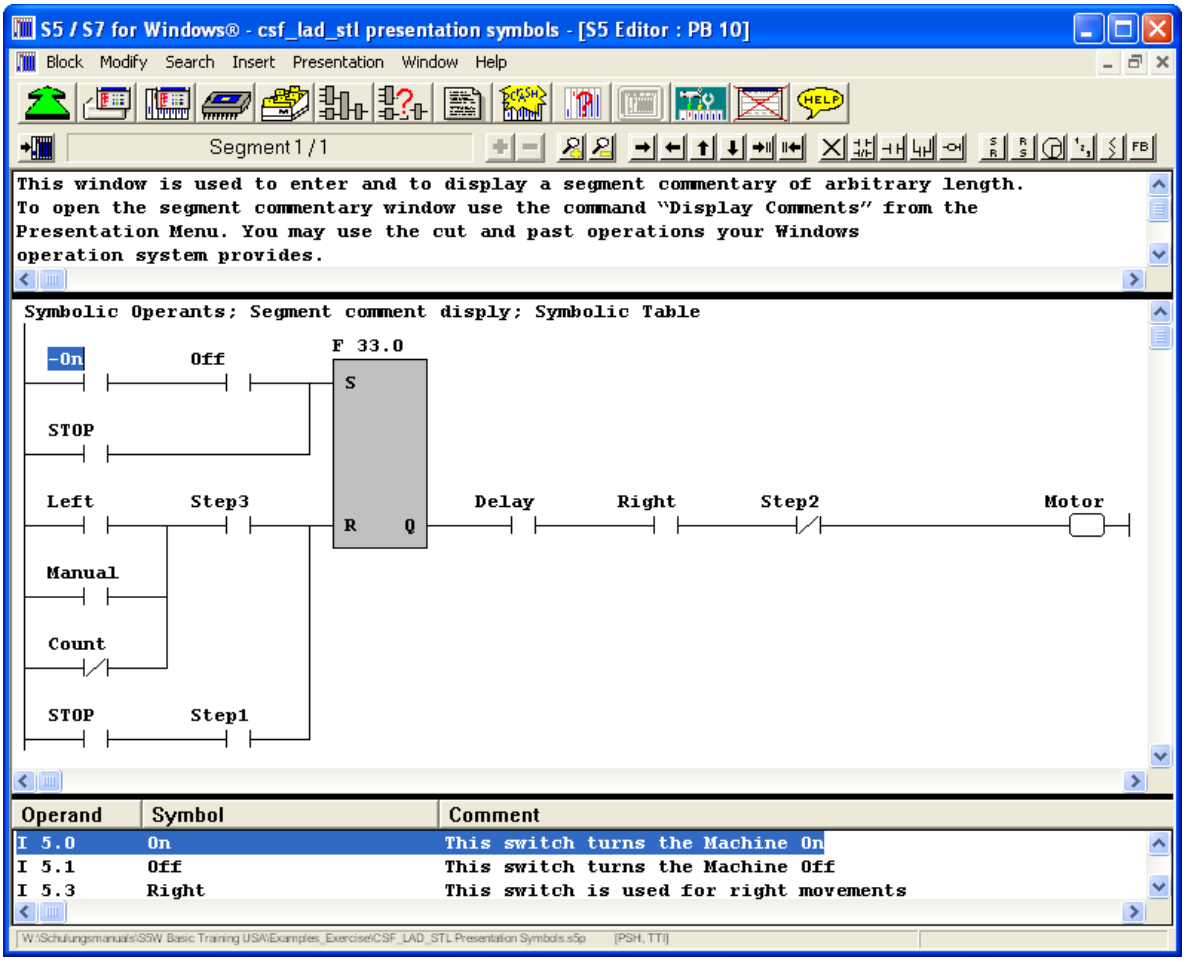


The screen can be rolled up. A rung (for an output, for instance) may comprise as many as 30 (ladder diagram) lines.

A “rung” corresponds to a “segment”.

Symbolic Programming

Segment commentary and symbolic programming in a ladder diagram



Symbolic operands (such as -On) may be used in place of absolute operand identifiers and parameters (such as I 5.0). An assignment list (Symbolic Table) must be generated, however, before symbolic operands can be used in the program.

To display symbolic operands the command “Symbolic Operands” from the Presentation Menu must be selected. To display the “Symbolic Table below the segment the command “Display Symbolic Table” from the Presentation Menu must be selected. This window displays the Operands in their absolute and symbolic form as well as an assigned comment.

Whenever an operand is marked in the segment the corresponding line is displayed in the Symbolic Table window and the line has also a blue background.

Statement List (STL)

In a statement list, the control task is described in form of a list using mnemonic (easily assimilated) abbreviations. The programming language is based on a standard for programmable controllers.

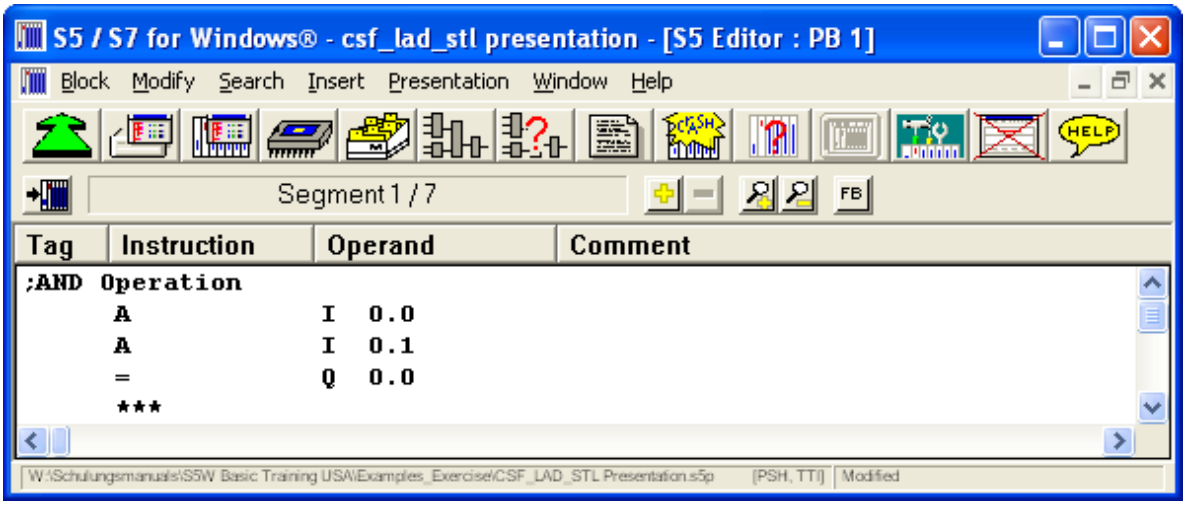
The program can be entered, modified and documented as statement list. This method of representation also enables dynamic status displays during on-line testing.

All operations available in the programming language (basic operations, supplementary operations and system operations) can be represented in statement list format.

Structure of a statement

A STEP 5 statement is the smallest independent unit of a program, and represents a processor directive. A statement comprises an operation code (such as A for the AND operation) and an operand (for instance I 1.7); an operand consists of an identifier (e.g. I for input) and a parameter (e.g. 1.7 for the 7th bit in the 1st byte).

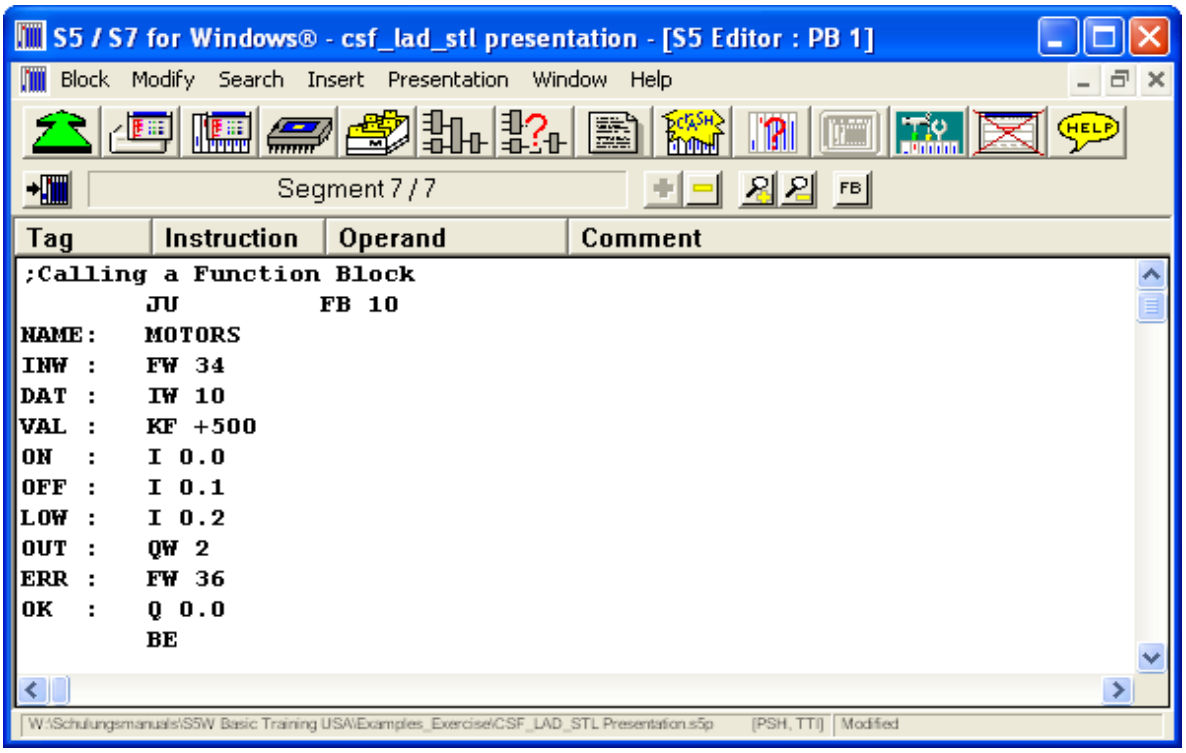
Example: Representing an AND operation



"Complex" operations for which there are special symbols in the graphic representation modes have no special symbols in a statement list; they are written in the same manner as any other statement.

Function Block Call (STL presentation)

A function block call is also represented in list form, and can be programmed together with other statements, whereby the function block's inputs and outputs (i.e. the block parameters) immediately follow the call statement.



All basic operations of the STEP 5 programming language, which can be represented in graphic form, can also be inputted as a statement list and outputted as control system flowchart or ladder diagram.

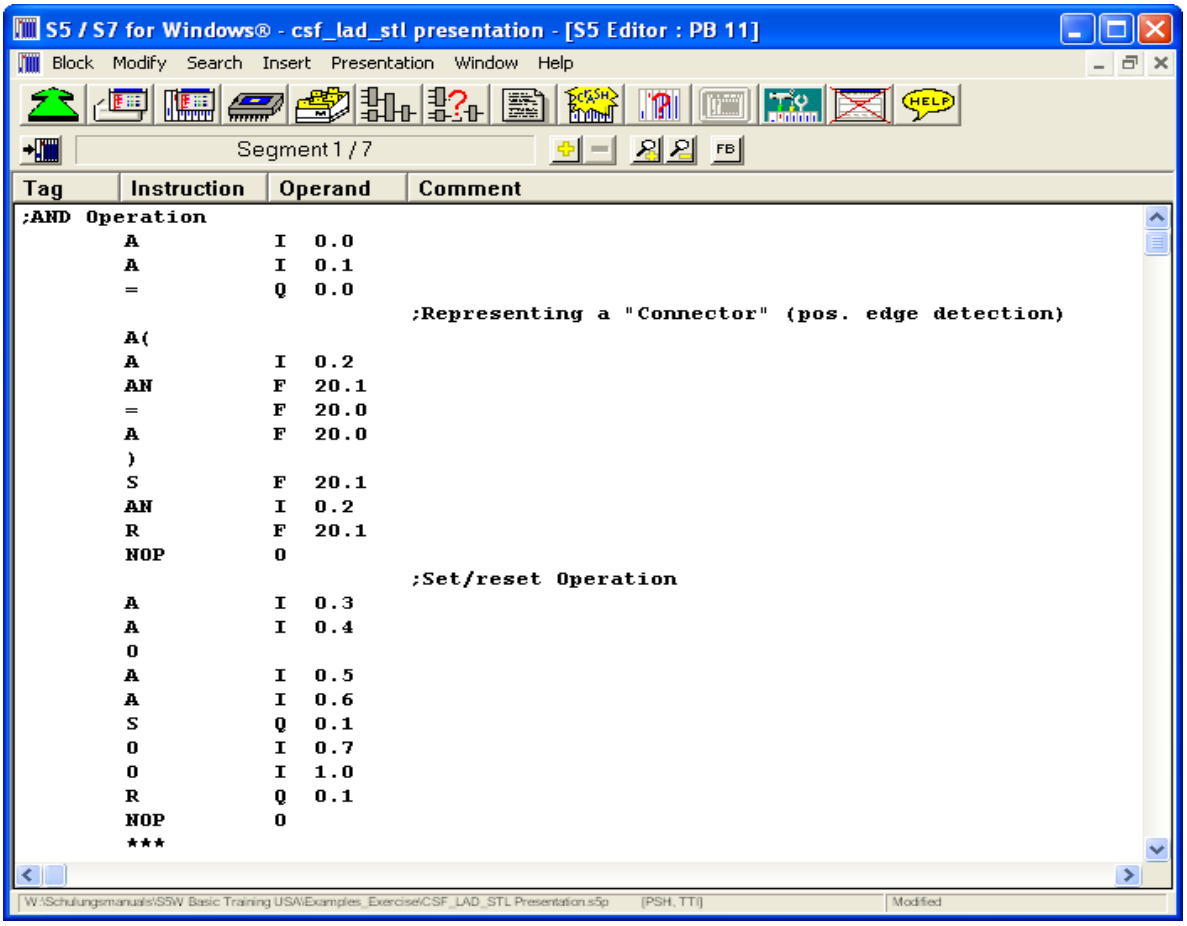
An input, however, certain conventions must be observed as regards auxiliary statements (such as "NOP 0"). Segments, which cannot be represented in graphic form, are always outputted as statement list.

In STL Presentation "rungs" do not have to be in separate segments. Several "rungs" can be combined into a segment. However such a combination of "rungs" in one segment cannot be displayed in a graphic mode (LAD, CSF presentation.

Note:

In STL Presentation there are no rules how to construct a segment. It is wise not to put to many lines of STL instructions into one segment.

Example for program representation



The logic operations are combined into segments. Several logic operations can be represented in each segment. The segments are numbered automatically.

A 60-character segment header may be specified in the fist line after the semicolon. Comments may be inserted in separate lines. Each comment line must start with a semicolon.

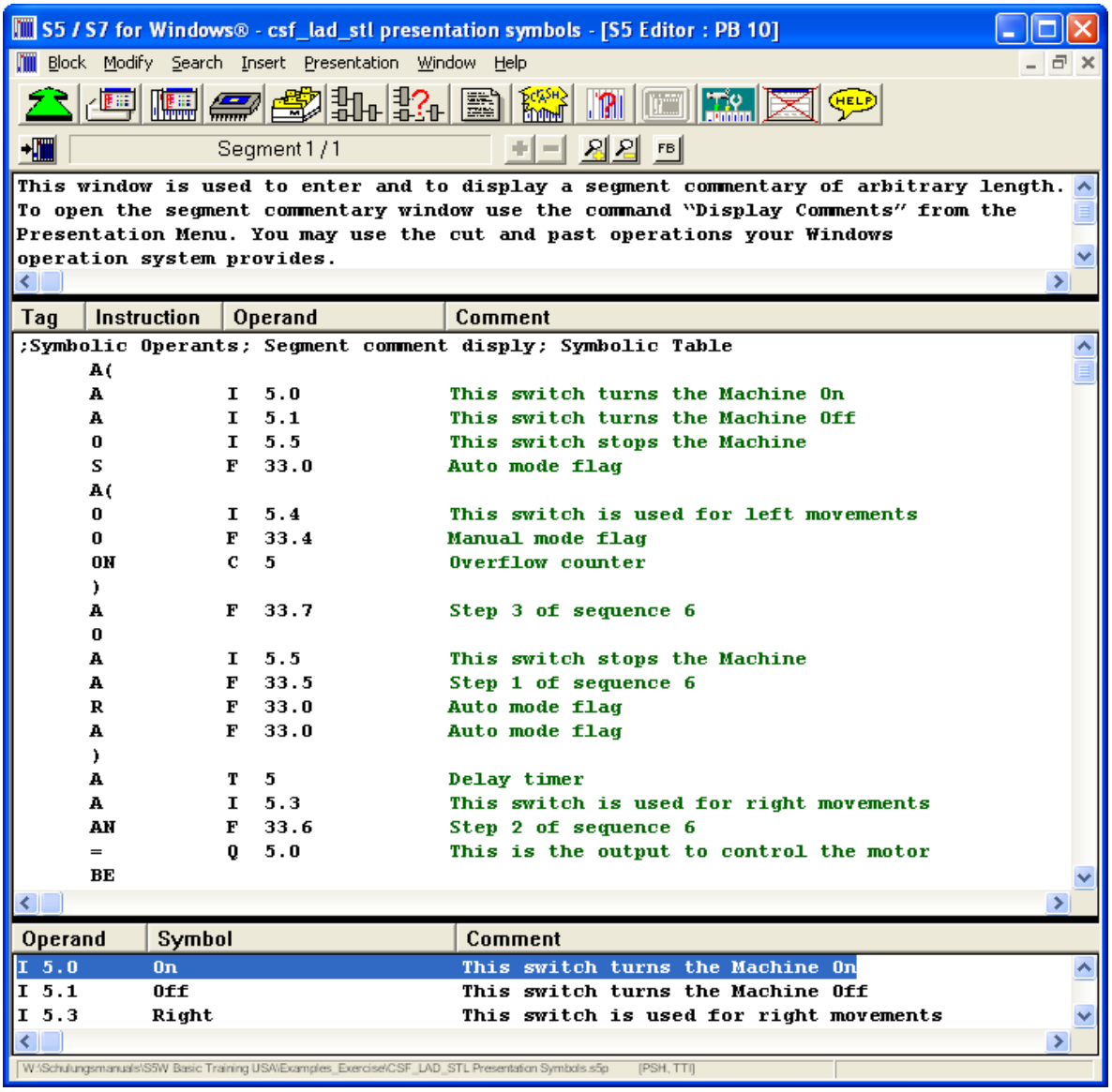
Statement Comments may also be inserted after the STL command, separated by a semicolon.

The segment end statement (“ ******* “) and blank lines are separate statements.

A segment commentary of arbitrary length may be written between the network header and the first statement to complete the documentation.

A block may comprise no more than 255 segments. As many as 256 MC5 statements may be programmed in each segment. A block is restricted to no more than 4091 statements.

Representation on the monitor



The STEP 5 statement begins with the operation code, followed by the operand identifier and the parameter (both of which are left-aligned). In function blocks, a symbolic entry point (jump label – Tag) may be written at the left of the colon.

To display the "Symbolic Table below the segment the command "Display Symbolic Table" from the Presentation Menu must be selected. This window displays the Operands in their absolute and symbolic form as well as an assigned comment.

Whenever an operand is marked in the segment the corresponding line is displayed in the Symbolic Table window and the line has also a blue background.

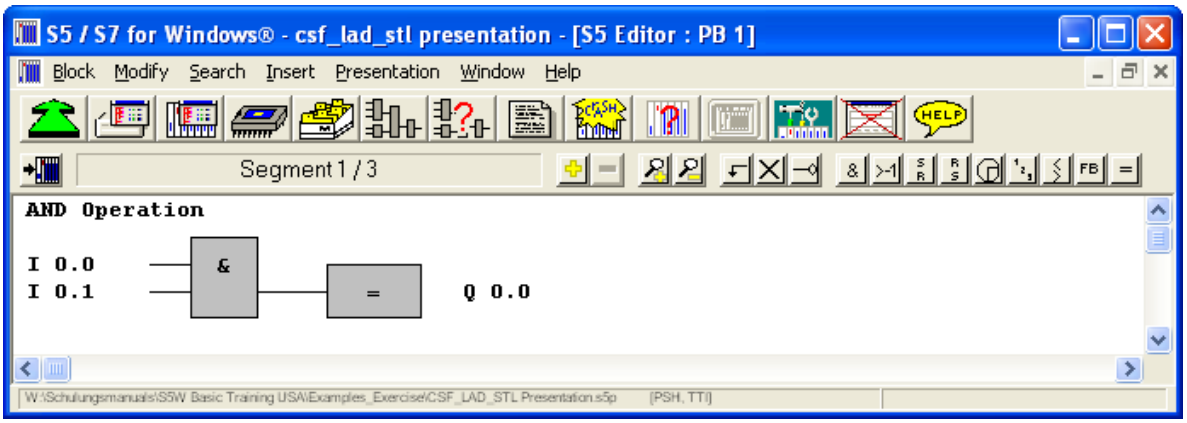
Control System Flowchart

A control system flowchart uses symbols to describe the control task. The program can be input, modified and documented as control system flowchart. This method of representation also enables the output of dynamic status displays during on-line testing.

Symbols used in control system flowcharts

The basic symbol used in control system flowcharts is the rectangular box. On the monitor screen unbroken lines form these boxes while standard characters are used to represent them in printouts. The symbol in a box identifies the operation, which the box represents. The inputs are shown at the left, the outputs at the right of the function symbol.

Example: Representing an AND operation



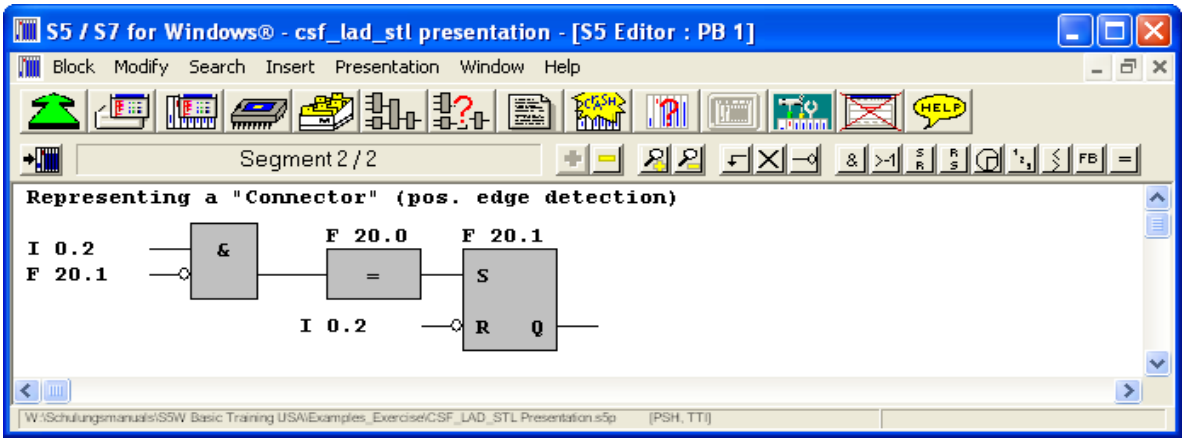
The binary logic operations represented in control system flowcharts are combinations of AND and OR operations.

A result assignment at the right of a logic operation always terminates that logic operation (see above), thus making it possible to represent individual set/reset operations and conditional block calls (except for those relating to function blocks).

In addition to logic operations AND and OR, there are also “complex” operations, i.e. set/reset, timer, counter and compare operations. These operations are also represented by boxes. A symbol in each box identifies the relevant operation. The inputs for these operations are shown at the left, the outputs at the right of the boxes.

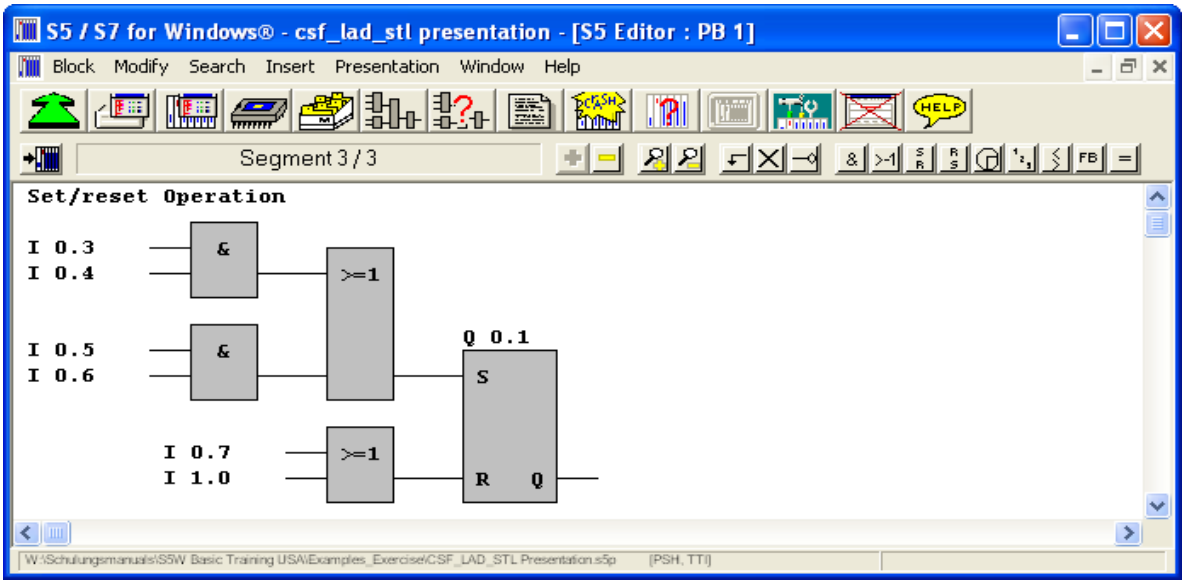
A special case is the “connector”, which represents a result assignment within a logic operation and is identified by a “=” symbol in the box.

Example: Representing a “connector” in a control system flowchart

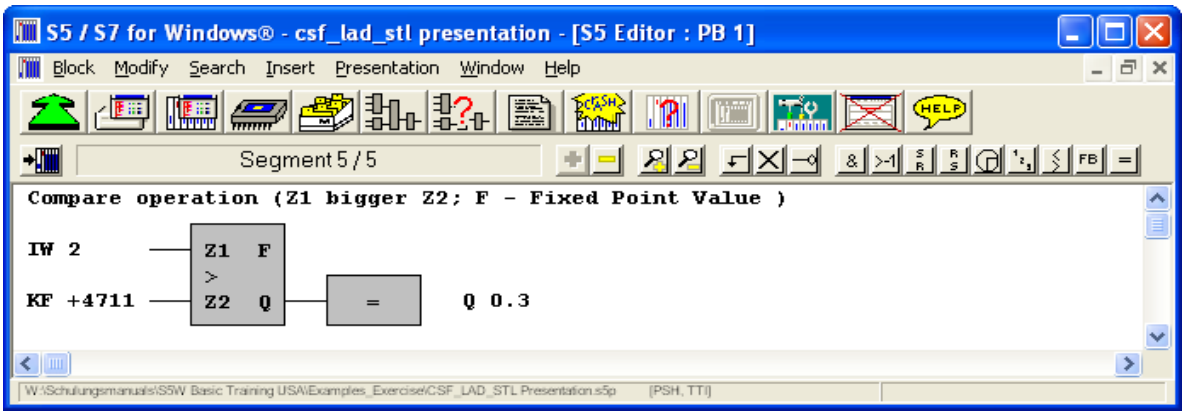


Examples for representing “complex” operations in a control system flowchart.

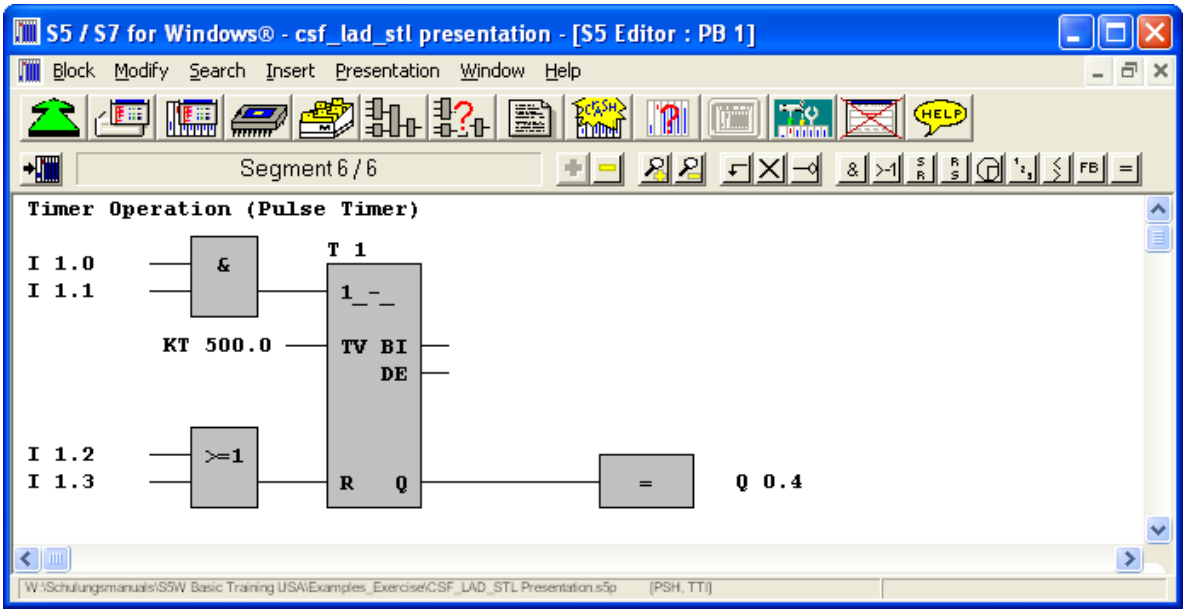
Set/reset operation



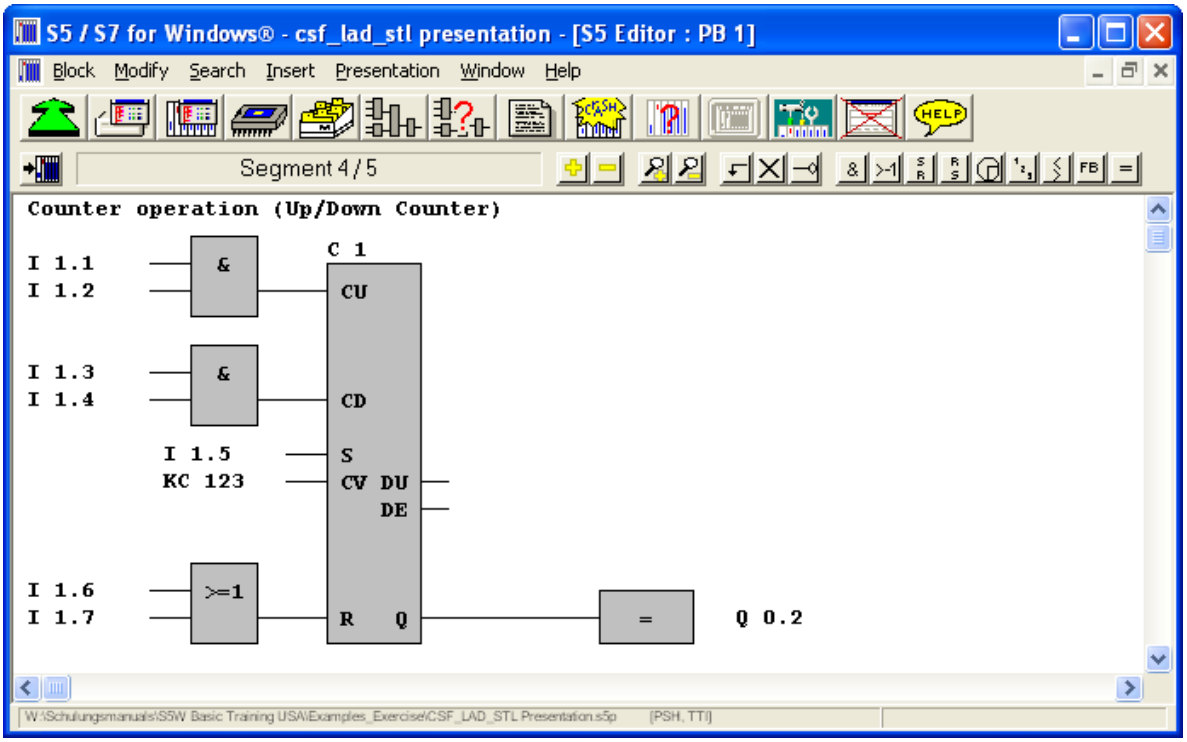
Compare operation



Timer operation



Counter operation



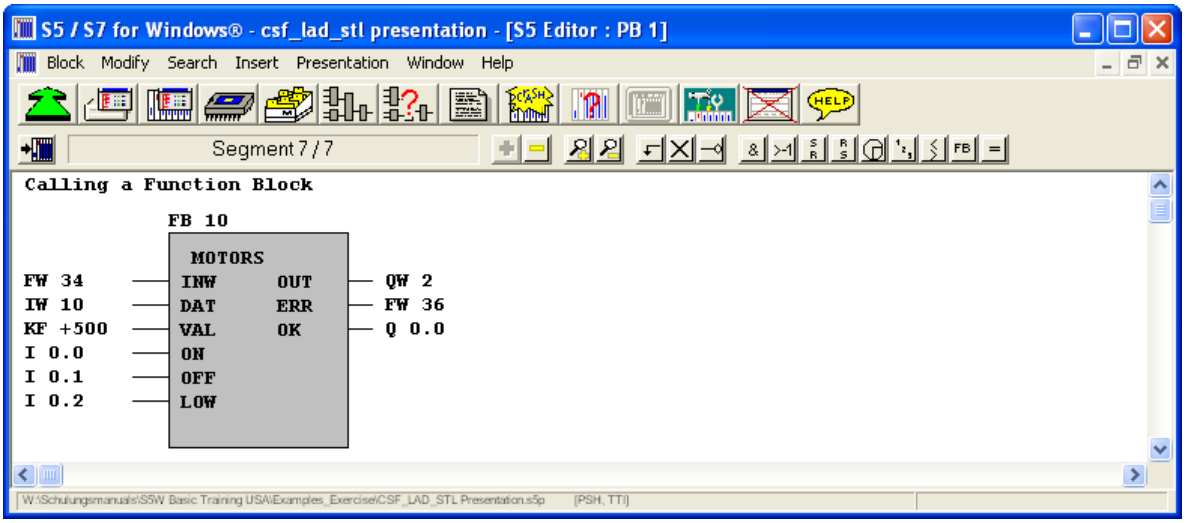
Calling a Function Block

A function block call can also be represented graphically.

To do so, the number of the function block is specified above the box and the function block name and the names of the block parameters (the function block's "inputs" and "outputs") in the box.

A function block call must be programmed in a separate segment.

Representing a function block call



Even when the control system flowchart has been selected as representation method, it is still possible to enter basic STEP 5 operations which cannot be represented in graphic form. To do so, you can switch to STL presentation any time. This segment can subsequently be entered as statement list. The system reverts to control system flowchart mode at the beginning of the next segment.

It is also possible to prevent *S5 for Windows*® from switching into CSF mode (if CSF is selected in the "Preference" settings). By entering "STL" at the beginning of a segment the segment will only be displayed in STL.

1.2 Structure of the Application Program

The program in the CPU is divided into the system program (Firmware) and the application, or user, program.

System programs (Firmware) are all statements and declarations relating to internal controller functions (e. g. the saving of data in the event of a power failure, organizational functions for the nesting of blocks, and so on). The system programs are stored in EPROMs in the CPU's program memory. The user has no access to system programs.

Application programs (User Program) comprise all statements and declarations for processing the signals affecting the controlled plant (process). Application programs are divided (structured) into blocks.

The **Blocks** in an application program are written in the STEP 5 programming language.

The organization blocks are the interface to the system program. A number of standard function blocks are integrated in the CPU's system program (integral special functions).

Blocks

A block is a portion of a program delimited by its function, structure or purpose. In STEP 5, a distinction is made between blocks containing statements for signal processing (organization blocks, program blocks, function blocks and sequence blocks) and blocks containing data (data blocks).

As a rule, a STEP 5 program consists of program sections which are invoked and processed sequentially. These sections are referred to as "blocks". There are several different types of blocks. These block types are application-dependent,

The program normally begins with organization block OB 1. The other blocks are then called as subroutines from within this block.

Organization blocks (OBs)

Organization blocks (OBs) control the application program either by listing the program blocks to be executed or by their existence as special functions in the CPU system program.

Organization blocks of the first type control cyclic, interrupt- driven and time-controlled program execution, restart performance, and responses in the event of errors and faults. These OBs are invoked in the system program, and are programmed by the user.

Organization blocks of the second type represent special functions, and the user may only invoke them.

Program blocks (PBs)

Program blocks (PBs) usually contain the largest part of the user program, and are programmed in accordance with process-related or function-related aspects. Program blocks can be entered and documented in all three methods of representation (CSF, LAD and STL).

Function blocks (FBs, FXs)

Function blocks (FBs, FXs) are used to implement frequently recurring or very complex functions. The user may make use of preprogrammed (standard) function blocks, and/or may also program his own FBs in STL notation.

In addition to the basic operations, additional operations (“supplementary operations” and “system operations”) may be used in function blocks.

Function blocks can be assigned parameters, i.e. the function implemented by a function block can execute with different operands (block parameters).

Sequence blocks (SBs)

Sequence blocks (SBs) are used to program sequencers. When the GRAPH 5 software is used, the entire sequencer is programmed in a single sequence block. If GRAPH 5 is not used, one sequence block must be written for each sequence step. These blocks are invoked by a “Sequence control” function block which assumes organization of the sequencer.

Data blocks (DBs, DXs)

Data blocks (DBs, DXs) contain the data for the application program.

The operand area for data (D) is used when the flag area does not offer sufficient capacity for storing signal states and data.

Generally speaking, the flag area is used primarily for storing binary signal states and the data area for storing digital values.

Data is organized in data blocks (DBs or DXs); 256 16-bit data words can be addressed directly in each data block.

The data is located either in user memory, where it must share the available space with the user program, or in a memory area reserved exclusively for data blocks (DR RAM).

Before a block can be processed, it must first be invoked, or called.

A call may be either unconditional or be dependent on the result of the previous logic operation (RLO).

Once the block has been processed and the Block End (BE) statement encountered, the program is resumed with the statement following the block call statement, i.e. in the “calling” block.

1.3 Segment

The program in a block is subdivided into segments.

In the graphic programming modes, a segment contains one logic operation (control system flowchart) or one rung (ladder diagram).

These strict conventions need not be applied to statement lists, in which a segment may contain as many as 255 16-bit statements (fewer when two-word statements are used).

It is nonetheless recommended that statement lists also have either a process-related or logic-related structure, thus enabling each segment to be documented as a self-contained program section.

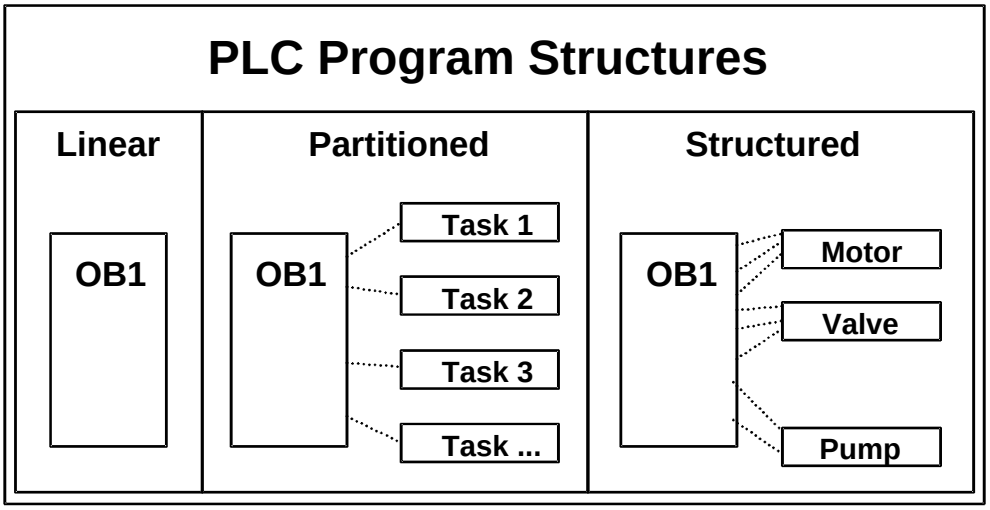
Each segment may be preceded by up to a 60-character segment header and a segment commentary of arbitrary length.

1.4 PLC Program Structures

The structure of PLC Programs can be segmented into three groups.

The kind of structure selected to build a PLC Program depends on the application.

The programming language STEP® 5 and/or *S5 for Windows*® provides the tools to build a PLC Program in all three structures.



1.5 Linear Programs

A program designed with a “linear” structure puts the entire program into one contiguous block of instructions, usually OB1. The program executes every instruction in sequence. This structure is a model of the hard-wired relay ladder logic that PLC systems initially emulated.

The linear program has a simple, straightforward structure.
Only one logic block (typically OB1) contains all of the instructions for the program.

Since all of the instructions reside within one block, this method of programming is best suited for projects that have one person writing the program. Since there is only one program file, software management functions (like archival of the program files) are simplified.

1.6 Partitioned Program

A partitioned program is divided into blocks, with each block containing the logic for a given set of devices or tasks. The instructions residing in an organization block (OB1) determines the execution of the partitioned blocks of the control program. For example, a partitioned program might contain the following elements:

- Functions for controlling each section of the equipment
- Functions for controlling each mode of operation for the equipment
- Functions for controlling the operator interfaces
- Functions for handling diagnostic logic

In the partitioned program, there is still no interchange of data or reusable code, however, each functional area is broken up into different blocks. This allows you to have several people programming at the same time without the conflict of editing the same file. The program that resides in OB1 contains all the instructions required to call the different blocks.

This design philosophy differs from a structured approach in that each block is completely self-contained: it gathers and manipulates its own data and processes its instructions in sequence. Unlike the structured program, the blocks of a partitioned program do not pass or receive parameters.

1.7 Structured programs

A “structured” program contains user-defined blocks of instructions with parameters, similar to user-defined instructions. In creating a program for a process or a machine, portions of the control logic are often repeated for common equipment or logic functions. Instead of repeating these instructions and then substituting a different addresses for specific equipment, you can write the instructions into a block and then have the program pass parameters (such as the specific address of the equipment and operational data) to the block. The following list shows examples of the use of generic (reusable) blocks in a program:

A block that contains the logic common to all of the AC motors in a conveyor system.

A block that contains the logic common to all of the solenoids in an assembly machine.

A block that contains the logic common to all of the operator station interfaces, in a canning line.

A block that contains the logic common to all of the drives in a paper machine.

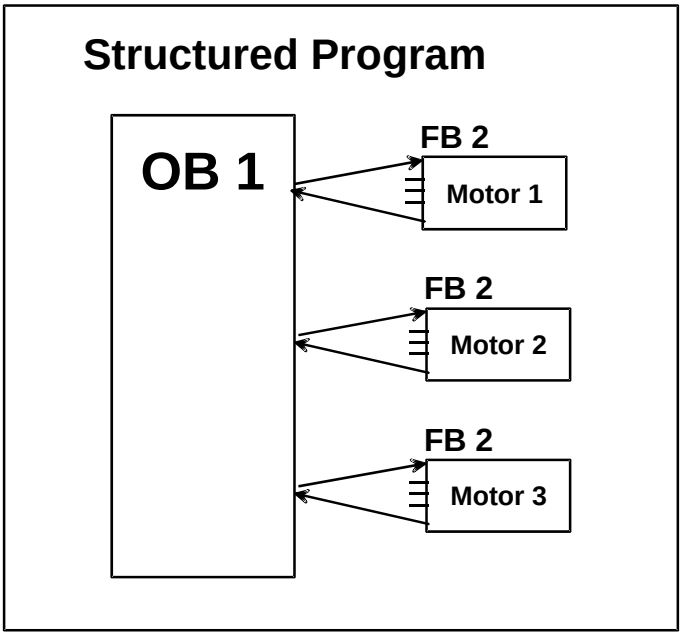
The structured program identifies the types of functions required by the process and attempts to provide a generic solution that can be used for several tasks.

For example, the pumps for both Ingredients A; & B; and the motor for the Agitator can be controlled by the same function block. By changing the parameters that are passed to the FB called “Motor” the program uses one block to control three different devices.

A structured program requires that you manage the data being stored and used by the program.

Example:

The Function (PLC Block)"Motor" in the illustration below contains the logical connection (S5-Code) of “Inputs” (Ix.x) and Outputs (Qx.x), which must be considered when switching on the different motors (e.g. mode of operation, temperature etc.).



The PLC Block calling the Block "Motor, supplies the information in order to switch on and control a specific Motor.

The programming language STEP® 5 offers different types of PLC Blocks to divide a PLC Program.

The instructions to define a logical problem are located in a PLC Block. Therefore a PLC Block can also be called “Sub-Routine”.

The Organization Block OB1 is used to supervise all the single PLC Blocks. The Organization Block OB1 is called by the operating system in a cyclic matter.

From the Organization Block OB1 the program branches out to all the other PLC Blocks (Sub-Routine) holding the actual control program.

It becomes thereby between Program Blocks (PB) and functional modules (FB) differentiated.

User programs for extensive tasks of automation are developed with partitioned and structured program sections.

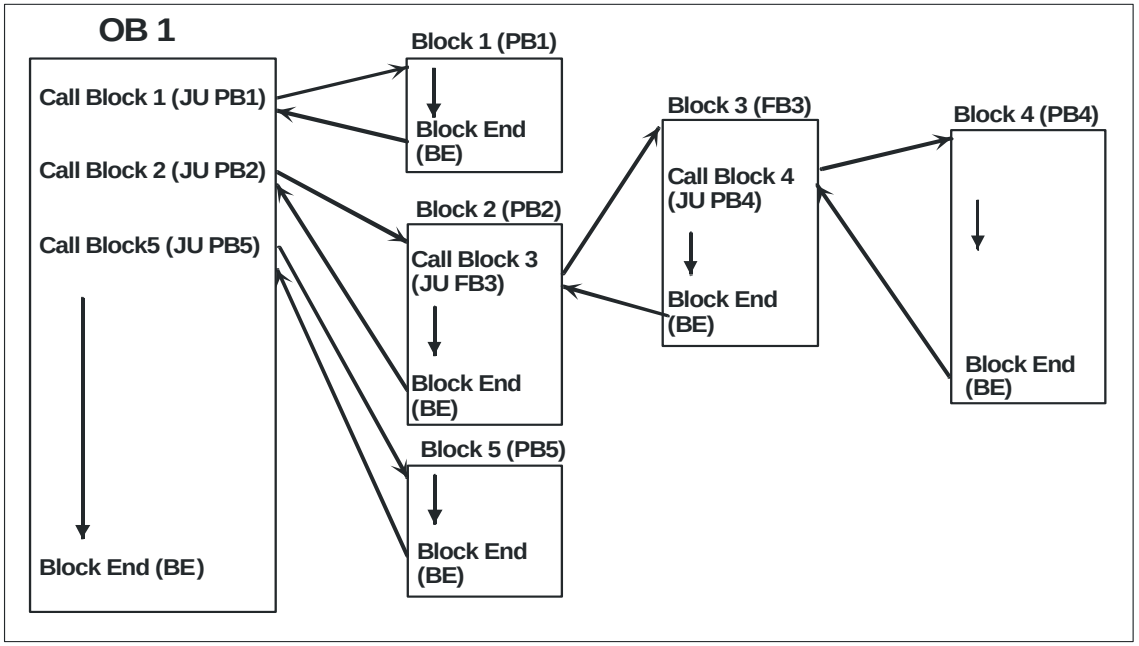
1.8 Example of a program structure

In the following example the absolute branching to a PLC Block (JU) is used not only in the Organization Block OB1, but also in the Program Block PB2 and the Function Block FB3.

With the instruction "JU PB2" within the Organization Block OB1 the program branches out to the Program Block PB2. The instruction "JU FB3" within the Program Block PB2 causes a further branching to the Function Block FB3, in order to branch out from there with the instruction "JU PB4" to the Program Block PB4.

Now the returns can take place. The arrows indicate that with “Block End” of the Program Block PB4 the program returns to the Function Block FB3. The “Block End” from the Function Block FB3 initiates a return to the Program Block PB2, and from there the program returns to the Organization Block OB1, initiated by “Block End”.

PLC Program Nesting



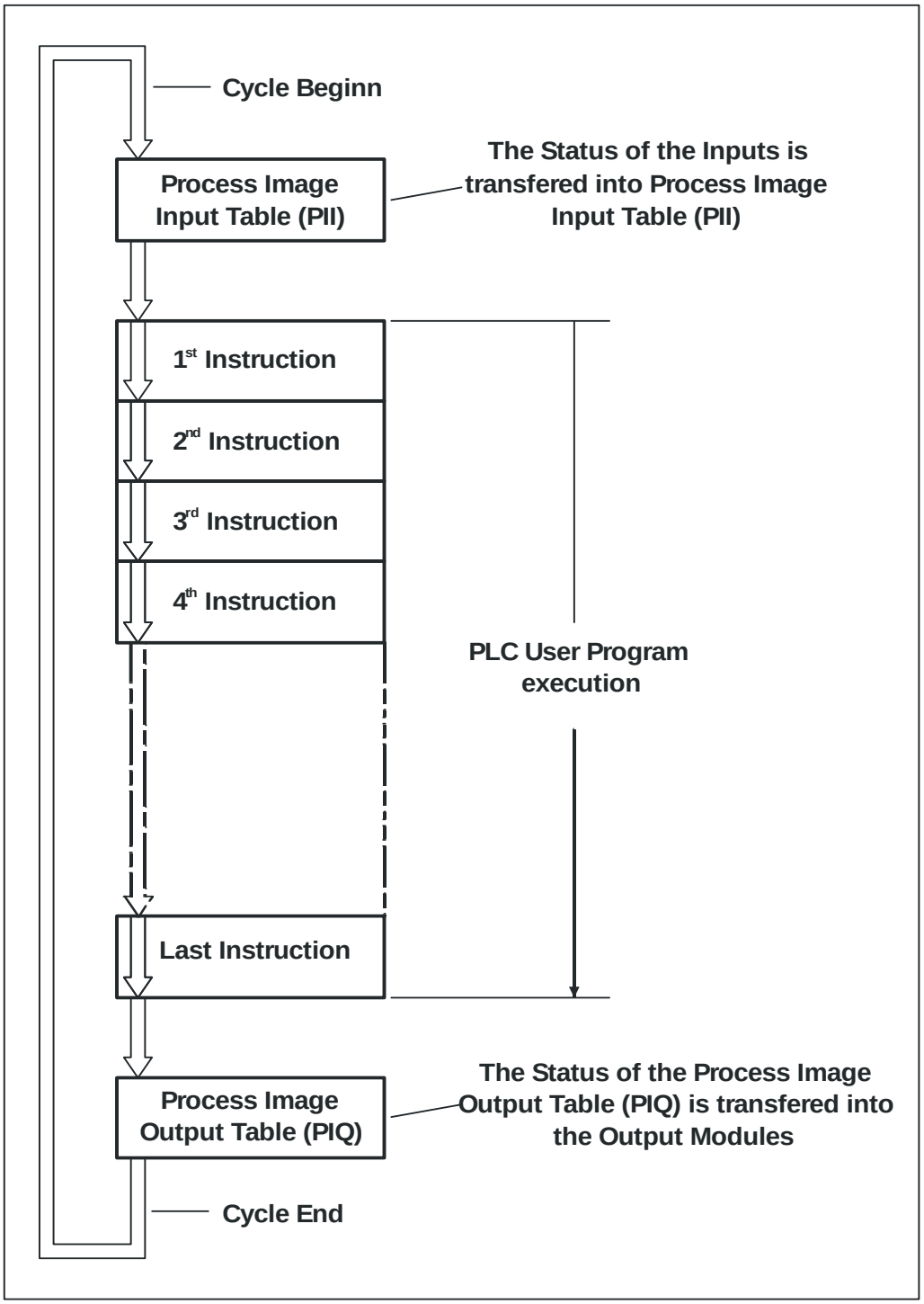
In a PLC Program "branching out" is called "nesting depth". The nesting depth states, how many PLC Blocks are called from OB1 in horizontal direction.

The maximum nesting depth allowed depends on the CPU type. If nesting goes beyond 32 levels, the PLC goes into the "STOP" mode

1.9 Cyclic Program Processing

In the following illustration, cyclic program processing is schematically represented. In the example a program is executed.

Cyclic PLC Program Execution



The Cyclic PLC Program Execution

The operating system starts the cycle time monitoring.

The data of the inputs of the Input modules (Peripheral Input Memory) are mapped into the Process Image Input Table. The Process Image Input Table is a storage area in the main memory (RAM) of the CPU.

The execution of the PLC user program is started. One instruction after another as written in the user PLC program is executed.

If all PLC Instructions are processed (Block End of OB1), the Process Image Output Table holds the results of the logical connections. The data of the Process Image Output Table are mapped to the “Peripheral Output Memory” and is now available at the outputs of the Output Modules.

The cyclic PLC Program execution described above shows some fundamental weaknesses in the function of a PLC.

The status of the inputs (actuators etc.) is read at the beginning of a cycle. Changes in status during the remaining cycle time are normally not recognized by the system.

The outputs are only updated after the complete PLC Program has been executed.

To overcome these restrictions, additional functions are available to bypass the cyclic program execution.

1.10 CPU Start-up

With the change of the mode selector from "STOP" (ST) to "RUN" (RN) the CPU executes the "RESTART" mode automatically and switches from "STOP" to "RUN".

RESTART OB's

Manual cold restart: OB21 is processed

Automatic cold restart: OB22 is processed (mode selector at "RN")

Restart Characteristics

Everything that takes place between

- a STOP RUN transition (manual cold restart) or
- a POWER UP RUN transition (automatic cold restart after power up)

is referred to as restart characteristics.

Two phases can be distinguished during restart:

- The cold restart routine (PLC cannot be directly influenced)
- The actual RESTART (PLC characteristics can be controlled in RESTART OBs (OB21 and OB22)).

Cold Restart Routine

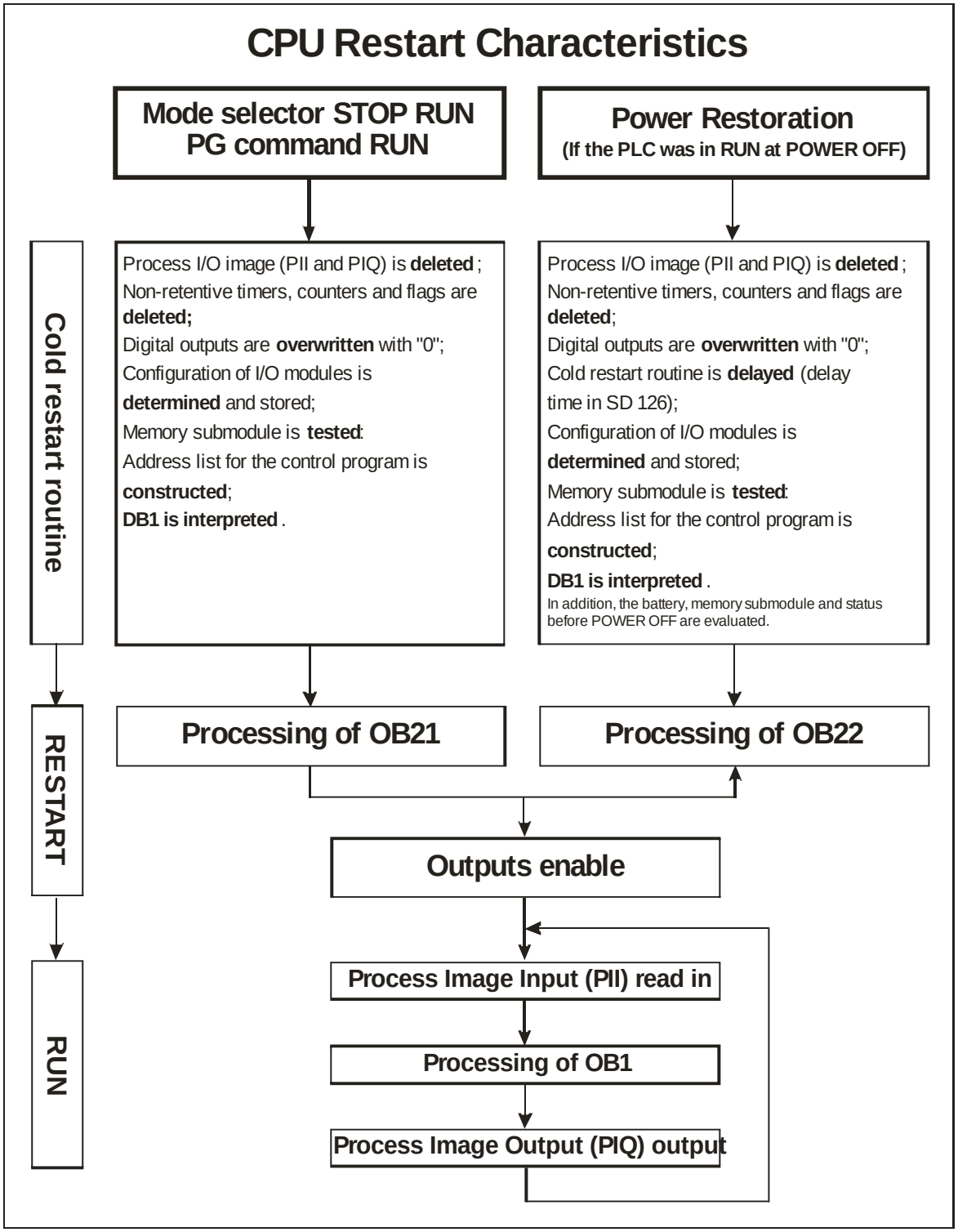
The following applies while the CPU runs the cold restart routine:

- The status of the error LEDs remains unchanged during manual cold restart.
- The error LEDs light up momentarily during automatic cold restart after power up
- Outputs display signal "0" if all output modules are disabled
- All inputs and outputs in the process I/O image display signal "0"
- Scan time monitoring is inactive.

During the cold restart routine, the processor configures the I/O modules and stores this information.

Restart Characteristics and Cyclic Operation

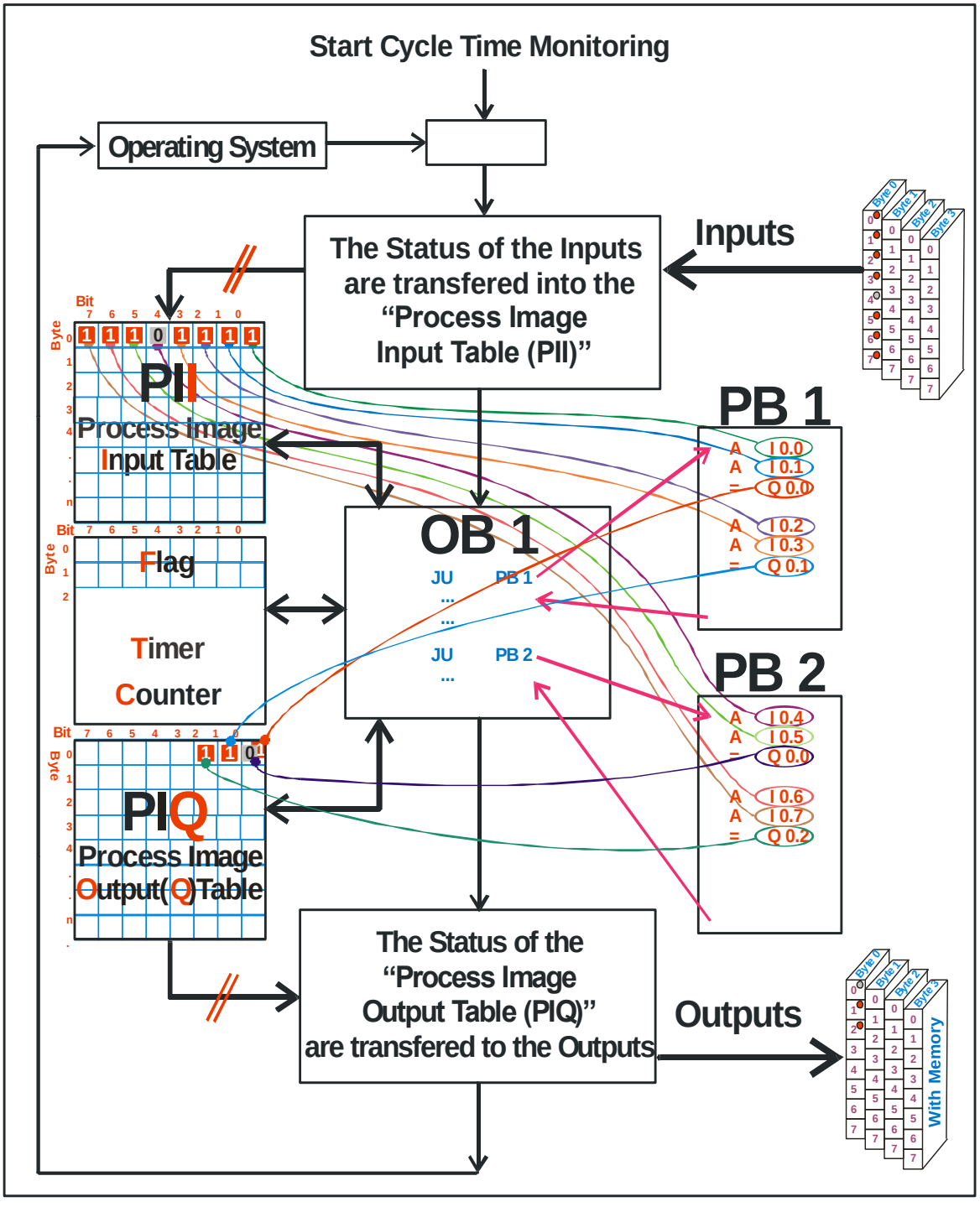
The following figures give an overview of the restart characteristics of the CPUs and of cyclic operation. They also show how the restart characteristics depend on the state of the backup battery and they indicate the conditions for changing the operating mode.



1.11 Cyclic Program Processing

In the following illustration, cyclic program processing is schematically represented. In the example a program is executed, which consists of the PLC Blocks OB1, PB1 and PB2.

Cyclic PLC Program Execution



The Cyclic PLC Program Execution

The operating system starts the cycle time monitoring.

The data of the inputs of the Input modules (Peripheral Input Memory) are mapped into the Process Image Input Table (PII). The Process Image Input Table is a storage area in the main memory (RAM) of the CPU.

The execution of the PLC user program is started. From OB1 the program execution branches out to PB1. The logic functions from PB1 are executed. Initiated by “Block End” from PB1 the program returns to the Organization Block OB1.

If all PLC Blocks are processed (Block End of OB1), the Process Image Output Table (PIQ) holds the results of the logical connections. The data of the Process Image Output Table (PIQ) are mapped to the “Peripheral Output Memory” and is now available at the outputs of the Output Modules.

The operating system tests the cycle time monitoring and restarts the execution.

If the preset watchdog is timed out, the CPU is switched into the stop condition and the cyclic PLC Program execution is terminated.

The cyclic PLC Program execution described above shows some fundamental weaknesses in the function of a PLC.

The status of the inputs (actuators etc.) is read at the beginning of a cycle. Changes in status during the remaining cycle time are normally not recognized by the system.

The outputs are only updated after the complete PLC Program has been executed.

To overcome these restrictions, additional functions are available to bypass the cyclic program execution.

1.12 Organization Blocks for Interrupt-Driven Program Execution

The events that lead to an OB being called are known as interrupts. Not all S5CPUs have the complete range of organization blocks.

Non-Cyclic Program Execution

With STEP 5, selected parts of the user program that do not need to be executed cyclically can be executed when the situation deems it necessary. The user program can be divided up into "subroutines" and distributed in different organization blocks. If the user program should react to an important signal that seldom occurs (for example a limit switch indicates that the slide is at the end), the section of program to be executed when this signal is present can be written in an OB that is not executed cyclically.

Apart from cyclic program execution, STEP 5 provides the following types of program execution:

- Time-driven program execution
- Hardware interrupt-driven program execution (from Inputs)
- Diagnostic interrupt-driven program execution
- Multi-computing interrupt-driven program execution
- Error handling

By providing interrupt OBs, the S5 CPUs allow the following:

- Program sections can be executed at certain times or intervals (time-driven).
- The User PLC Program can react to external signals from the process.

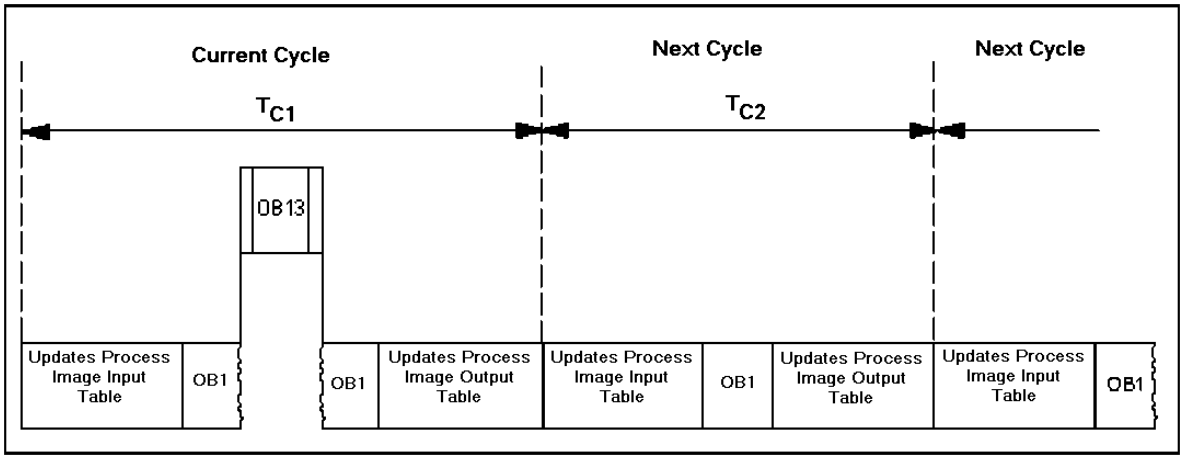
The cyclic user program does not need to query whether or not interrupt events have occurred. If an interrupt does occur, the operating system makes sure that the user program in the interrupt OB is executed, so that there is a programmed reaction to the interrupt by the PLC.

Cyclic Interrupt Organization Blocks (OB10 to OB18)

The S5 CPU's have integrated “Cyclic Interrupt OBs”. These OBs can be called in a certain time pattern by the operating system, completely independent from the cyclic program (OB1)

The internal CPU clock calls for the interrupted OB to interrupt the cyclic program sequence.

After processing the OB the program returns to its cyclic execution.



Note:

The cycle time of a program (OB1) can be substantially changed by processing “Interrupt OBs”.

This difference in execution time, per OB1 cycle, can cause problems.

Overview of the Cyclic Interrupt OBs default settings

(not all CPUs have all Cyclic Interrupt OBs)

Organization Block	Time Pattern
OB 10	10 ms
OB 11	20 ms
OB 12	50 ms
OB 13	100 ms
OB 14	200 ms

Organization Block	Time Pattern
OB 15	500 ms
OB 16	1 s
OB 17	2 s
OB 18	5 s

Interrupt Driven Program Scanning

The S5 CPUs provide interrupt OBs that react to signals from the input modules.

Interrupts are triggered when a signal module, with system interrupt capability passes on a received process signal to the CPU.

System interrupts (IR-A, IR-B, IR-C, IR-D) can only be executed when the corresponding organization block exists in the CPU program. If this is not the case, an error handling is executed (OB70 to OB87 / OB121 to OB122)).

If the system interrupt OBs are deselected in the parameter assignment, these cannot be started. The CPU recognizes a programming error and changes to STOP mode.

Overview of the System Interrupt OBs default settings

(not all CPUs have all Hardware Interrupt OBs)

Organization Block	Parameter
OB 2	System Interrupt A
OB 3	System Interrupt B
OB 4	System Interrupt C
OB 5	System Interrupt D

Error Handling Organization Blocks

The following table shows the types of errors that can occur, divided in to the categories of the error OBs.

	Error Type
	OBs for handling programming errors and PLC faults
OB19	When a block is called which has not been loaded
OB23	CPU Redundancy Error (only in H CPUs, e.g., failure of a CPU)
OB24	Timeout during update of the process image and the inter-processor communication flags
OB27	Substitution error
OB32	Transfer error
OB34	Battery failure

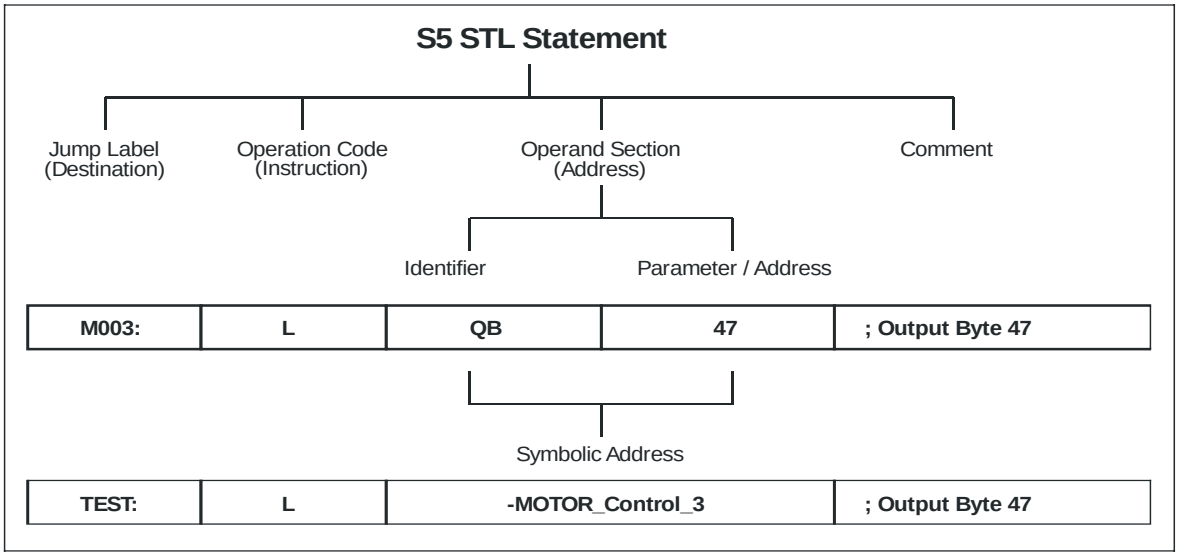
	Error Type
	OBs for handling system errors
OB26	Scan time exceeded
OB33	Collision of two timed interrupts
OB35	I/O error
	OBs which offer operating functions
OB31	Scan time triggering
OB160	Programmable time loop
OB250	Operating system services
OB254	Read in process I/O image
OB255	Output process I/O image

2 Statement List Instructions Structure

A large number of STEP® 5 instructions can only be displayed in the “Statement List” (STL) presentation.

An instruction statement is the smallest executable part of a PLC program and is made up of individual components. The statement is interpreted, according to its structure, and is executed by the CPU. Depending on the type of statements, the structure may vary.

Basically there are two types of statements. One is a statement made up of an instruction alone (e.g. NOP, NOT, etc.) and the other is a statement made up of an instruction and an address / parameter field (e.g. L +12, L -Stop, etc.).



Note:

With the **“Format”** (key **F9)** command all “Key Words/Characters” entered in the “Instruction and Address Field” that are lower case characters are converted into upper case characters. In addition the “Format” command puts every field in its predefined column.

Jump Labels, Symbolic Addresses and Comments are not changed with the “Format” command.

S5 for Windows® supports all instructions used with the programming language STEP®5.

Jump Label

The destination of a jump instruction is indicated with a label. The label may have up to four (4) characters. The first character must be an alpha character. The destination label itself is terminated with a colon (:) (e.g. TEST:).

Instruction (Operation Code)

In the instruction field of a statement, the task that the CPU should execute is defined (e.g. A for AND, O for OR, T for a transfer, etc.).

The S5 for Windows® Format (F9) command converts all typed characters into capital letters and puts them into the instruction field column.

S5 for Windows® supports all the instructions available in the Siemens® S5 PLC series. A list of the instructions that your particular CPU can support will be found in the instruction list manual for that CPU.

Address (Operand Section)

In the address field of a statement, who should participate is defined, when the instruction is executed by the CPU. This could be an absolute addressed variable (e.g. QB47), a defined symbolic variable (e.g. Limit_Switch), or a constant (e.g. KT 500.1), etc. Some instructions do not require an operand.

Absolute Address

Identifier:

S5 for Windows® supports all “Identifiers” used with absolute addresses for the instructions available in the Siemens® S5 PLC series.

The S5 for Windows® command Format (F9) converts all identifier characters into capital letters and puts inserts them into the appropriate column.

Parameters:

A parameter is an address made up of numbers. The S5 for Windows® Format (F9) command does not change the address but inserts it into the appropriate column.

Symbolic Addresses

The S5 programming syntax requires that the Symbolic Addresses must be typed in the statement in the same form as they are declared, with regard to lower and upper case letters. The S5 for Windows® Format (F9) command does not change the variable but inserts them into the appropriate column.

Symbolic Addresses are defined in the symbolic table. A symbolic variable is assigned to an absolute variable. This declaration must be done in the STL editor prior to using the symbolic variable. A Symbolic Address must be clearly defined for all blocks and may be used throughout the entire PLC program.

Comments

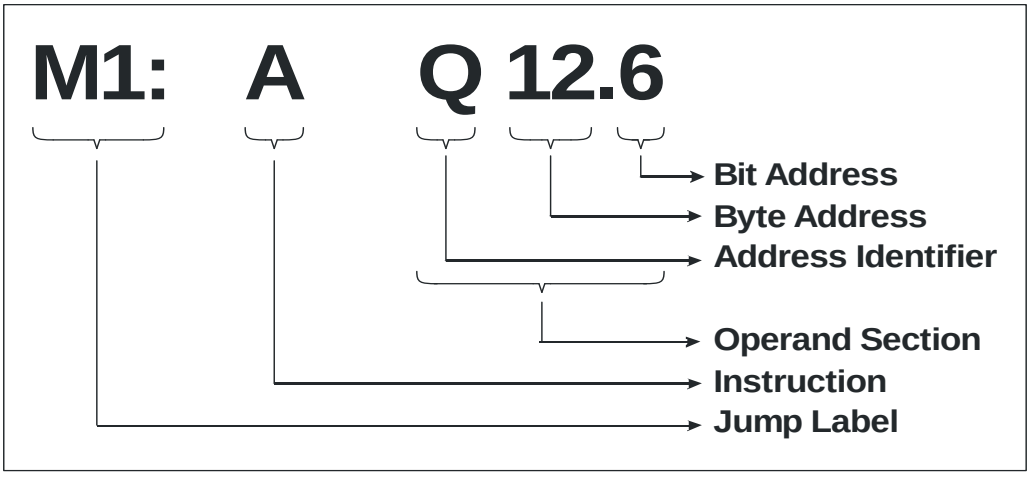
Each statement line may have a comment assigned to it. The optional comment starts with the semicolon character (;) and is valid up to the end of the line. The comment may have up to 60 printable characters.

A comment may also be entered into a separate line. This line must starts with the semicolon character (;).

The S5 for Windows® Format (F9) command does not change the comment but inserts it into the appropriate column.

```
L       FW 106              ; This is a STL Line comment
                            ; This comment is entered in a separate line
```

STL Instruction



2.1 STEP 5 Operands

The programmable controller processes the signal states of sensors made available via "inputs". The results of the logic operations are forwarded to the actuators via "outputs".

These variables (inputs and outputs) are referred to as operands, and are processed using functions or operations. The STEP 5 programming languages recognizes several different types of operands, the most important of which are inputs (I), outputs (Q), I/Os (P), flags (F), timers (T), counters (C) and data (D). The letters in parentheses are the abbreviations used for the various operand types in STEP 5.

The majority of operands can be processed bit by bit. A group of eight contiguous bits combined to form a single unit is referred to as a byte. A word consists of 16 bits, a double word of 32 bits.

Inputs, Outputs, I/Os

The I/O area (P – Peripherals) is used for direct addressing of the I/O modules in the user program. This area enables addressing of 256 bytes on input modules and 256 bytes on output modules.

As a rule, the operand areas for inputs (I) and outputs (Q), rather than the P (Input / Output modules) area, are used in the program to address the I/O modules (which cannot be referenced by bit).

These operand areas are located on the CPU in an area of memory referred to as the "process image".

The central processor's system program loads the signal states of the input modules into the process input image at the start of the program scan. The signal states of this operand area are then interrogated and logically gated as per the operations and functions written in the user program, and the appropriate bits subsequently set in the process output image. At the end of the program scan, the system program automatically transfers the signal states of the process image to the output modules.

Should the 256-byte I/O area prove insufficient, expansion units can be interfaced to enable use of the extended I/O area (called the O area) or to provide an extended addressing capacity.

Flags, Timers, Counters

The flags (F) are the controller's "contactor relays", so to speak, and are used primarily for storing binary signal states. The operand area for flags is a special memory area on the CPU.

There are 256 flag bytes (the equivalent of 2048 flag bits).

Some CPU's have an extended flag area (S) with 1024 additional flag bytes (equivalent to 8192 flag bits). These "S flags" are handled in the same way as the "F flags".

The operand area for timers (T) corresponds to the timing relays in a contactor control system. The timers are located in a special operand area on the CPU. Up to 256 timers are possible (depending on the CPU).

Five different kinds of timers can be implemented; these can be programmed for times in the range from 10ms to 9990s (2h 46min 30s).

The counters (C) function as hardware counters, but are located in a special memory area on the CPU. Up to 256 counters are possible (depending on the CPU).

All of these counters can be used as up or down counters, and all have a counting range of from 0 to 999.

Counts in the negative range are not possible. The count is made available in binary or BCD.

Because these counters are software counters, their operating frequency depends on the program scan time.

Data

The operand area for data (D) is used when the flag area does not offer sufficient capacity for storing signal states and data.

Generally speaking, the flag area is used primarily for storing binary signal states and the data area for storing digital values.

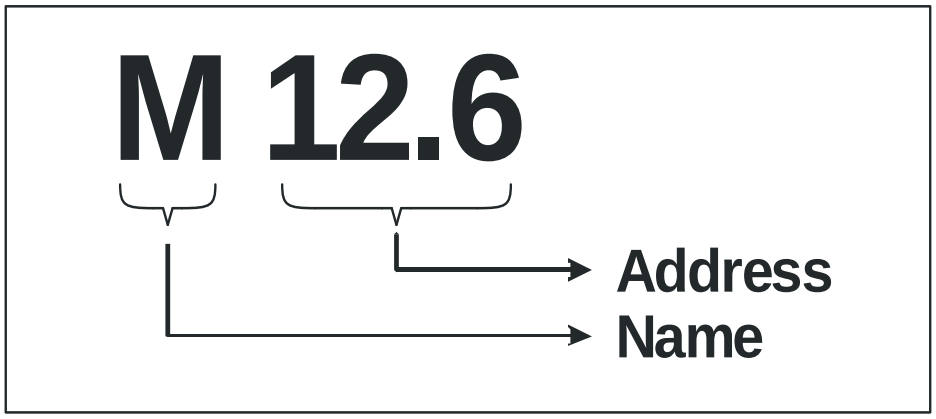
Data is organized in data blocks (DBs or DXs); 256 16-bit data words can be addressed directly in each data block. The data is located either in user memory, where it must share the available space with the user program, or in a memory area reserved exclusively for data blocks (DB RAM).

2.2 Operands, Addressing Overview

In the following chapter the most common S5 Operands are listed.

Operands Addressing

An Operand is built up from an “Address Identifier” (Name) and an “Address”.



Depending on the number of bits addressed the Operand (Variable) has different “Address Identifiers”.

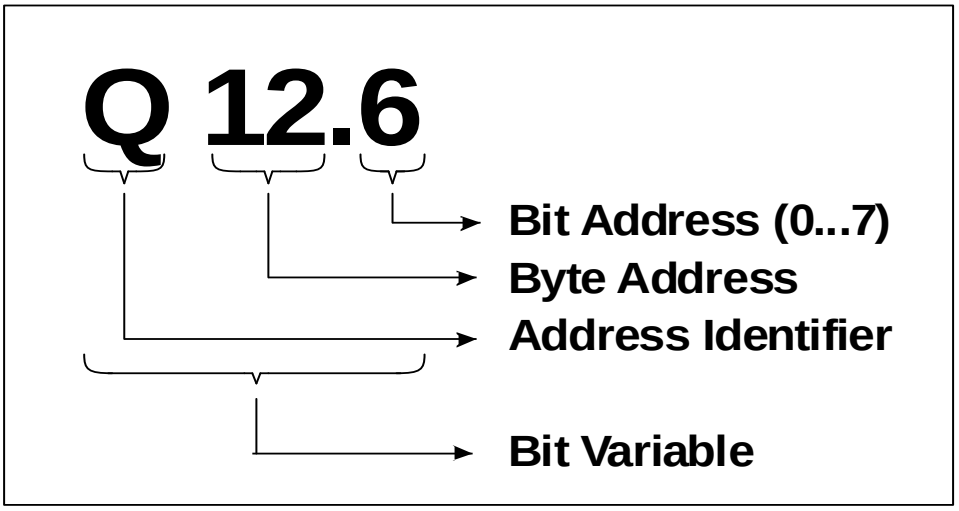
STEP®5 uses the following number of bits with Variables:

Data Width	Description	Example
1 Bit	Bit Variable (1 Bit)	I2.3; Q45.6; M34.3; DBX43.1; DIX14.6
8 Bit	Byte Variable (8 Bit)	IB12; QB45; MB23; DBB12; DIB14
16 Bit	Word Variable (16 Bit)	IW38; QW32; MW66; DBW3; DIW16
32 Bit	Double Word Variable (32 Bit)	ID55; QD43; MD62; DBD23; DID33

Bit Variables (Bit Operands)

Bit variables have an address identifier, a byte number, and – separated by a period – a bit number (Binary Address).

Addressing of an Output Variable



- The numbering of Bit Variables start with the byte number at zero for each address area. The upper limit is CPU specific.
- Bits (from the I, O, F, or S area) are numbered from 0 to 7.
- Bits (from Data Blocks, D / DX) are numbered from 0 to 15.

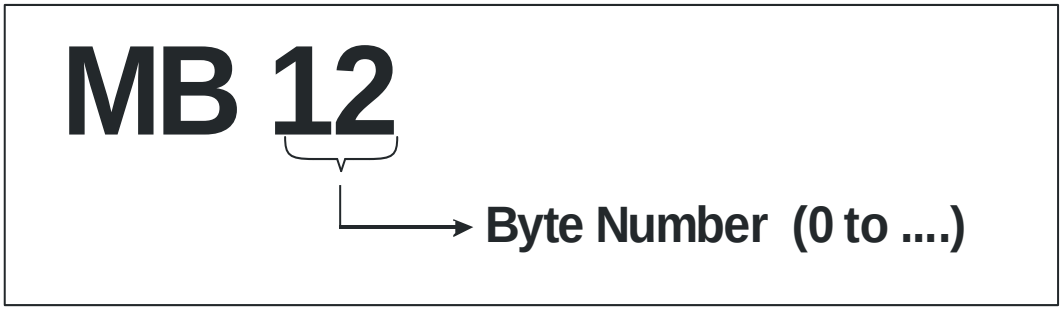
Variable	Description	Example
I	A single Bit Input from the Process Image Input Area (PII)	I 63.1
Q	A single Bit Input from the Process Image Output Area (PIQ)	Q 45.1
F	A single Bit Input from the Flag Memory Area	F 88.4
S	A single Bit Input from the Extended Flag Memory Area	S 12.7
D	A single Bit from a Data Block or Extended Data Block. Before the operand area for data can be used, the data block (DB or DX) containing the relevant operand must be selected in the program	D 74.15
T	A Bit Information from / to a Timer	T 12
C	A Bit Information from / to a Counter	C 15

Byte Variable (Byte Operands)

The absolute address of a Byte Variable consists of the address identifier and the number of the byte containing the variable.

The address identifier is supplemented with a “B” (not for Bytes from Data Blocks).

Addressing a Memory Byte



- The numbering of Byte Variables start at zero for each address area. The upper limit is CPU specific.

Variable	Description	Example
IB	Input Byte from the Process Image Input Area (PII)	IB 63
QB	Output Byte from the Process Image Output Area (PIQ)	QB 45
FY	A Byte from the Flag Memory Area	FY 88
SY	A Byte from the Extended Flag Memory Area	SY 12
DL	A Low Byte (right Byte from a Data Word) from a Data Block or Extended Data Block. Before the operand area for data can be used, the data block (DB or DX) containing the relevant operand must be selected in the program	DL 74
DH	A High Byte (left Byte from a Data Word) from a Data Block or Extended Data Block. Before the operand area for data can be used, the data block (DB or DX) containing the relevant operand must be selected in the program	DH 74
PY	Peripheral Byte (direct I/O access)	PY 123
OB	Extended Peripheral Byte(direct I/O access)	OB 234

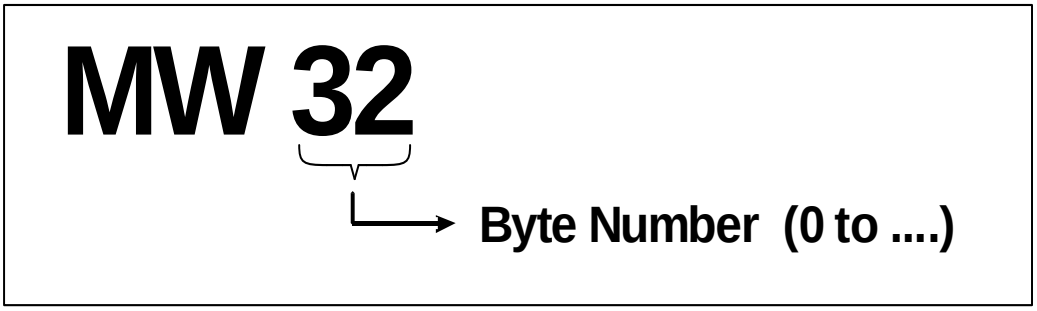
Word Variable (Word Operands)

A “Word Variable” consists of two (2) bytes, the “Low Byte” and the “High Byte”.

The absolute address of a Word Variable consists of the address identifier and the number of the byte addressing (high order byte) the word variable.

The address identifier is supplemented with a “W”.

Addressing a Memory Word



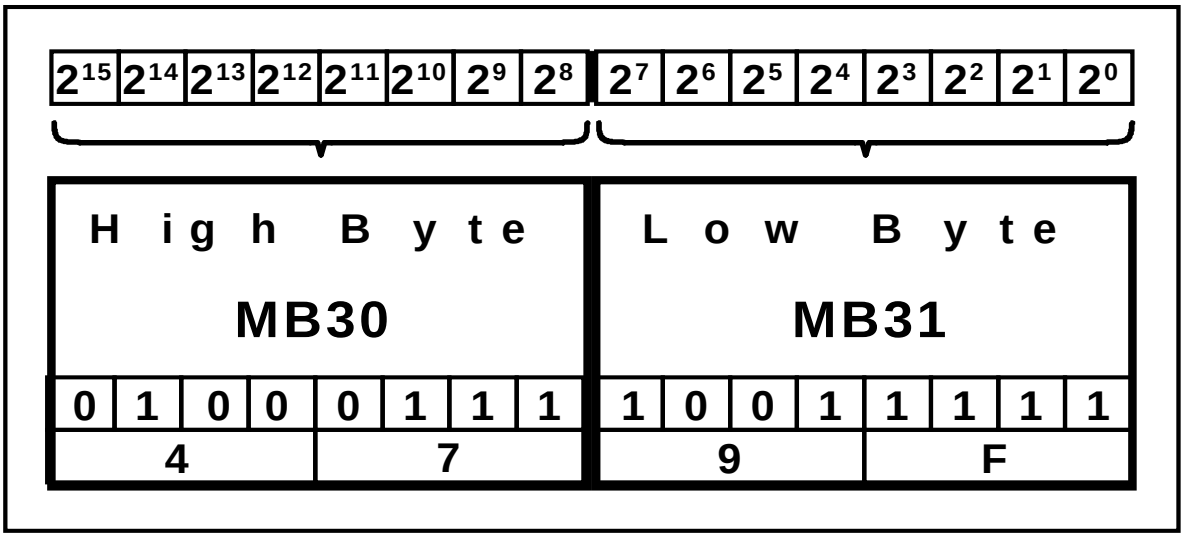
- The numbering of Word Variables start at zero for each address area. The upper limit is CPU specific.
- Data from Data Blocks are addressed with word numbers.

Variable	Description	Example
IW	Input Word from the Process Image Input Area (PII)	IW 63
QW	Output Word from the Process Image Output Area (PIQ)	QW 45
FW	A Word from the Flag Memory Area	FW 88
SW	A Word from the Extended Flag Memory Area	SW 12
DW	A Data Word from a Data Block or Extended Data Block. Before the operand area for data can be used, the data block (DB or DX) containing the relevant operand must be selected in the program	DW 74
PW	Peripheral Word (direct I/O access)	PW 123
OW	Extended Peripheral Word (direct I/O access)	OW 234

High Byte and Low Byte in a Word

Because of the “Byte Addressing” the orientation of digits within a Word is very important.

A word is divided into a High Byte and a Low Byte.



The example in the picture shows that value of 18,335 is stored in the Memory Word MW30 (hex 479F).

Note:

The **“High Byte”** specifies the **Word Variable Address**. It contains the high order digits of the Word.

The **“Low Byte”** is **the Word Variable Address +1**. It contains the low order digits of the Word.

Note:

In Data Blocks the data is organized with Word Numbers. A Data Word is made up by the **“High Byte”** and the **“Low Byte”.**

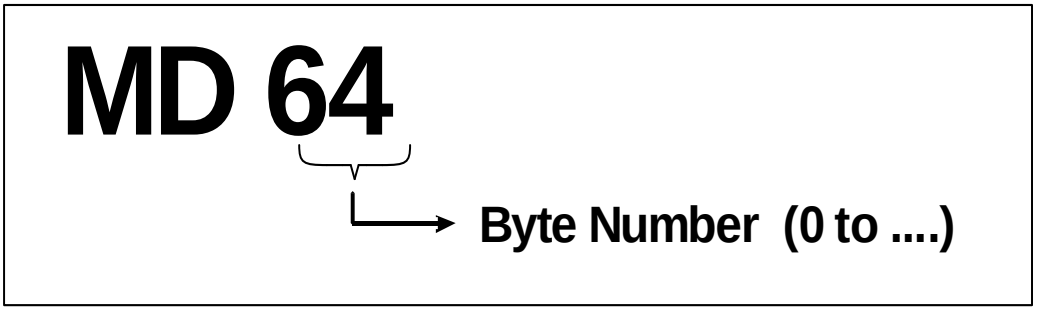
Double Word Variable

A “Double Word Variable” consists of four (4) bytes, the “Low, Low Byte” the “Low Byte”, the “High Byte” and the “High, High Byte”.

The absolute address of a Double Word Variable consists of the address identifier and the number of the byte addressing (highest byte) the double word variable.

The address identifier is supplemented with a “D”.

Addressing a Memory Double Word



- The numbering of Double Word Variables start at zero for each address area. The upper limit is CPU specific.

Variable	Description	Example
ID	Input Double Word from the Process Image Input Area (PII)	ID 63
QD	Output Double Word from the Process Image Output Area (PIQ)	QD 45
FD	A Double Word Input from the Flag Memory Area	FD 88
SD	A Double Word from the Extended Flag Memory Area	SD 12
DD	A Double Data Word from a Data Block or Extended Data Block. Before the operand area for data can be used, the data block (DB or DX) containing the relevant operand must be selected in the program	DD 74

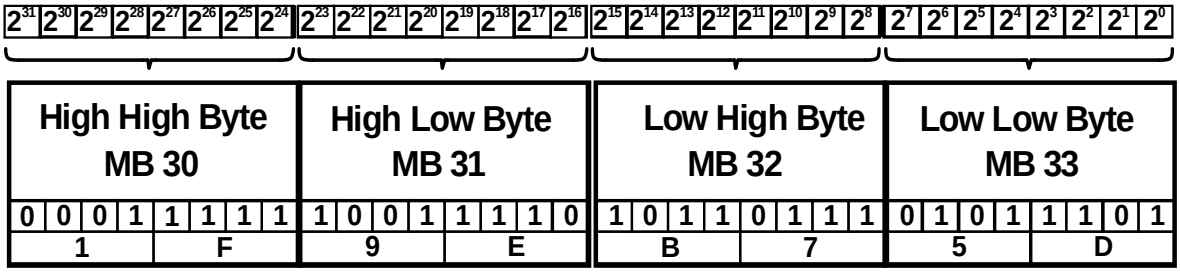
Note:

Direct I/O access (Peripheral Data) cannot be accessed using Double Words.

Byte Order in a Double Word Variable

Because of the “Byte Addressing” the orientation of digits within a Double Word is very important.

A double word is divided into a High High Byte, High Byte, a Low Byte, and a Low Low Byte.



The example in the picture shows that value of 530.495.323 (decimal) is stored in the Memory Double Word MD30 (hex 1F9E B75D).

In a Double Word Variable Floating Point values (Real) are also stored. The bit, in a Floating Point presentation, has other values

Note:

Data Blocks are organized with Words.

A Data Double Word is made up from two (2) Words.

S5 Variables (Operands)

Variables	Description	Maximum Address	Example
Qn.n	Single Output Bit	0.0 to 127.7	Q0.1
QBn	Output Byte	0 to 127	QB12
QWn	Output Word	0 to126	QW23
QDn	Output Double Word	0 to 124	QD45
In.n	Single Input Bit	0.0 to 127.7	I0.1
IBn	Input Byte	0 to 127	IB12
IWn	Input Word	0 to126	IW23
IDn	Input Double Word	0 to 124	ID45
Mn.n	Single Flag Bit	0 to 255.7	M0.1
MBn	Flag Byte	0 to 255	MB12
MWn	Flag Word	0 to 254	MW23
MDn	Flag Double Word	0 to 252	MD45
Sn.n	Single Extended Flag Bit	0 to 1023.7	L0.1
SYn	Extended Flag Byte	0 to 1023	LB12
SWn	Extended Flag Word	0 to 1022	LW23
SDn	Extended Flag Double Word	0 to 1020	LD45
PYn	Peripheral Byte	0 to 255	PQB14
PWn	Peripheral Word	0 to 254	PQW55
OBn	Extended Peripheral Byte	0 to 255	PIB12
OWn	Extended Peripheral Word	0 to 254	PIW45
Dn.n	Single Bit in Data Block (DBn / DXn)	0.0 to 255.15	D 1.0
DLn	Left Data Byte in a Data Block (DBn / DXn)	0 to 255	DL 12
DRn	Right Data Byte in a Data Block (DBn / DXn)	0 to 255	DR 12
DWn	Data Word in al Data Block (DBn / DXn)	0 to 255	DW 6
DDn	Data Double Word in a Data Block (DBn / DXn)	0 to 254	DD 5
Tn	Timer	0 to 255	T12
Cn	Counter	0 to 255	C14
OBn	Organization Block	1 to 122	OB 1
PBn	Program Block	0 to 255	PB 23
FBn	Function Block	0 to 255	FB 12
DBn	Data Block	0 to 255	DB 17
DXn	Extended Data Block	0 to 255	DX 27
BBn	Picture Blocks	0 to 255	BB 15

Overlapping of Variables

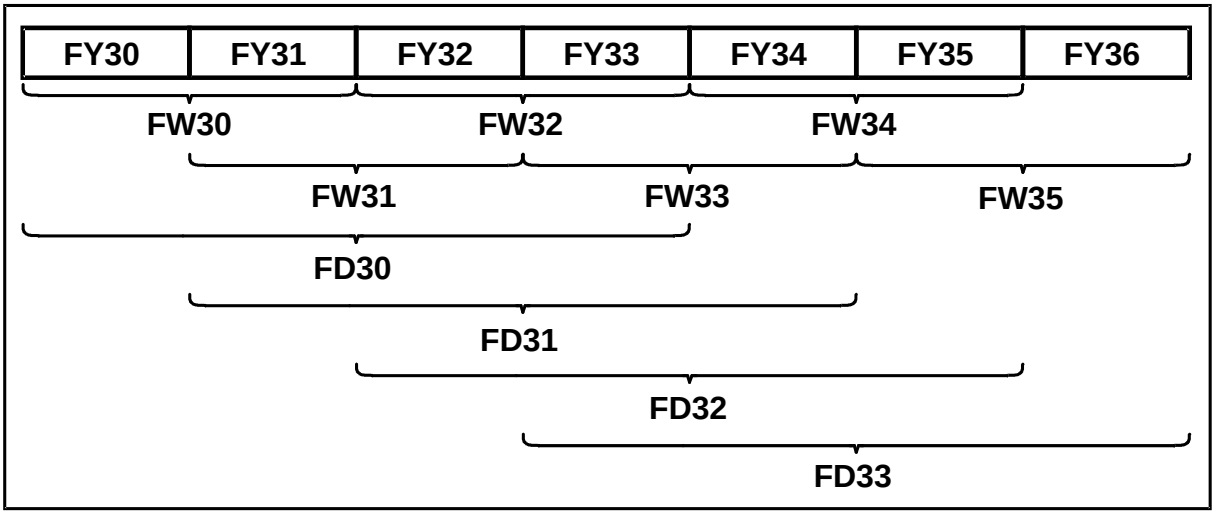
Addressing Word Variables, Double Word Variables, and Byte Variables is done in the same way. The addressing for all three types of variables is done using a byte number. Such an addressing scenario of Word Variables and Double Word Variables allows for the overlapping of data.

The Flag Word FW10 consists of the Flag Bytes FY10 and FY11.
The Flag Word FW11 consists of the Flag Bytes FY11 and FY12. Both Flag Words overlap each other with the Flag Byte FY11.

For the same reason an overlap with Double Word Variables is also possible.

The Flag Double Word FD30 consists of the Flag Bytes FY30, FYB31, FY32, and FY33.
The Flag Double Word FD31 consists of the Flag Bytes FYB31, FY32, FY33, and FY34. Both Flag Double Words overlap each other with the Flag Bytes FY30, FYB31, FY32, and FY33.

Variable Data Overlapping



Note:

To avoid data overlaps when working with Word Variables it is a good idea to use even byte address numbers only.

To avoid data overlaps when working with Double Word Variables it is a good idea to use only byte address numbers that can be divided by four (4).

2.3 Symbolic Programming

In Step® 5, symbolic programming is a very common way to address operands (variables).

Symbolic operands (such as –Test_01) may be used in place of operator identifiers and parameters (e.g. I 4.0). An assignment list must be generated, however, before symbolic operands can be used in the program.

Note:

In the **"Symbolic Table"** the user assigns a **"Symbol"** to an absolute operand (Variable). These "Symbols" can be used within the entire PLC program. The symbolic name assigned to an operand (Variable) must be unique within the whole PLC program.

- Symbolic names can be used instead of absolute addressing.
- Symbolic names must be assigned prior using them in the PLC program.

Symbolic Table Format

The maximum column width is set as **follows:**

Variable (Operand)	Symbol	Comment
The number of characters is given by the absolute address. (maximum 10 characters) **PW 22**	maximum 24 alpha numerical characters **Temperatue_Comp_2**	maximum 40 alpha numerical characters **Motor temperature of Compressor 2**

The assigned symbol may be up to 24 characters (alpha and numeric characters). The actual number of characters displayed on the screen in the LAD and CSF presentation is dependent upon the selection from the preferences dialog box.

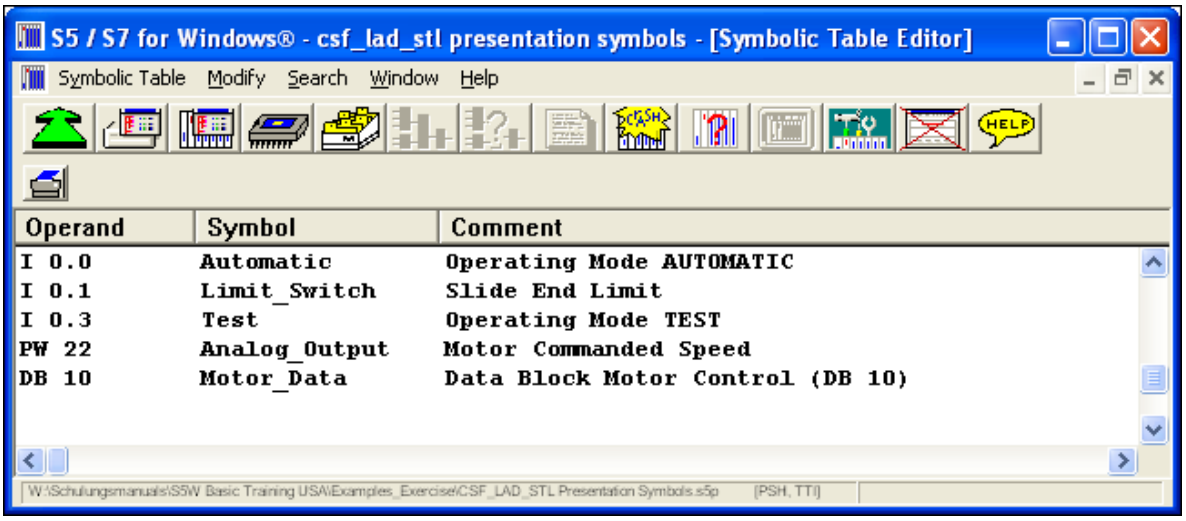
With *S5 for Windows*® up to 24 characters can be displayed. In STL and Block-STL (source text) presentation the symbolic address is always displayed with up to 24 characters,

Note:

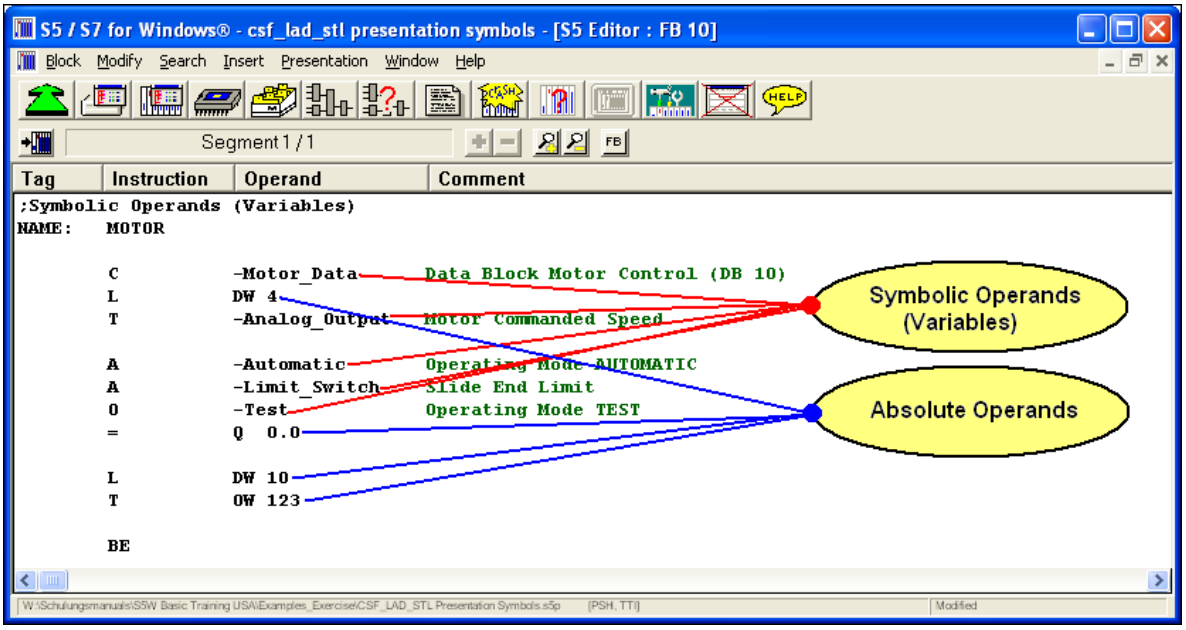
In LAD, CSF STL and Block-STL presentation a symbolic operand (variable) must be entered with a minus sign in front (**–Symbol**).

- Symbolic Operands (Variables) (**–Symbols**) are not modified with the format command (key F9).

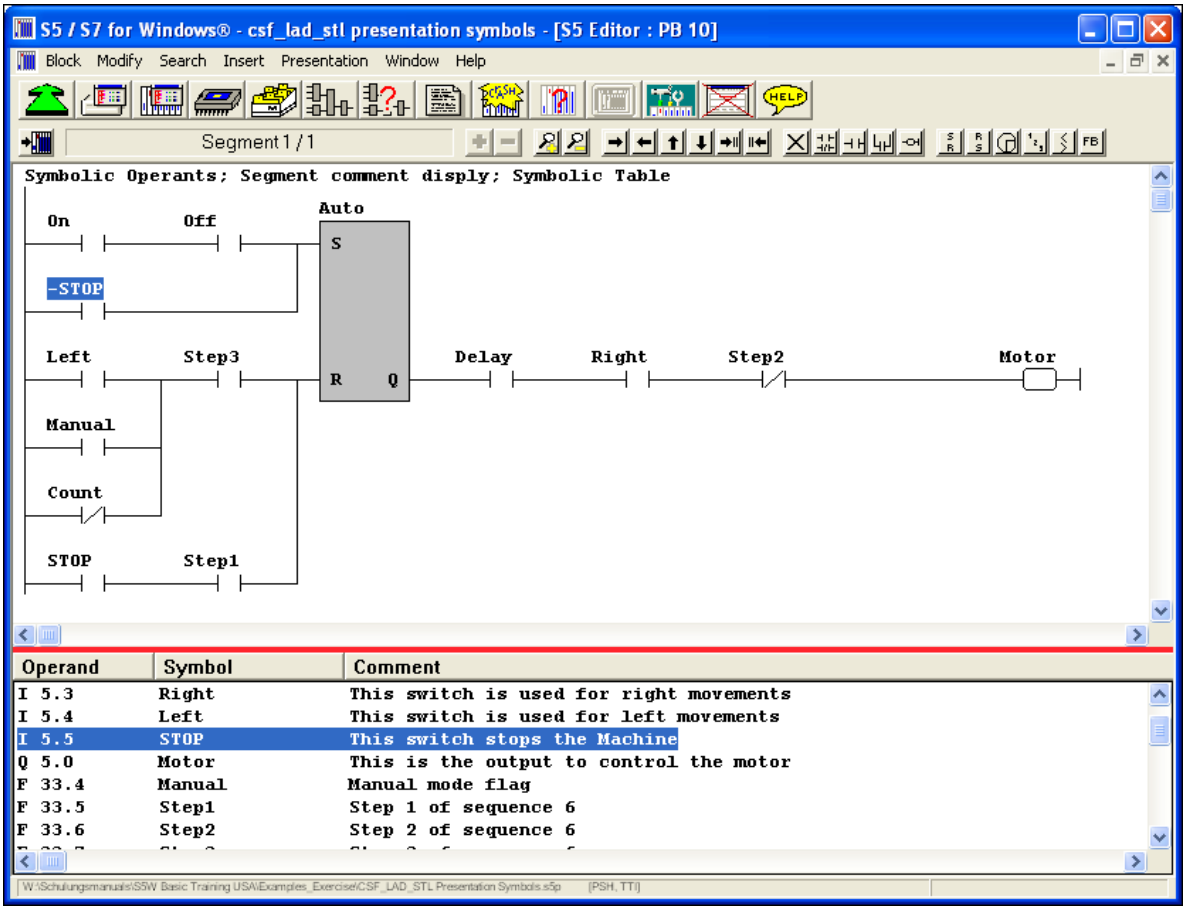
Symbolic Table (Global Variables)



Absolute / Symbolic Operands (Variables)



Symbolic Operands (Variables)



Beneath the actual logic a part of the symbolic table can be displayed.

2.4 Block Calls

Before the blocks in a user program can be processed, they must be called. These calls are special STEP 5 instructions known as “Block Calls”. Block calls can only be programmed within logic blocks (OBs, PBs, and FBs). They can be compared with jumps to a subroutine. Each jump means that the execution is branched to a different block.

The return address in the calling block is temporarily saved by the system.

The base of all “Block Calls” is OB1. Due to the complexity of the Block Call instruction, the Block to be “called” should be programmed prior inserting the call instruction into a Block.

Program Blocks (PB), Function Blocks (FB), or Organization Blocks (OB) may be called.

STEP® 5 knows two (2) instructions JC, JU), to call a Program Block (PB), Function Block (FB), or Organization Blocks (OB).

Unconditional Call

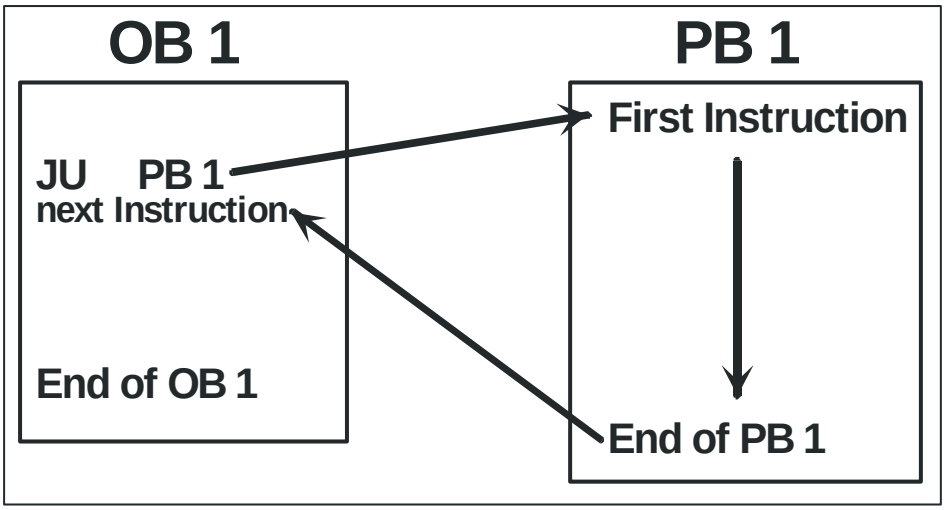
JU PBn = Unconditional Call	JU PB12
JU FBn = Unconditional Call	JU FB20
JU OBn = Unconditional Call	JU OB13

The Instruction JU <logic block identifier> (unconditional block call) calls a logic block of the PB, FB or OB type.

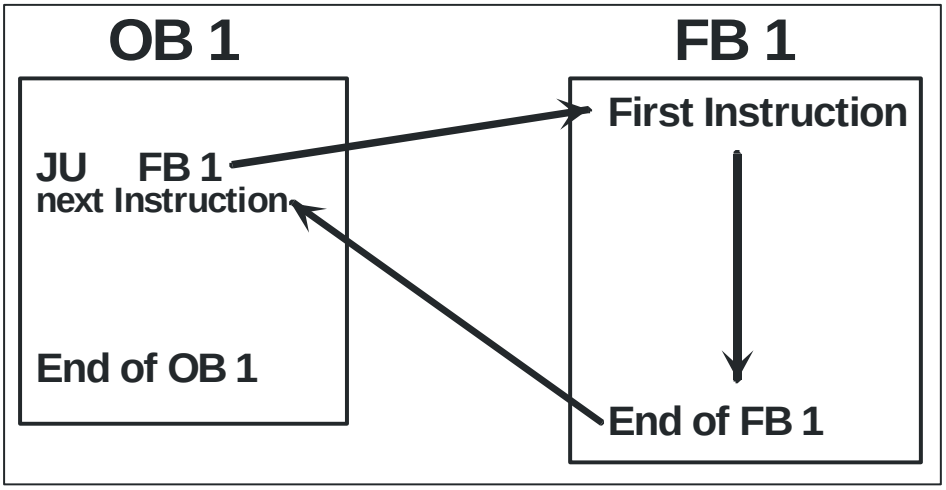
The call is executed as soon as the CPU recognizes the instruction and the program execution branches to the first instruction in the called Block.

Unconditional means the instruction is executed regardless the status of the RLO bit in the Status Word.

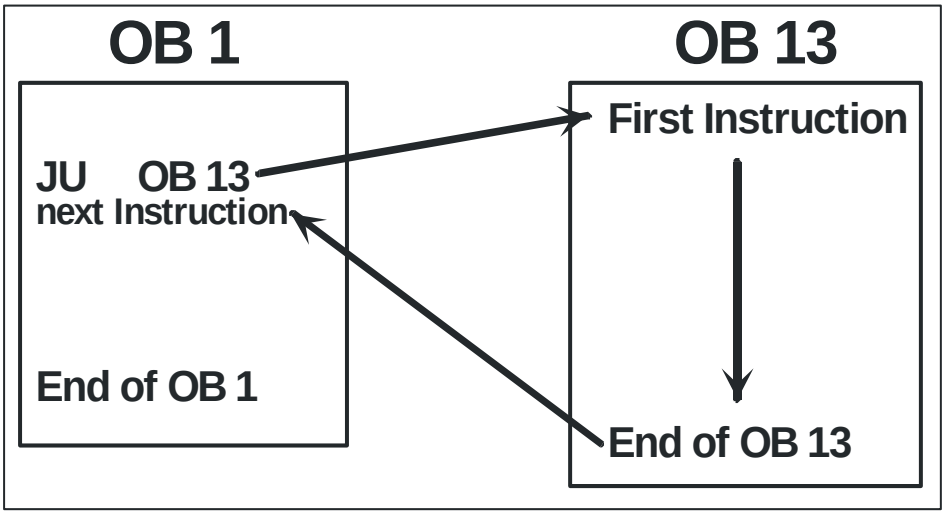
Unconditional call JU PB1



Unconditional call JU FB1



Unconditional call JU OB13



Note:

With STEP® 5 the end of a Block is not indicated by an instruction in CSF, or LAD presentation.

In the presentation STL (Statement List) the end of a Block is indicated with the “BE”.

Practice Exercise 2–1; Unconditional Call (JU)

1. Create a “New Project” (Exercise 2–1; Unconditional Call).
2. Create the Program Block PB1 (STL presentation) with the following logical instructions:

```
A   I0.0    ; Transfer status of input I0.0 into the RLO bit
A   I0.1    ; Logical AND with the status of I0.1
A   I0.2    ; Logical AND with the status of I0.2
=   Q0.0    ; Assign status of the RLO bit to output Q0.0
```

3. Create Organization Block OB1
4. Insert the unconditional block call

```
JU PB 1     // Unconditional Call
```

5. Transfer the Blocks into the S5 Test PLC
6. Test the PLC user program

Conditional Call

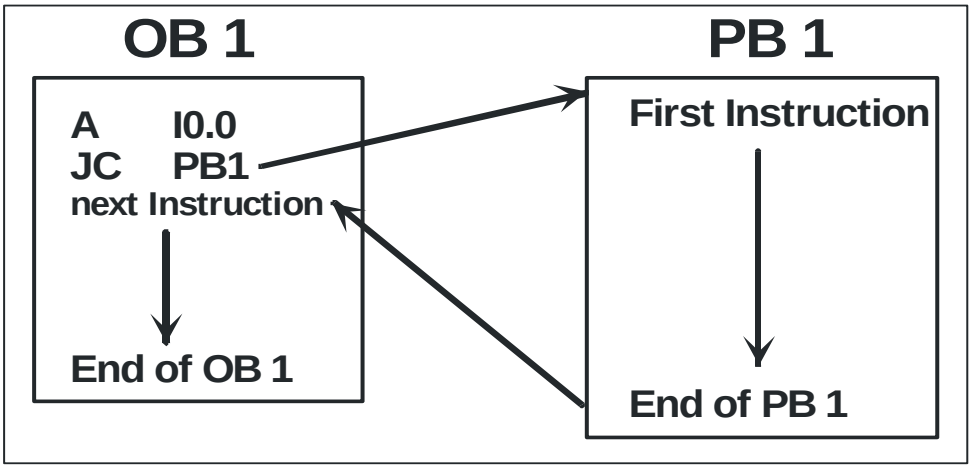
JC PBn = Unconditional Call	JC PB12
JC FBn = Unconditional Call	JC FB20
JC OBn = Unconditional Call	JC OB13

The Instruction JC <logic block identifier> (conditional block call) calls a logic block of the PB, FB or OB type if a special condition is true.

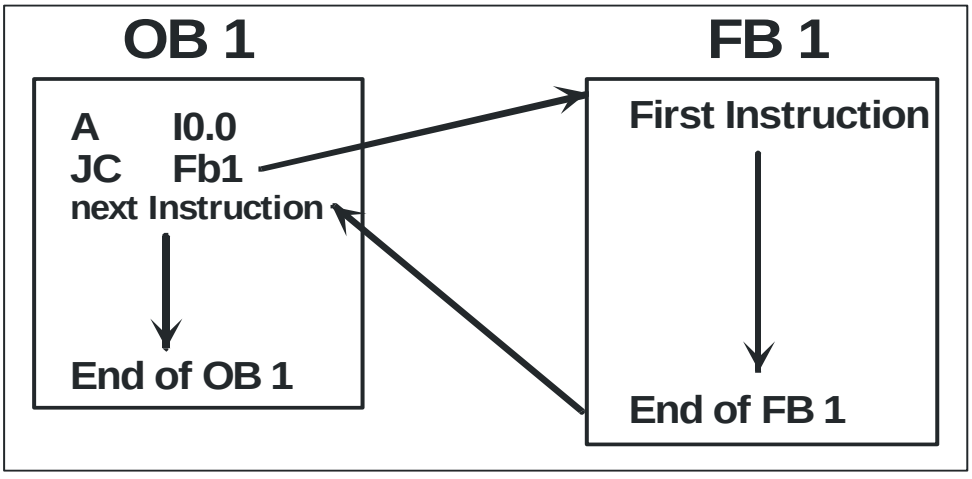
If the RLO bit in the status word is “1” and the CPU recognizes the instruction, the program execution branches to the first instruction in the called Block.

Conditional means, the instruction is only executed if the status of the RLO bit in the Status Word is “1” prior the CPU tries to execute the “conditional” instruction.

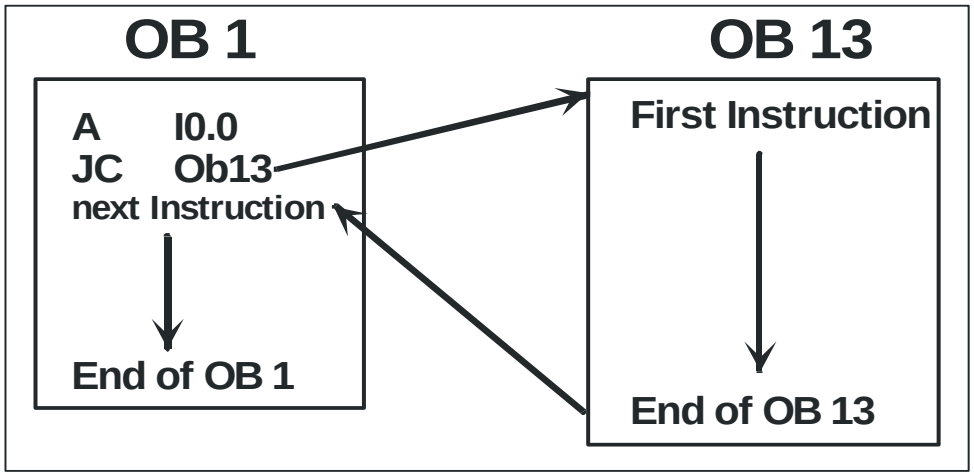
Conditional Call JC PB1



Conditional Call JC FB1



Conditional Call JC OB13



The Blocks PB1; FB1 or OB 13 are only called if the status of the input I0.0 is “1”.
After branching to the called block the instructions in this Block are executed.

Note:

Result of **L**ogical **O**peration (RLO).

The RLO (Result of Logical Operation) is the status of a bit in the “Status Word” (Bit 1) located in the system memory area of the CPU.

The RLO (Result of Logical Operation) is used within binary logical data processing.

The status of the RLO (Result of Logical Operation) can be logically connected with operands.

Also operands can set or reset the RLO depending on the status of the operand.

Practice Exercise 2–2; Conditional Call (JC)

1. Create a “New Project” (Exercise 2–2; Conditional Call)
2. Create the Program Block PB1 (STL presentation) with the following logical instructions:

```
A   I0.4    ; Transfer status of input I0.4 into the RLO bit
A   I0.5    ; logical AND with the status of I0.5
=   Q0.4    ; Assign status of the RLO bit to output Q0.4
```

3. Create Organization Block OB1
4. Insert the conditional block call

```
A   I0.0    ; Transfer status of input I0.0 into the RLO bit
A   I0.1    ; logical AND with the status of I0.1
=   Q0.0    ; Assign status of the RLO bit to output Q0.0
JC  FC 1    ; Conditional Call
```

5. Transfer the Blocks into the S5 Test PLC
6. Test the PLC user program

Calling Organization Blocks

The organization blocks form the interface between the system program (the CPU's operating system) and the user program.

Organization Blocks are divided into two categories: those called by the system program (these have the numbers 1 to 39) and those called by the users (which have the numbers 40 to 255).

Those called by the system program control cyclic, interrupt-driven and time-controlled program execution, the programmable controller's restart performance, and device error recovery procedures.

Like Program Blocks and Function Blocks, these blocks are part of the user program. The user programs these Organization Blocks himself, and can thus control the programmable controller's subsequent performance.

Only basic operations may be used in Organization Blocks of this type.

Organization Blocks with numbers above 40 represent special system program functions, and are not part of the user program; they may be neither read nor modified.

When the user wants to make use of these special functions, he simply calls the relevant Organization Block either conditionally or unconditionally, and the system program does the rest.

Calling Program Blocks

The blocks obtained by structuring the user program are called Program Blocks. When used correctly, the major Program Blocks provide an excellent overview of the user program as a whole.

The various process-related functions, e g. those of an actuator, are then written in the subordinate Program Blocks.

As a rule, a user program consists in the main of Program Blocks.

Only basic operations may be programmed in these blocks.

Calling Sequence Blocks

Sequence blocks are mainly used in conjunction with a "Sequencer control" Function Block in sequence control systems.

The Function Block then invokes the Sequence Blocks. When the GRAPH 5 software is used, the entire sequencer, including all command output and step enabling conditions or transitions, are contained in a single sequence block.

When GRAPH 5 is not used, the user can program individual sequence blocks.

In this case, their performance is identical to that of program blocks, and they may be used as such (for example, when the number of program blocks proves insufficient).

Only basic operations may be programmed in these blocks.

Calling Function Blocks

Function blocks are used to implement frequently recurring or extremely complex functions.

A Function Block represents a sequence of operations describing a self-contained function. It is present in memory only once, and can be invoked as needed by Program Blocks or other Function Blocks.

A Function Block call is "programmable", i.e. it can be assigned the parameters or operands with which the Block is to execute.

This parameter list is an integral part of a f Function Block call.

All STEP 5 operations may be programmed in Function Blocks, but the program in a Function Block must be written as a statement list.

In addition to user-written Function Blocks, a number of pre-tested or "standard" Function Blocks are also available.

Function Block FB 0 can be used as "substitute" for Organization Block OB 1

The extended function blocks (FXs), with the exception of the call statement, are handled in exactly the same way as Function Blocks (FBs).

The FX Function Blocks are not available with all S5 CPUs.

2.5 Block End (BE)

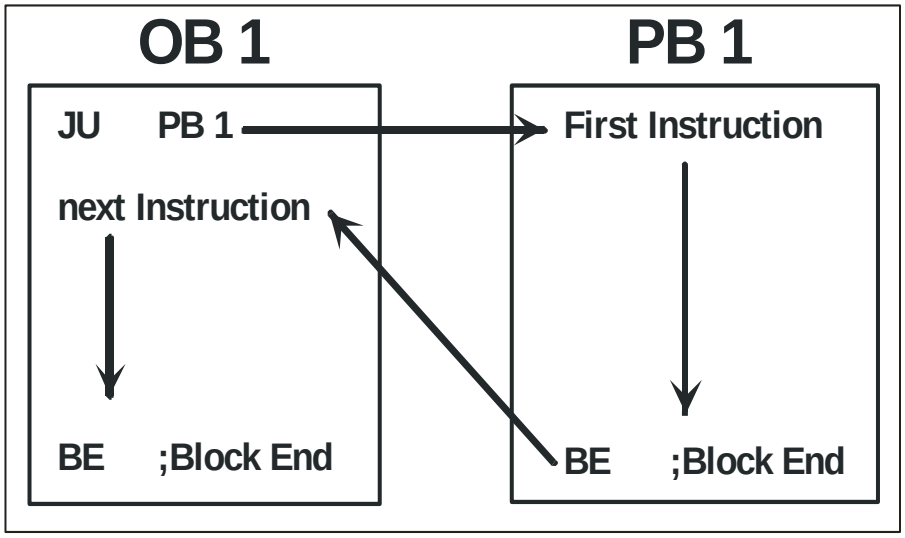
Each Block is terminated with the Block End command (BE).

The command “BE” is only displayed in the STL presentation.

Instructions placed after the Block End command (BE) will be automatically removed as soon the Block is saved.

If the CPU recognizes a Block End, the CPU terminates the program scan of the current block and causes a jump to the block that called the current block.

The program scan resumes with the first instruction that follows the block call statement in the calling program.



Example:

Due to the instruction "JU PB 1" (in OB 1) the CPU sets the CPU internal address counter to the starting address of the Program Block PB1.

The return address (old address counter contents plus one) is saved in the Block Stack (B Stack).

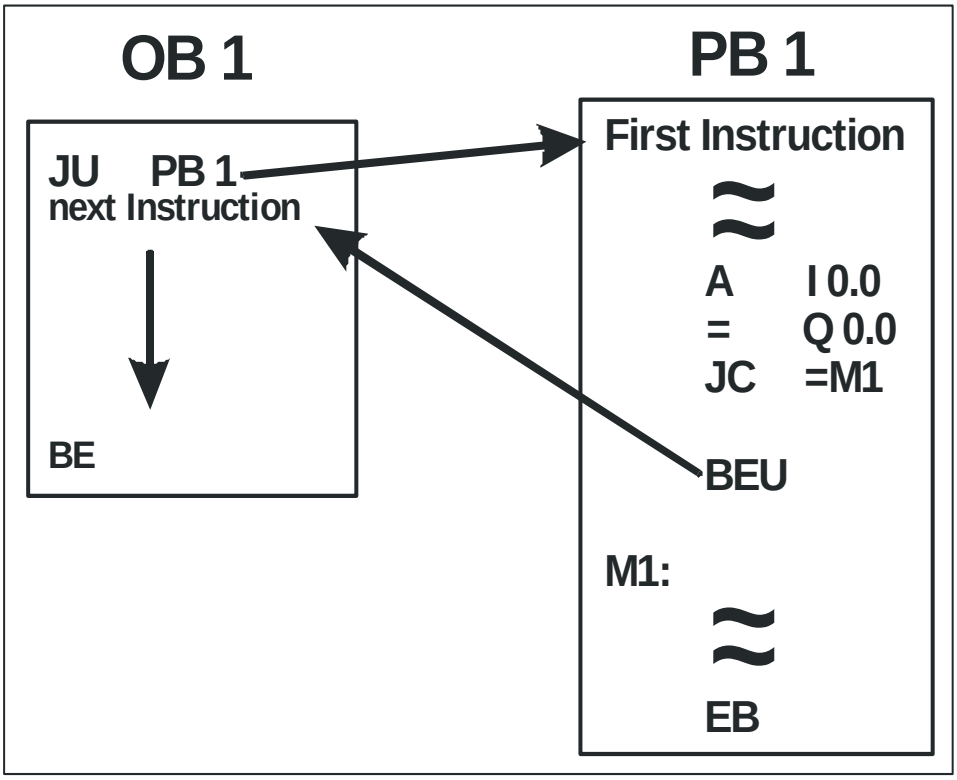
The instructions from PB1 are executed until the CPU recognizes the “BE” (Block End). Now the CPU pushes the contents of the B Stack into the address counter.

The instruction executed next is the first instruction that follows the block call.

Block End Unconditional (BEU)

BEU (**B**lock **E**nd **U**nconditional) terminates the program scan in the current block and causes a jump to the block that called the current block. The program scan resumes with the first instruction that follows the block call.

The current local data area is released and the previous local data area becomes the current local data area.



The difference between the block end being automatically inserted by the programming system and the instruction “BEU” is that logic can be programmed beyond this block end.

If the CPU recognizes the “BEU” instruction, the program scan resumes with the first instruction that follows the block call.

However, if the BEU instruction is jumped over (conditional jump “JC =M1” to the label “M1:”), the current program scan does not end and will continue starting at the jump destination within the block.

Practice Exercise 2–3; Conditional Call, BEU

1. Create a “New Project” (Exercise 2–3; Conditional Call, BEU)
2. Create Program Block PB1 (STL presentation) with the following logical instructions:

```
      A   I0.0     ; Transfer status of input I0.0 into the RLO bit
      =   Q0.0     ; Transfer status of the RLO bit into output Q0.0
      JC  =M1      ; Conditional jump to the label M1
      BEU          ; Block End Unconditional
M1:   A   I0.1     ; Transfer status of input I0.1 into the RLO bit
      A   I0.2     ; logical AND with the status of I0.2
      =   Q0.1     ; Transfer status of the RLO bit into output Q0.1
```

3. Create Organization Block OB1
4. Insert the conditional block call

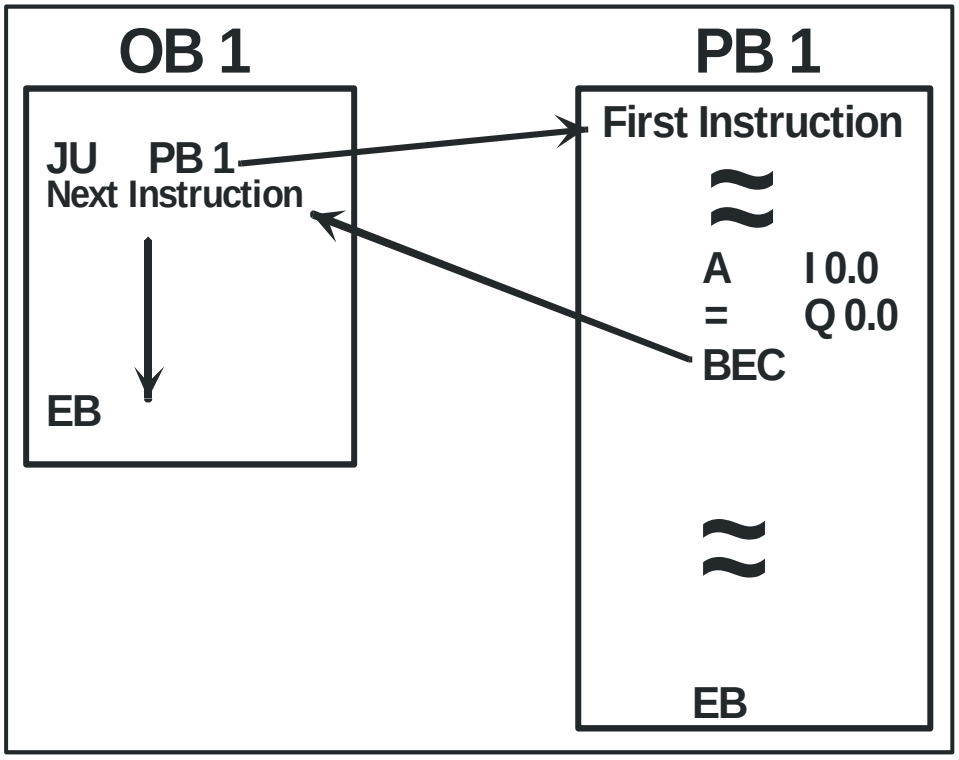
```
      A   I0.3     ; Transfer status of input I0.3 into the RLO bit
      A   I0.4     ; logical AND with the status of I0.4
      =   Q0.2     ; Transfer status of the RLO bit into output Q0.2
      JC PB 1      ; Conditional Call
```

5. Transfer the Blocks into the S5 Test PLC
6. Test the PLC user program

Block End Conditional (BEC)

If RLO = 1, then BEC (**B**lock **E**nd **C**onditional) interrupts the program scan in the current block and causes a jump to the block that called the current block. The program scan resumes with the first instruction that follows the block call.

The current local data area is released and the previous local data area becomes the current local data area.



How the CPU interprets the instruction “BEC” depends on the status of the RLO bit. If the status of the RLO = 1, then the CPU pushes the contents of the B Stack into the address counter.

The next instruction executed is the first instruction that follows the block-call in the Block, which called the block containing the BEC instruction.

Otherwise, if the status of the RLO = 0, then the BEC instruction is jumped over, the current program scan does not end and will continue with the instruction following the BEC command.

Practice Exercise 2–4; Conditional Call, BEC

1. Create a “New Project” (Exercise 2–4; Conditional Call, BEC)

2. Create the Program Block PB1 (STL presentation) with the following logical instructions:

```
A   I0.0    ; Transfer status of input I0.0 into the RLO bit
=   Q0.0    ; Transfer status of the RLO bit into output Q0.0
BEC         ; Block End Conditional
A   I0.1    ; Transfer status of input I0.1 into the RLO bit
A   I0.2    ; logical AND with the status of I0.2
=   Q0.1    ; Transfer status of the RLO bit into output Q0.1
```

3. Create Organization Block OB1

4. Insert the conditional block call

```
A   I0.3    ; Transfer status of input I0.3 into the RLO bit
A   I0.4    ; logical AND with the status of I0.4
=   Q0.3    ; Transfer status of the RLO bit into output Q0.3
JC  PB 1    ; Conditional Call
```

5. Transfer the Blocks into the S5 Test PLC

6. Test the PLC user program

3 Bit Logic Instructions

Bit Logic Instructions are described in the following chapter.

Binary Logical Instructions

Bit logic instructions work with two digits, 1 and 0. These two digits form the basis of a binary number system. The two digits 1 and 0 are called binary digits or bits. In the world of contacts and coils a 1 (true) indicates activated or energized, and a 0 (false) indicates not activated or not energized.

The bit logic instructions interpret the signal status of 1 and 0 and combine them according to Boolean logic. These combinations produce a result of 1 or 0 that is called the “result of logic operation” (RLO).

Boolean bit logic applies to the following basic instructions:

Name	Mnemonics
AND	A
OR	O
Assignment	=
Nesting Open	(
Nesting Close	)

In addition to the logic operations (O, A, X) one more operation is necessary for a signal assignment. The assignment is the output of a logical connection.

The destination of an assignment can be an Output, a Memory location (Flag, Variable) or even an Input.

The first instruction of a logic one (1) bit connection is called **"First Check"**.

In nested logic operations single expressions are separated using parenthesis ( ).

All logic connections follow the rules of the Boolean bit.

The logic operations (O, A, X) can be negated. The letter **N** following the mnemonics of the instruction indicates the negation.

Combinations of the Logical Instructions

Name	Mnemonics
AND NOT	AN
OR NOT	ON
AND - with Nesting Open	A(
OR - with Nesting Open	O(
Nesting Closed	)

Processing the Result of a Logic Operation

The result of the logic operation is the signal state in the CPU that is used for the further processing of binary signals. The RLO can be logically combined with the signal state of an operand or an operand which signal stage dependent on the RLO.

Scan statements are used to scan the signal states of operands. A scan statement also contains the directive with which the signal state scanned is to be logically combined with the RLO in the CPU:

Example:

```
A     I0.0     ;Scan input I 0.0  for "1" (TRUE) and AND
AN    I0.1     ;Scan input I 0.1  for "0" (FALSE) and AND
O     I0.2     ;Scan input I 0.2  for "1" (TRUE) and OR
ON    I0.3     ;Scan input I 0.3  for "0" (FALSE) and OR
```

Result of the scan

To be precise, it is not the actual signal state of the operand scanned that is logically combined with the RLO; instead, a result is generated from the scan. It is this result that is then processed further.

In a scan for "1", the result is identical to the signal state of the operand scanned. In a scan for "0", on the other hand, the result is the negated signal state of the operand scanned.

The result of a logical connection of two or more operands is called **RLO**.

Example RLO:

Line No.:	STL Instruction		Status of the Operand	RLO
0001	O	I 1.1	0	0
0002	O	I 1.2	1	1
0003	O	I 1.3	0	1
0004	A	I 1.4	1	1
0005	=	Q 0.4	1	1
0006				

The RLO is 1 bit information saved in a CPU register (Status Word).

The value of the RLO can be therefore "0 or "1".

If in a new logical connection (line 1), the state of the operand ("0 or "1") is transformed into the RLO bit. The state of the operand is not changed.

The first instruction in a new logical connection is called **“First Scan”**.

In the following lines the contents of the RLO are logically connected with the status of the operands (lines2 to 4).

This procedure is continued until the RLO is assigned to an operand (line 5).

Such an assignment is called **“RLO delimiting command”**.

In line 5 the RLO delimiting command (= Q0.4) is executed.

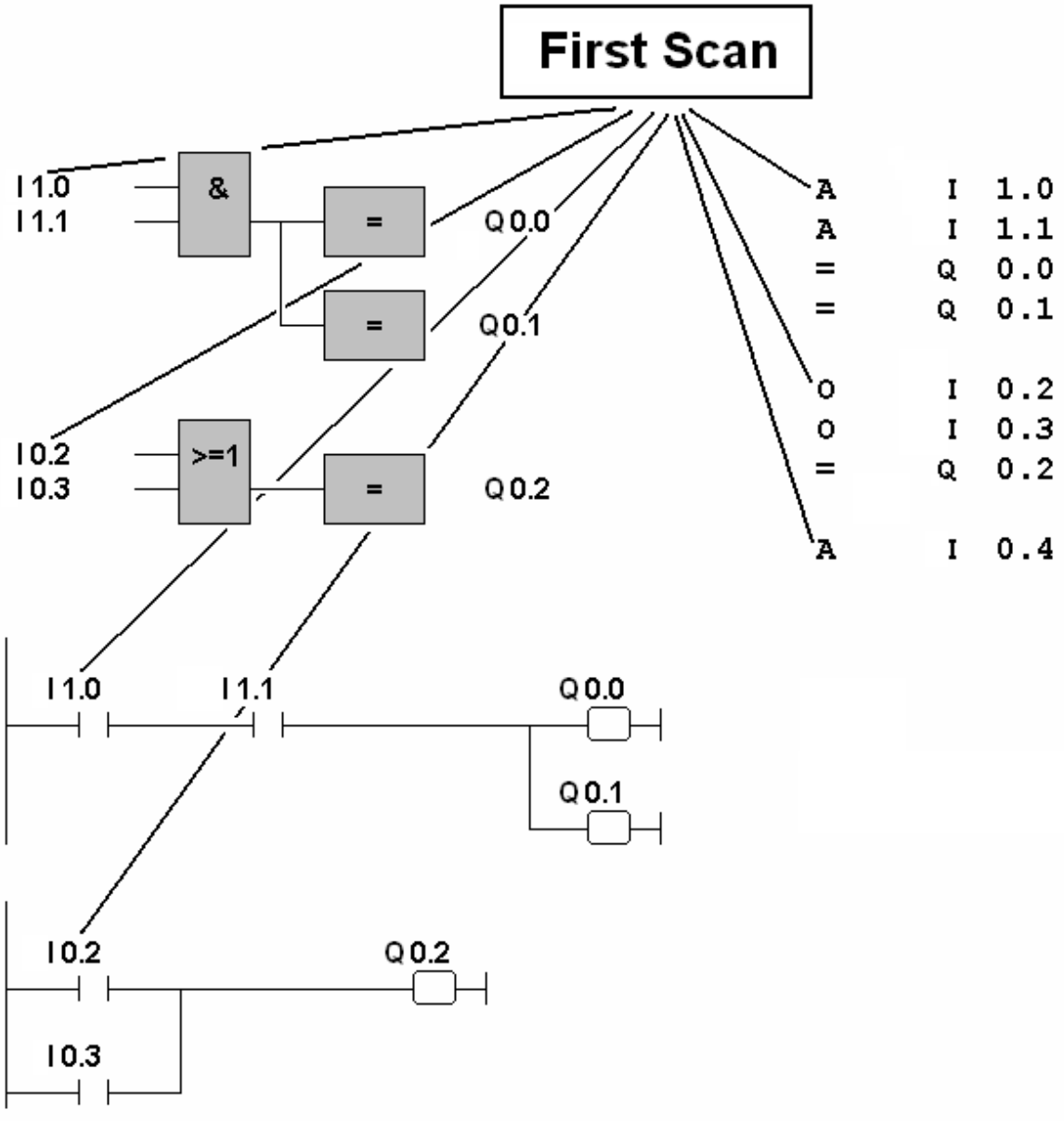
The status of the RLO is assigned to the output Q0.4.

The First Scan instruction has a special significance as the result of this statement is entered directly into the RLO as the result of the logic operation. The "old" RLO is thus lost. A first scan always represents the beginning of a logic operation.

First Scan instruction

If a new logical connection starts, the contents of the first operand are transferred to the RLO.

The RLO is independently set to the status of the first operand of a new logical connection whether the instruction has an AND (A), an OR (O), or an EXCUSVE OR (X) command.



Example:

Both examples have precisely the same behavior.

The assignment (= I 1.2, line n) delimits the RLO. In the next line (line n+1) a new logical connection starts (**first scan**).

The RLO is put on the state of the operands (I 1.3), independent of the type of command.

Line	STL Instruction		STL Instruction	
….			….	
n	=	I 1.2	=	I 1.2
n + 1	**O**	**I 1.3**	**A**	**I 1.3**
n + 2	O	I 1.4	O	I 1.4
n + 3	=	Q 1.1	=	Q 1.1
n + 4	**A**	**I 1.5**	**O**	**I 1.5**
n + 5	O	I 1.6	O	I 1.6
n + 6	=	Q 1.2	=	Q 1.2

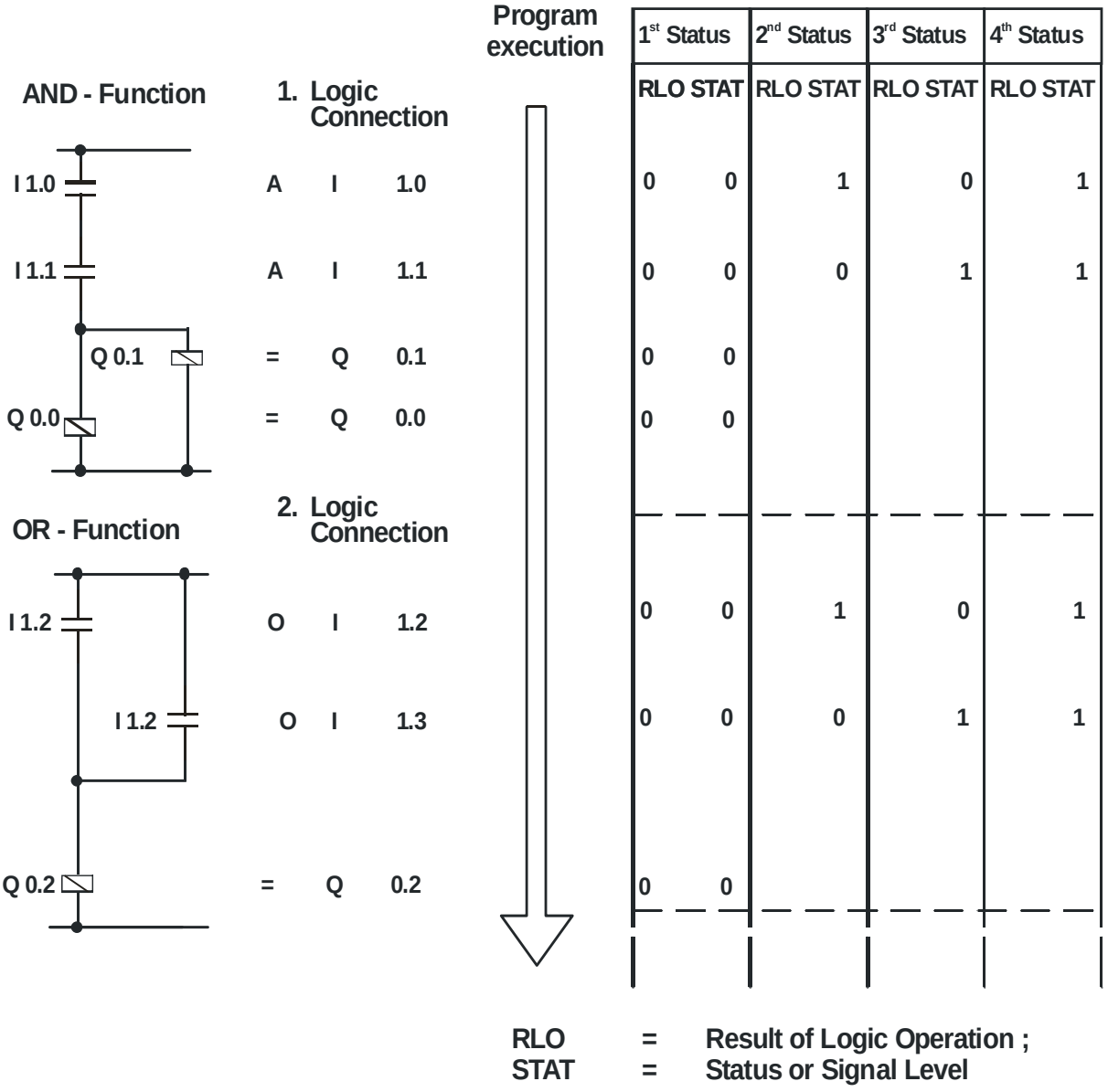
Both examples have precisely the same behavior.

The assignment (= I 1.2, line n) delimits the RLO. In the next line (line n+1) a new logical connection starts (**first scan**).

The RLO is put on the state of the operands (I 1.3), independent of the type of command.

Line	STL Instruction		STL Instruction	
….			….	
n	=	I 1.2	=	I 1.2
n + 1	**ON**	**I 1.3**	**AN**	**I 1.3**
n + 2	O	I 1.4	O	I 1.4
n + 3	=	Q 1.1	=	Q 1.1
n + 4	**ON**	**I 1.5**	**AN**	**I 1.5**
n + 5	O	I 1.6	O	I 1.6
n + 6	=	Q 1.2	=	Q 1.2
n + 7	BE		BE	

Practice Exercise 3–1; Result of the Logic Operation, Status



The result of the logic operation generated by scan statements is used as the basis for executing (RLO "1") or not executing (RLO "0") these conditional operations. The conditional operations do not change the RLO making it possible to process several conditional operations with the same RLO.

RLO delimiting

An assignment of the RLO is a delimiting instruction

The next logical instruction will start a new logical connection.

Assignments (= Q0.1; = M0.0) or SET and RESET (S. M0.0, R A0.0) are RLO delimiting commands.

RLO delimiting Instructions

In this table all types of RLO delimiting instructions STEP®7 provides are listed.

RLO delimiting Operation	Example
Assignment	= M1.1, = Q1.1
Set Instruction	S M0.1; S Q0.1
Reset Instruction	R M0.1; R Q0.1
Nesting Open / Close Instruction	A(, O(, etc
Counting Instruction	CU C1, CD C2
Timer Instruction	SP T1, SE T2 etc
Jump Instruction	JU =M001, JC =M002, (JZ, JP; JN, JM, etc.)
S5 Block Call	JC PB10; JU FB22;; etc.
Block End Instructions (Return from S5 Block Call)	BE, BEU, BEC

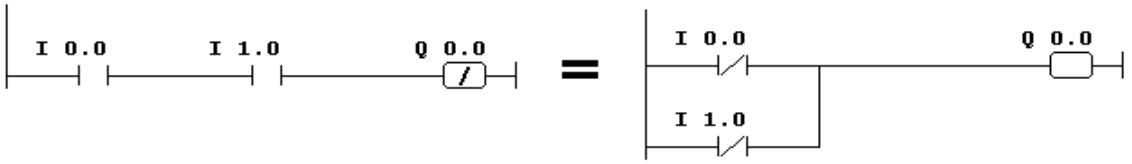
3.1 Basic Rules of Boolean Algebra

The AND instruction is executed before the OR instruction

Rule: AND before OR.

Conversion AND / OR

Executing an AND function whose output is inverted is identical with one OR function whose inputs are inverted.

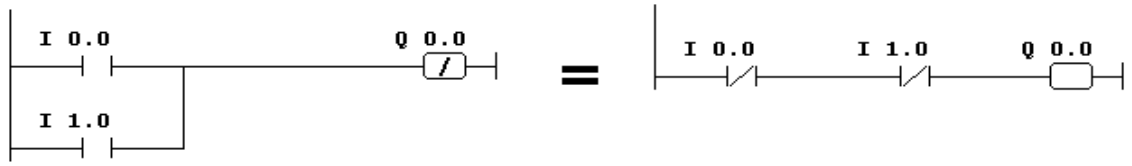


I0.0	I1.0	AND	Q0.0
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

I0.0 negated	I1.0 negated	OR Q0.0
1	1	1
1	0	1
0	1	1
0	0	0

Conversion OR / AND

Executing an OR function whose output is inverted is identical with one AND function whose inputs are inverted.



I0.0	I1.0	OR	Q0.0
0	0	0	1
0	1	1	0
1	0	1	0

I0.0 negated	I1.0 negated	AND Q0.0
1	1	1
1	0	0
0	1	0

1	1	1	0

0	0	0

Example of a Logical Connection

Four sensors (S1, S2, S3 and S4) are installed in a system. If at least two sensors detect a faulty condition a warning signal should be generated.

The logic connections necessary are listed in the table below:

	S1	S2	S3	S4	
0	0	0	0	0	
1	1	0	0	0	
2	0	1	0	0	
3	1	1	0	0	**Warning**
4	0	0	1	0	
5	1	0	1	0	**Warning**
6	0	1	1	0	**Warning**
7	1	1	1	0	**Warning**
8	0	0	0	1	
9	1	0	0	1	**Warning**
10	0	1	0	1	**Warning**
11	1	1	0	1	**Warning**
12	0	0	1	1	**Warning**
13	1	0	1	1	**Warning**
14	0	1	1	1	**Warning**
15	1	1	1	1	**Warning**

From the table above the logic functions can be programmed. From the sixteen (16) possibilities eleven (11) indicate a warning.

It is also possible to use the inverted function. A warning is always generated except for the following conditions:

	S1	S2	S3	S4	
0	0	0	0	0	**No Warning**
1	1	0	0	0	**No Warning**
2	0	1	0	0	**No Warning**
4	0	0	1	0	**No Warning**
8	0	0	0	1	**No Warning**

A AND Function

Format: A <Bit>

Address	Data type	Memory area
<Bit>	BOOL	I, Q, M, L, D, T, C

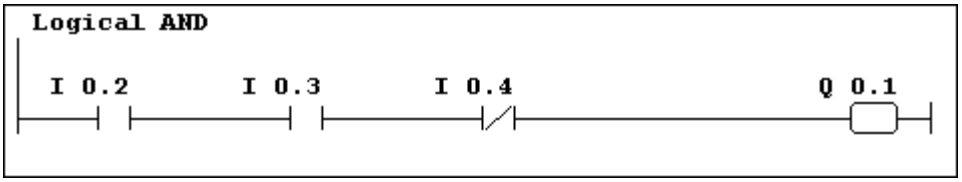
The AND (**A**) checks whether the state of the addressed bit is "1" (TRUE) or "0" (FALSE), and AND's the test result with the RLO.

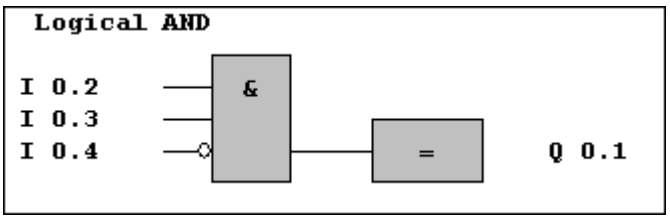
Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	x	x	x	1

Example:

STL Program		Relay Logic	
		Power rail	
A	I 0.0	I 0.0 signal state 1	NO contact
A	I 0.1	I 0.1 signal state 1	NC contact
=	Q 0.0	Q 0.0 signal state 1	Coil






```
; Logical AND
      A    I  0.2
      A    I  0.3
      AN   I  0.4
      =    Q  0.1
```

In the example above the inputs I 0.2 and I 0.3 must have the signal state of "1" and the input I0.4 must have the signal state of "1" to activate the output Q0.1

Practice Exercise 3–2; Logical AND

A compressor K1 should be switched on if the following conditions are fulfilled:

ON – Switch S1 in its ON position.

Pressure switch S2 must be closed (operated).

The security valve S3 must be closed (not operated).

The oil pressure switch S4 must be closed (operated).

Function	PLC Operand
S1 has the signal state of "1" if operated	I 0.0
S2 has the signal state of "1" if operated	I 0.1
S3 has the signal state of "0" if operated	I 0.2
S4 has the signal state of "1" if operated	I 0.3
K1	Q 0.0

Tasks:

1. Write a PLC program with the S5 Blocks PB10 and OB1.
2. Transfer of the program into the S5 TEST PLC.
3. Test the PLC program.

IF the input I 0.0 is **"ON"**

AND the input I 0.1 is **"ON"**

AND NOT the input I 0.2 is **"ON"**

AND the input I 0.3 is **"ON"**

IS the output Q 0.0 **"ON"**

With:

IF	**A/O**
AND	**A**
AND NOT	**AN**
IS	**=**

O OR Function

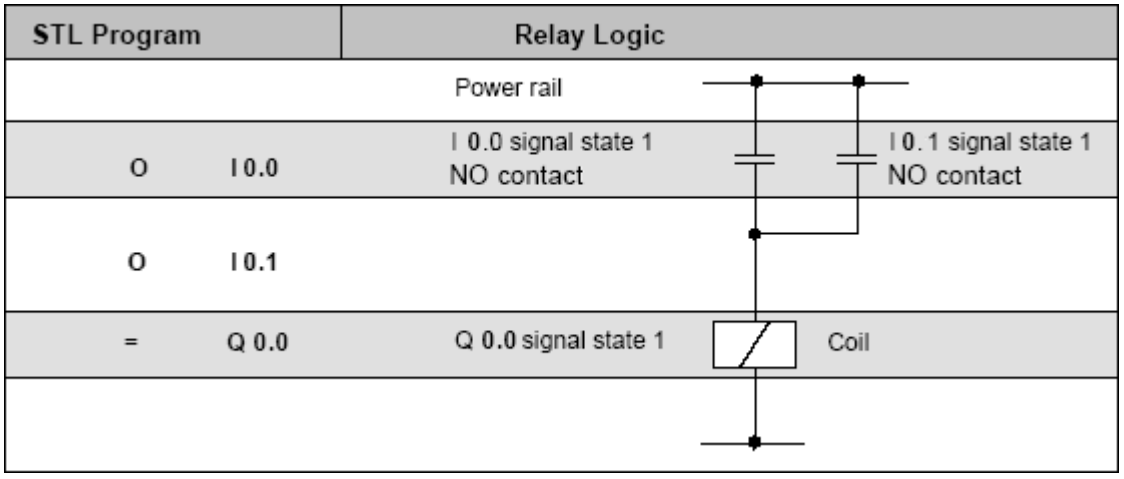
Format: O <Bit>

Address	Data type	Memory area
<Bit>	BOOL	I, Q, M, L, D, T, C

The **OR (O) Function** checks whether the state of the addressed bit is "1" or "0" (FALSE), and OR's the test result with the RLO.

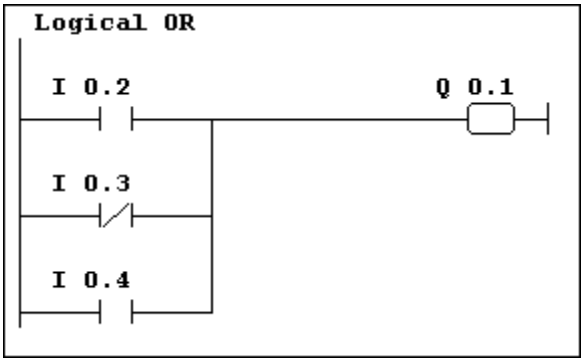
Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	x	x	x	1

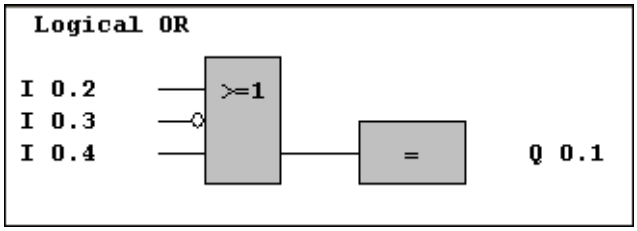


The output of an OR function has the signal state of "1" if one or several inputs have the signal state of "1".

Only if all inputs have the signal state of "0", the output of an OR function has the signal state of "0".



O OR Function (continued)




```
; Logical OR

      O     I  0.2
      ON    I  0.3
      O     I  0.4
      =     Q  0.1
```

In the example above one of the inputs I 0.2 and I 0.4 must have the signal state of “1” or the input I0.3 has the signal state of “0” to activate the output Q0.1

Practice Exercise 3–3; Logical OR

A message should be displayed if one of the following conditions are true:

Pressure switch S1 indicates “no Pressure”

The security pressure switch S2 indicates “Pressure to high”

The motor temperature switch S3 indicates “Temperature to high”

The compressor temperature switch S4 indicates “Temperature to high”

Function	PLC Operand
S1 has the signal state of "1" if “no Pressure”	I 0.0
S2 has the signal state of "1" if “Pressure to high”	I 0.1
S3 has the signal state of "0" if “Temperature to high”	I 0.2
S4 has the signal state of "0" if “Temperature to high”	I 0.3
K1	Q 0.0

Tasks:

1. Write a PLC program with the S5 Blocks PB10 and OB1.
2. Transfer of the program into the S5 TEST PLC.
3. Test the PLC program.

IF the input I 0.0 is **“ON”**

OR the input I 0.1 is **“ON”**

OR NOT the input I 0.2 is **“ON”**

OR NOT the input I 0.3 is **“ON”**

IS the output Q 0.0 **“ON”**

With:

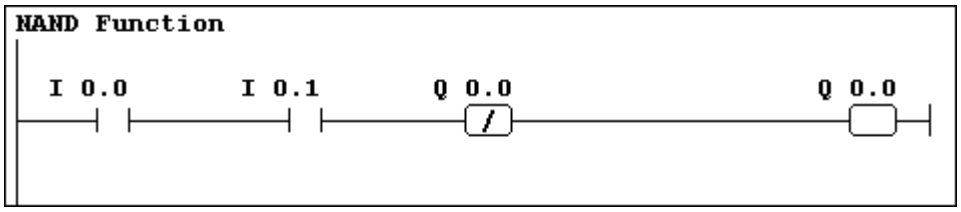
IF	**A/O**
OR	**O**
OR NOT	**AN**
IS	**=**

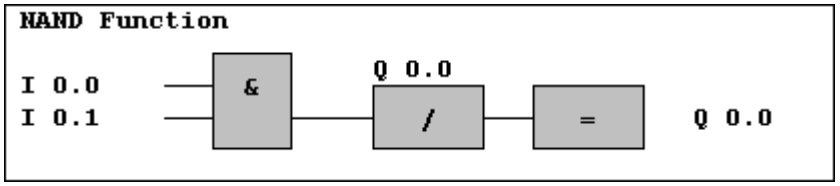
NAND Function

A NAND function is an AND function with a negated output. STEP® 5 does not know a special NAND function.

The following logical connections are possible to represent a NAND function:

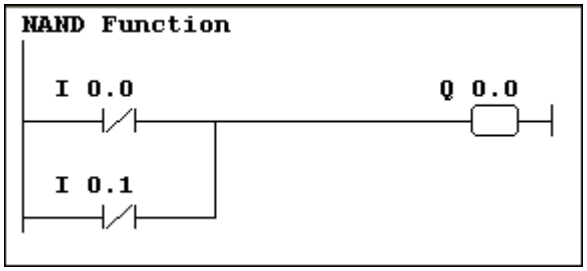
```
;NAND Function
        A           I  0.0      ; Transfer status of input I0.0 into the RLO bit
        A           I  0.1      ; logical AND with the status of I0.1
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        AN          Q  0.0      ; Transfer inverted status of output Q0.0 into the RLO bit
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        BE
```

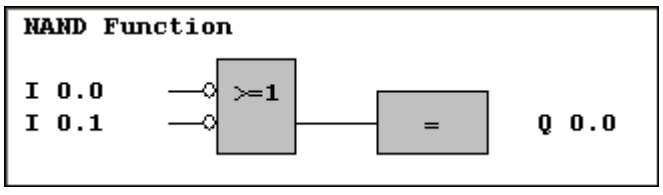




Using the Boolean Algebra Rule AND / OR Conversion

```
;NAND Function
        ON          I  0.0      ; Transfer inverted status of input I0.0 into the RLO bit
        ON          I  0.1      ; logical OR with the inverted status of I0.1
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        BE
```



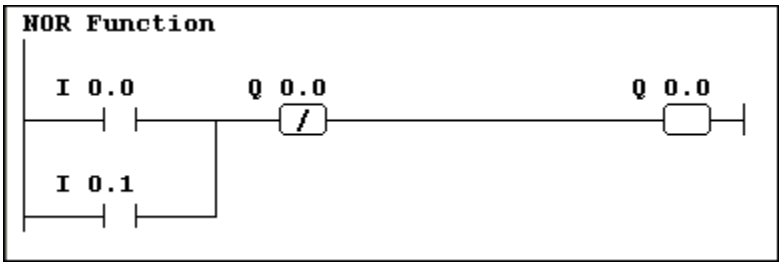


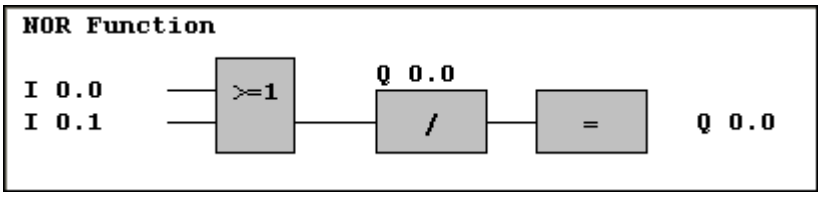
NOR Function

A NOR function is an OR function with a negated output. STEP® 5 does not know a special NOR function.

The following logical connections are possible to represent a NOR function:

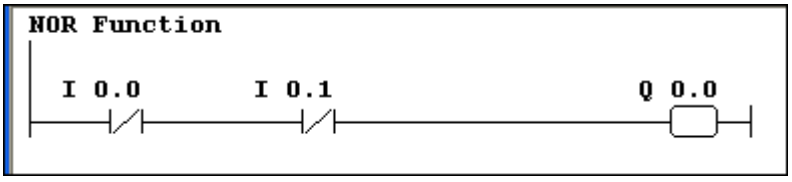
```
;NOR Function
        O           I  0.0      ; Transfer status of input I0.0 into the RLO bit
        O           I  0.1      ; logical OR with the status of I0.1
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        AN          Q  0.0      ; Transfer inverted status of output Q0.0 into the RLO bit
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        BE
```

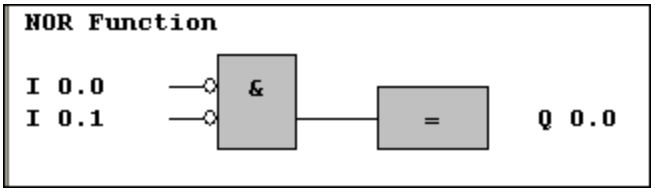




Using the Boolean Algebra Rule AND / OR Conversion

```
;NOR Function
        AN          I  0.0      ; Transfer inverted status of input I0.0 into the RLO bit
        AN          I  0.1      ; logical AND with the inverted status of I0.1
        =           Q  0.0      ; Transfer status of the RLO bit into output Q0.0
        BE
```





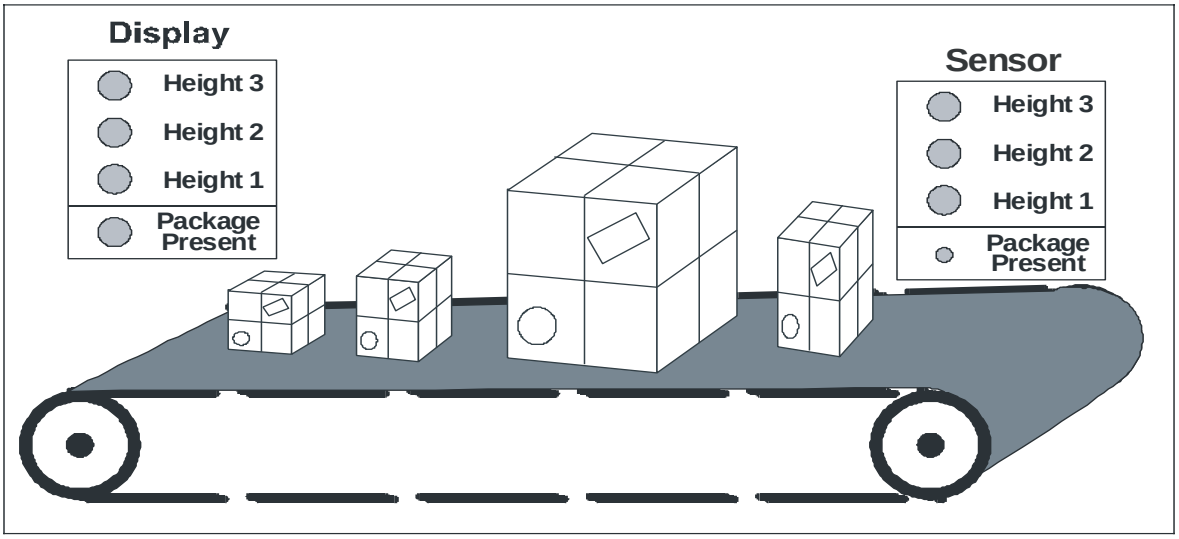
Practice Exercise 3–4; Conveyer Belt, Package Height

The height of packages transported on a conveyer belt should be tested.

As soon as the sensors detect the height of a package the height should be indicated on a lamp display.

Only one (1) "height" should be displayed (one lamp only). The lamp should stay on until a new package appears before the sensors (Call of PB10, only if package present – Conditional Call).

The actual logic should be programmed in the S5 Block PB10.
A conditional call may be used to call the Function PB10 from the OB1. The condition is true only if the sensor has recognized a package (Sensor). The sensor “Signal Package Present” should be made visible with a lamp.



Operand	Explanation
I 0.0	Sensor Package Present
I 0.1	Sensor Height 1
I 0.2	Sensor Height 2
I 0.3	Sensor Height 3
Q 0.1	Lamp Height 1
Q 0.2	Lamp Height 2
Q 0.3	Lamp Height 3
Q 0.0	Lamp Package Present

Tasks:

1. Write a PLC program with the S5 Blocks PB10 and OB1.
2. Transfer of the program into the S5 TEST PLC.
3. Test the PLC program.

Block PB 1

IF the sensor "Height 1" is **“ON”**

AND NOT the Sensor "Height 2" is **“ON”**

AND NOT the Sensor "Height 3" is **“ON”**

IS the Lamp "Height 1" **“ON”.**

IF the sensor "Height 1" is **“ON”**

AND the Sensor "Height 2" is **“ON”**

AND NOT the Sensor "Height 3" is **“ON”**

IS the Lamp "Height 2" **“ON”.**

IF the sensor "Height 1" is **“ON”**

AND the Sensor "Height 2" is **“ON”**

AND the Sensor "Height 3" is **“ON”**

IS the Lamp "Height 1" **“ON”.**

Block OB1

IF the “Sensor Package Present” is **“ON”**

IS the Lamp "Sensor Active” **“ON”**

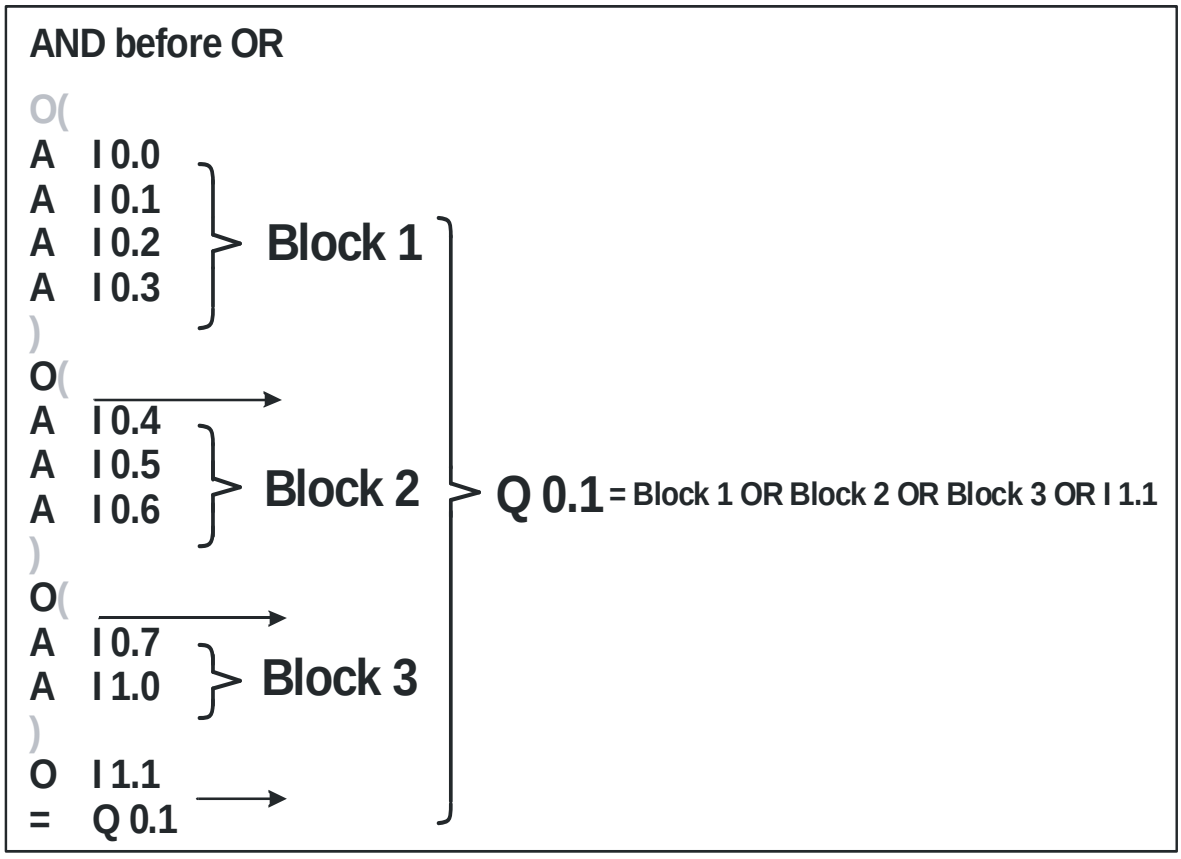
Only then the Block PB1 **is called.**

With:

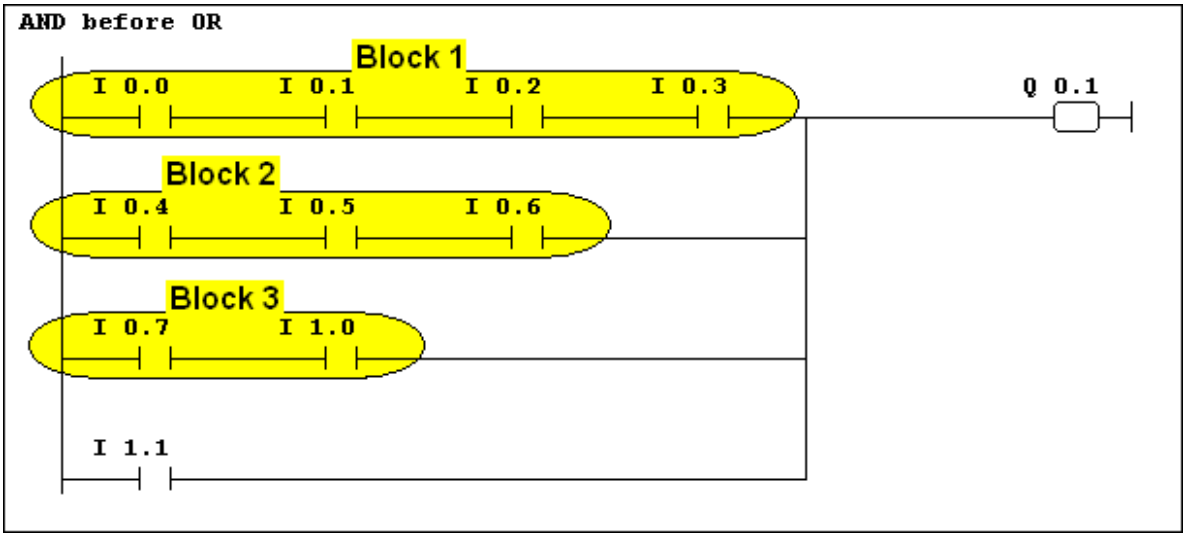
IF	**A/O**
AND NOT	**AN**
AND	**A**
IS	**=**
Only then the Block is called	**JC**

AND before OR

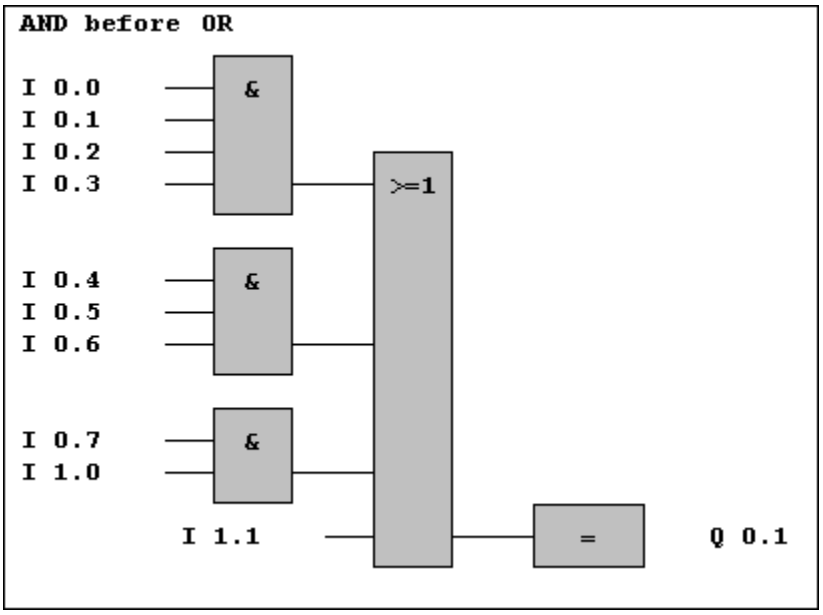
The O function performs a logical OR instruction on AND functions according to the rule: AND before OR.



The “Blocks” indicated in STL are marked in LAD.



The "Blocks" indicated in STL can also be recognized in CSF.



AND before OR

No Nesting required

Programming an AND before OR function in STL the AND connections do not be put into parenthesis.

O(Or with Nesting Open

Format: **O(**

O((OR nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible.

Status Word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	1	-	0

Practice Exercise 3–5; AND before OR

A Relay K1 (Q0.1) is energized if the following conditions are true:

- The signal state of the switch at input I0.0 is “On” and
- the signal state of the switch at input I0.1 is “Off” and
- the signal state of the switch at input I0.2 is “On”.

Also the relay should be energized independent of the signals at I0.0, I0.1 and I0.2. if the switch at the input I0.3 and the switch at the input I0.4 are “On” or the switch at the input I0.5 and the switch at the input I0.6 are “On”.

Tasks:

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

Practice Exercise 5–4; AND before OR

IF the input I0.0 is **“ON”**

AND NOT the input I0.1 is **“ON”**

AND the input I0.2 is **“ON”**

OR

IF the input I0.3 is **“ON”**

AND the input I0.4 is **“ON”**

OR

IF the input I0.4 is **“ON”**

AND the input I0.4 is **“ON”**

IS the output (relay) **“ON”.**

With:

IF	**A/O**
AND NOT	**AN**
AND	**A**
OR	**O**
IS	**=**

OR before AND

A(And with Nesting Open

Format: **A(**

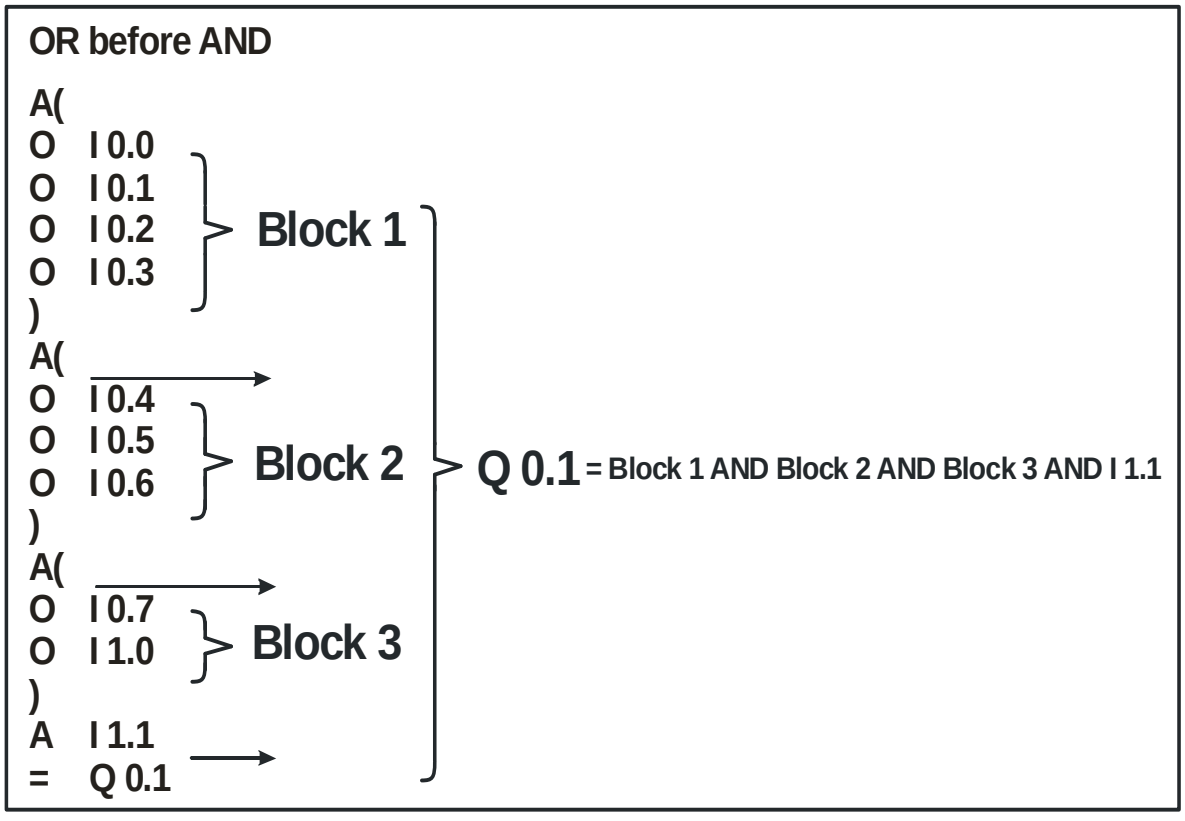
A((AND nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible.

Status Word

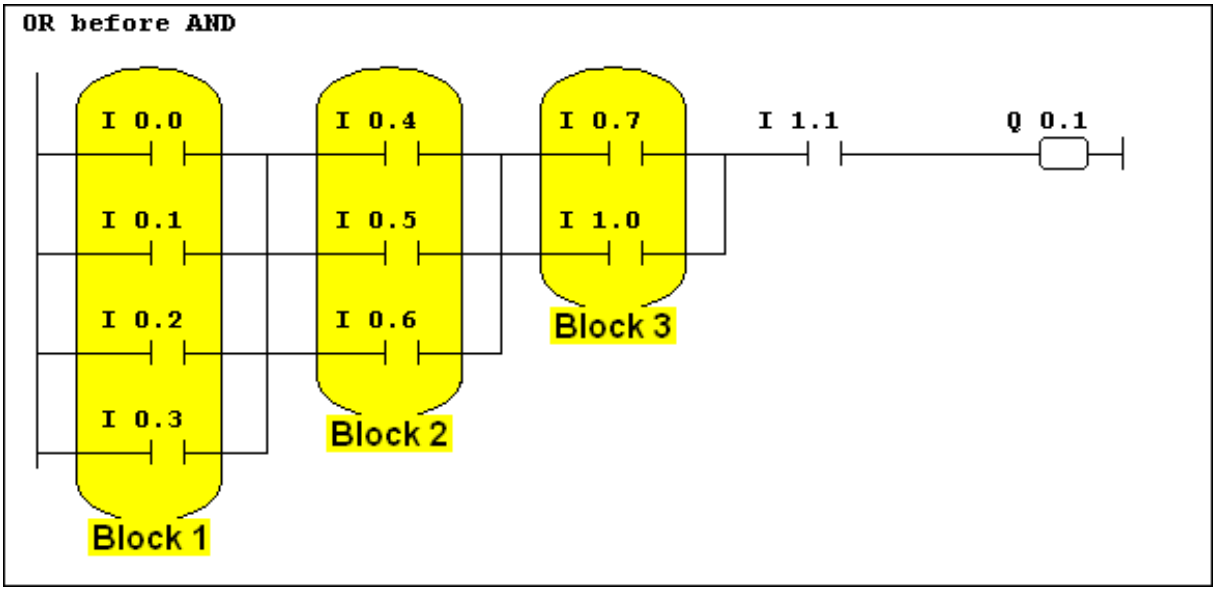
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	1	-	0

Programming an OR before AND function in STL the OR connections must be put into parenthesis.

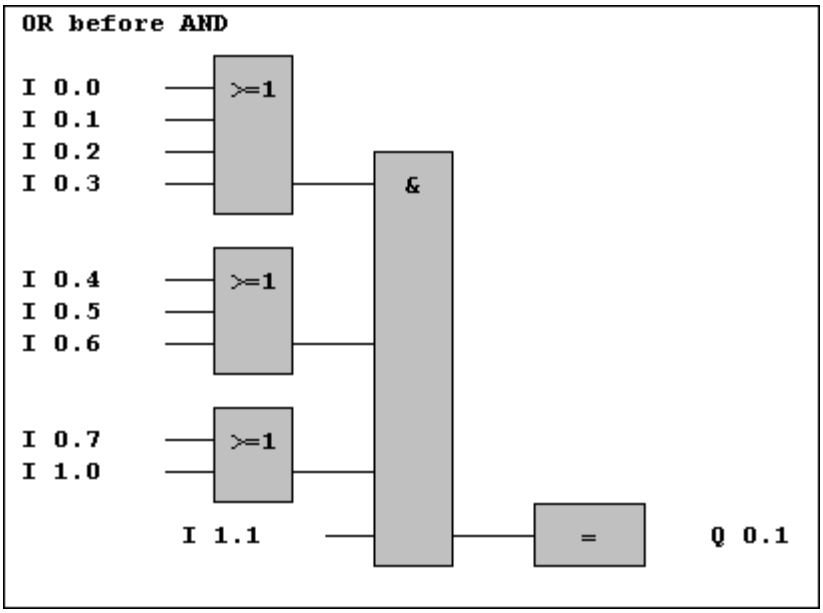
The function A((AND nesting open) is an “RLO delimiting command” therefore a new logical connection starts with the first instruction after the parenthesis. The nesting close is not a RLO delimiting command.



The “Blocks” indicated in STL are marked in LAD.



The “Blocks” indicated in STL can also be recognized in CSF.



Practice Exercise 3–6; OR before AND

A Relay K1 (Q0.1) is energized if

- The signal state of the switch at input I0.0 is “On” or
- the signal state of the switch at input I0.1 is “Off” or
- the signal state of the switch at input I0.2 is “On”.

Also the switch at the input I0.3 or the switch at the input I0.4 are “On” and the switch at the input I0.5 or the switch at the input I0.6 are “On”.

Tasks:

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

IF the input I0.0 is **“ON”**
OR NOT the input I0.1 is **“ON”**
OR the input I0.2 is **“ON”**
UND
IF the input I0.3 is **“ON”**
OR the input I0.4 is **“ON”**
UND
IF the input I0.5 is **“ON”**
OR the input I0.6 is **“ON”**
IS the output Q0.1 **“ON”.**

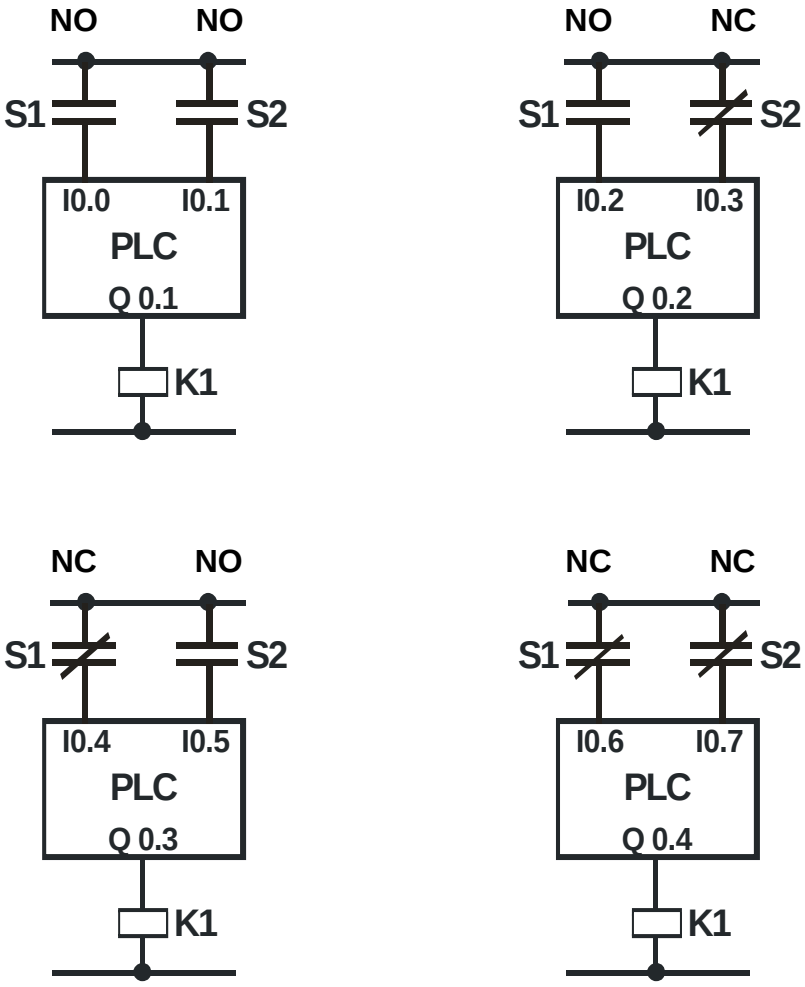
With:

IF	**A/O**
OR	**O**
OR NOT	**ON**
AND	**A(**
	)
IS	**=**

Practice Exercise 3–7; Normally Open (NO), Normally Closed (NC)

The relay K1 should be energized if the Switch 1 (S1) is operated and the Switch 2 (S2) is not operated.

The above exercise has the following four (4) possibilities.



Tasks:

Write a PLC program with the S5 Blocks PB10 and OB1.

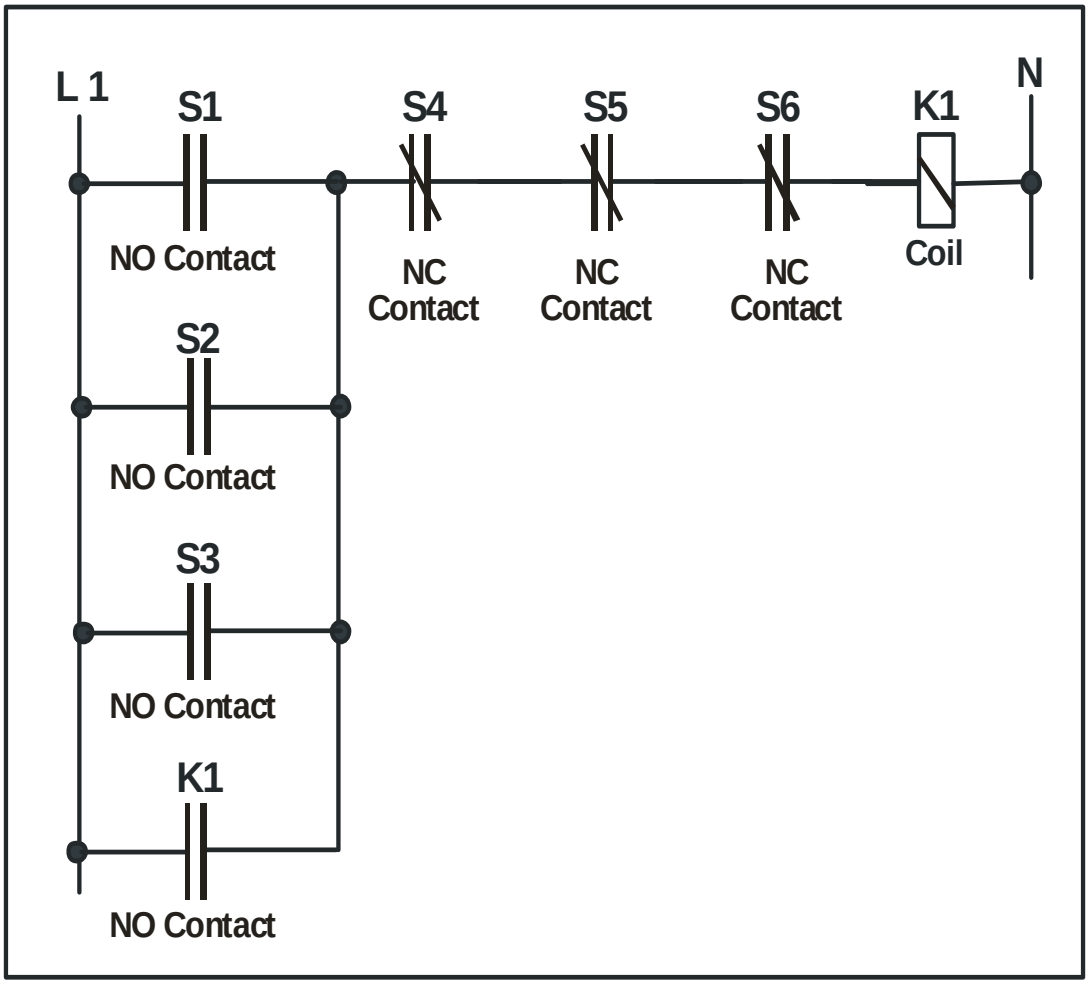
Transfer of the program into the S5 TEST PLC.

Test the PLC program.

Converting a relay logic into a PLC Program

Example: Motor ON / OFF with Locking

The simple relay circuit shown in the picture needs to be converted to a PLC Program.



Name	PLC Operand
S1 (NO) Push Button	I 0.0
S2 (NO) Push Button	I 0.1
S3 (NO) Push Button	I 0.2
K1 Relay (Motor On)	Q0.0

Name	PLC Operand
S4 (NC) Push Button	I 0.3
S4 (NC) Push Button	I 0.4
S4 (NC) Push Button	I 0.5

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

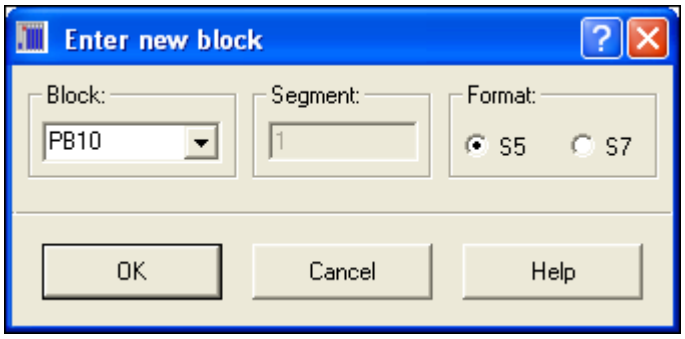
Test the PLC program.

Using the LAD Editor

Programming an Example using Ladder Diagram (LAD)

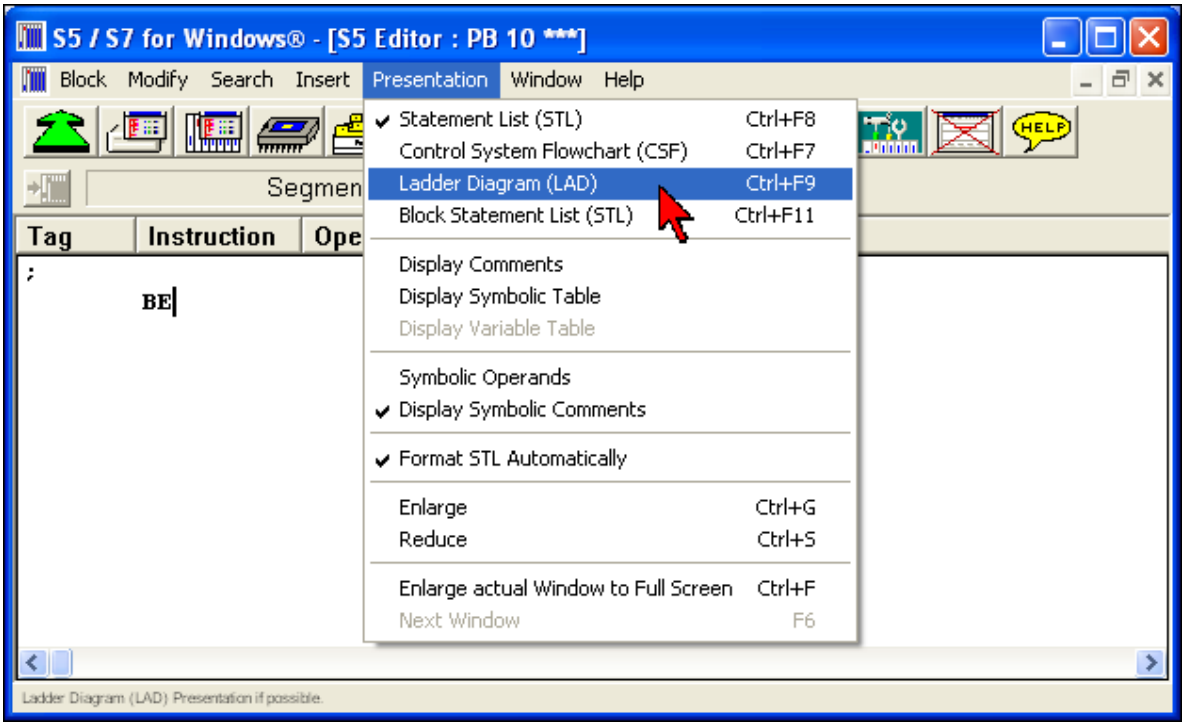
The programming of the example is explained using **Ladder Logic** (**LAD**).

Confirming the name in the " Enter new Block" dialog box the editor windows is opened and the new block is ready to be programmed.



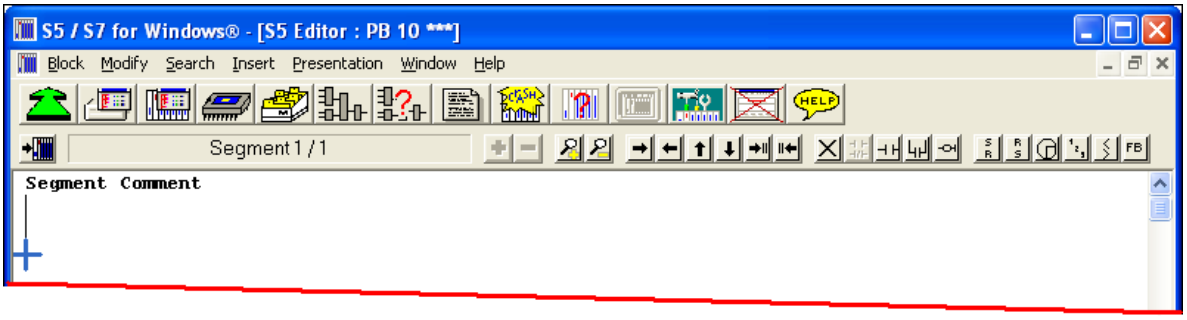
Selecting Ladder Diagram (LAD) Presentation

The Command **Ladder Diagram (LAD)** from the **Presentation Menu** selects the PLC logic presentation Ladder Diagram.



The workplace is ready for entering the program block PB10.

S5 Programming Software Editor Window, Ladder Diagram Presentation

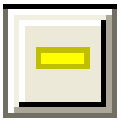


The first segment is opened.

The first line is reserved to enter a comment.

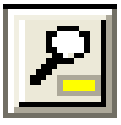
The tool bar II provides the icons for easy programming. Clicking an icon with the mouse calls the desired function.

Next Segment selection

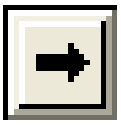


Previous Segment selection

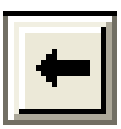
Enlarge the size of the logic shown in the workplace. The selected font must be a true type font to allow scaling.



Reduce the size of the logic shown in the workplace. The selected font must be a true type font to allow scaling.

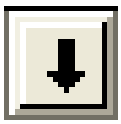


Draw a line to the right of the insertion mark. If a line is already to the right of the insertion mark the line is erased. A line replaces a contact to the right of the insertion mark.

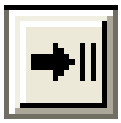


Draw a line to the left of the insertion mark. If a line is already to the left of the insertion mark the line is erased. A line replaces a contact to the right of the insertion mark.

Draw a line to upward from the insertion mark. If a line is already exists the line is erased.

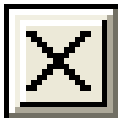


Draw a line to downward from the insertion mark. If a line is already exists the line is erased.



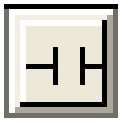
Insert a contact to the right of the insertion mark. If a contact already exists to the right of the insertion mark the contact is erased. A line to the right is replaced by a contact.

Insert a contact to the left of the insertion mark. If a contact already exists to the left of the insertion mark the contact is erased. A line to the left of the insertion mark is replaced by a contact.



Delete the contact to the right of the insertion mark. The delete button only works in a logical operational segment.

Change the selected (mark operand or insertion mark to the right of the contact) contact from normally open (NO) to normally closed (NC) or vice versa (NC to NO). The operand must be defined prior to the change command.

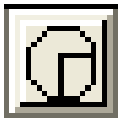


Insert a normally open (NO) contact to the right of the insertion mark. The insert button only works in a logical operational segment.

Insert a parallel branch with a normally open (NO) contact to the right (and down) of the insertion mark. The insert button only works in a logical operational segment.

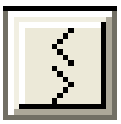
Insert a SR Flip Flop (latch) with a dominating reset input.

Insert a RS Flip Flop (latch) with a dominating set input.



This icon opens a dialog box to select timers.

This icon opens a dialog box to select counters.



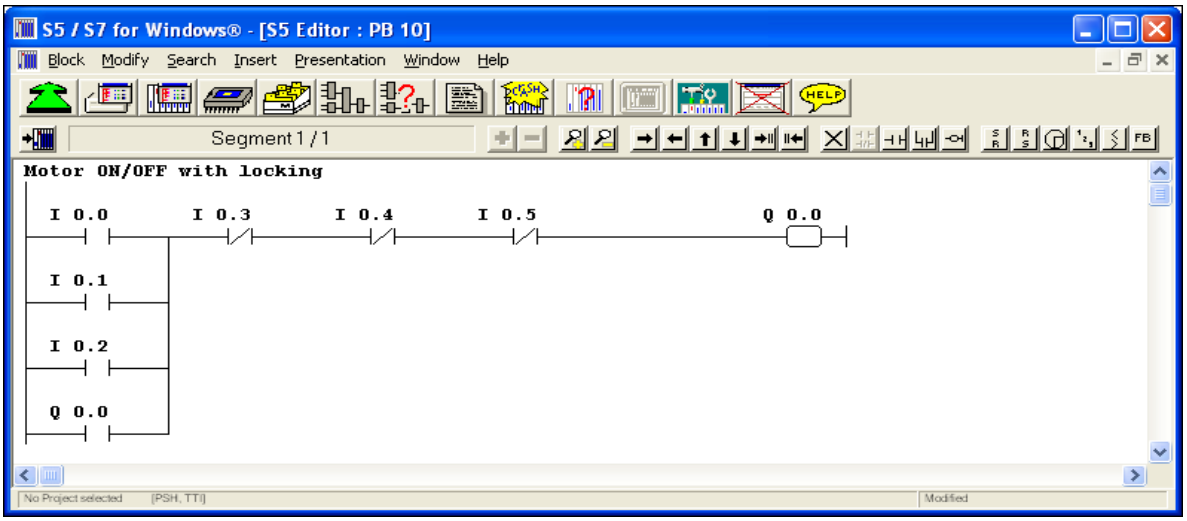
This icon opens a dialog box to select comparators.

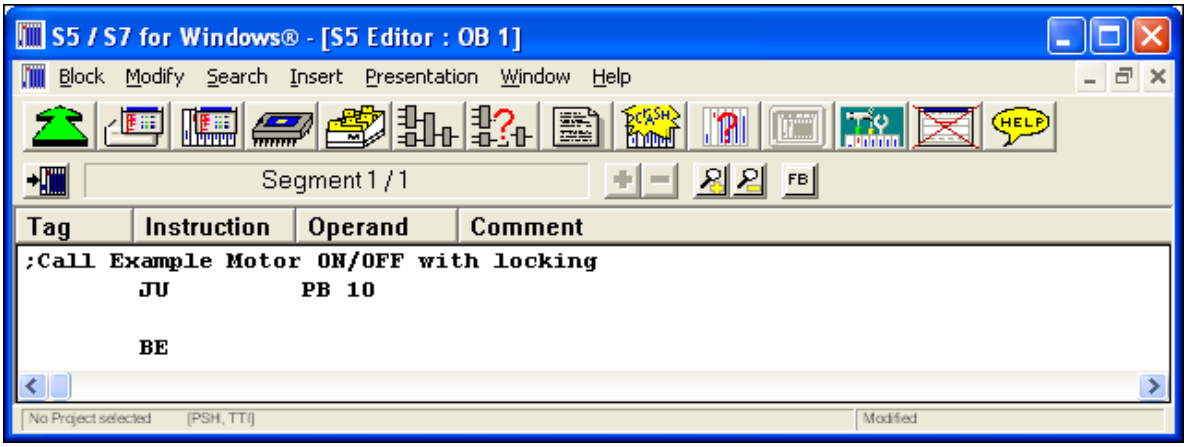
Insert a function block (FB) call. A function block call is only be permitted in a separate segment

Clicking the bottom of the vertical line the insertion mark, a blue cross appears.

Building a Segment using the Mouse. Click the icon to insert a contact to the right of the insertion mark. By using the icons explained above the logic is built.

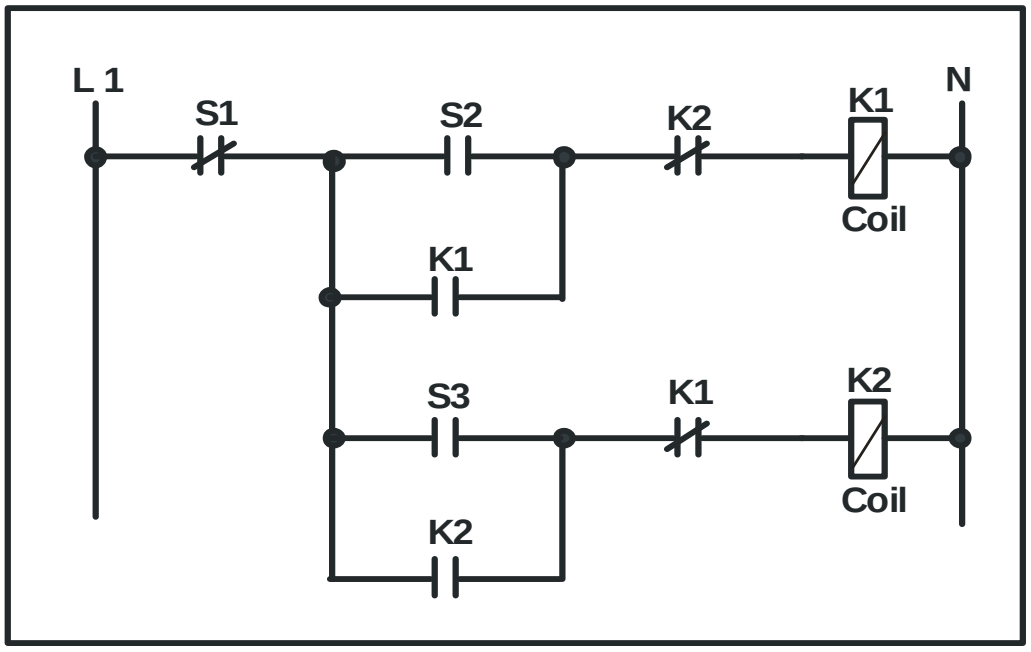
Program Block BB10





Practice Exercise 3–8; Motor right/left

The relay logic shown in the picture (motor right, motor left) needs to be converted to a PLC program.



Name	PLC Operand
S1 (NC) Motor OFF	I 0.0
S2 (NO) Motor right	I 0.1
S3 (NO) Motor left	I 0.2
K1 Relay (Motor right)	Q0.0
K2 Relay (Motor left)	Q0.0

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

3.2 Number Systems

In order to understand the definition of variables in the STEP® 5 programming language, the type of numbering systems used in the PLC technology must be known.

Decimal system

Normally we use the decimal system to indicate numbers.

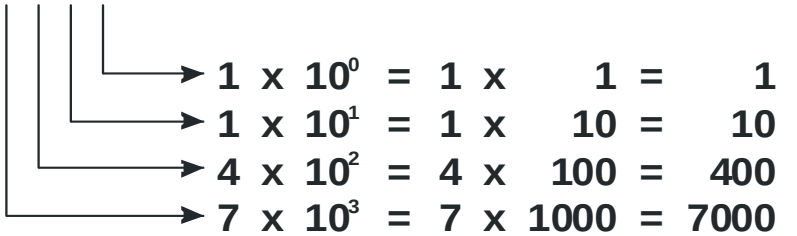
The decimal number system has the base number of ten (10). Each number in the decimal system is expressed as a multiple of a power of ten.

Figure: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

Base: 10

Value: Power of 10 (Base)

Example: 7 4 1 1



$1 \times 10^0 = 1 \times 1 = 1$

$1 \times 10^1 = 1 \times 10 = 10$

$4 \times 10^2 = 4 \times 100 = 400$

$7 \times 10^3 = 7 \times 1000 = 7000$

Sum : 7411

Note:

A number by the power of zero (0) is one (1) e.g. $10^0 = 1$; $2^0 = 1$; $16^0 = 1$;

A number by the power of one (1) is the number itself. e.g. $10^1 = 10$; $2^1 = 2$; $16^1 = 16$

Binary Numbers

Binary Numbers can only have two (2) values:

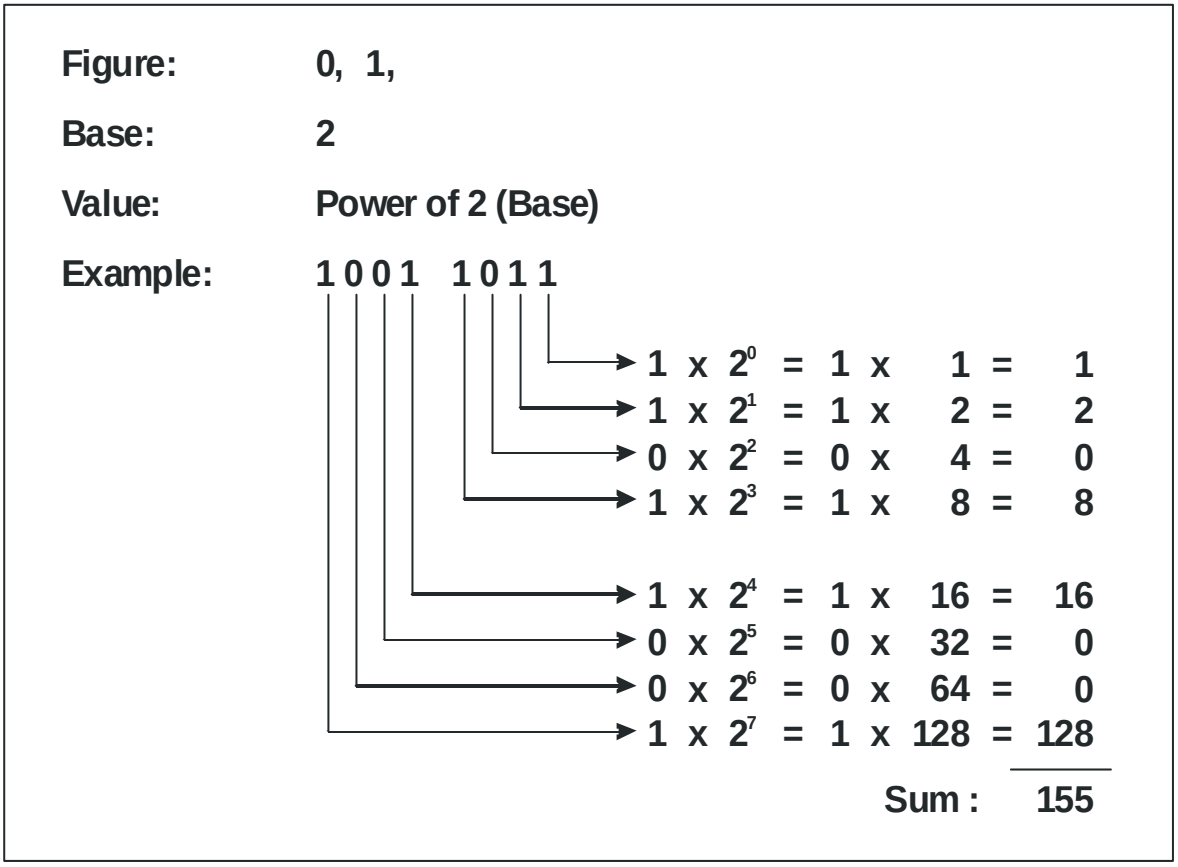
FALSE (Zero (0) or low)

TRUE (One (1) or High)

Therefore the binary number system is used in the digital system (on/off).

The binary number system has the base number two (2).

Each number in the binary number system is expressed as multiple of a power-of-two number.



Eight (8) digits are needed, in order to represent the decimal value of 155.

The maximum value, which can be represented with eight (8) digits, is decimally 255.

A "digit" in the binary number system is called "bit". It is common that eight (8) bits are called a **Byte**.

Hexadecimal Numbers

The example of the binary numbers shows that the number of digits required to express a large number will increase drastically.

In order to take advantage of the binary numbering system for digital systems (on/off) and to reduce the number of digits required to express a large number, four (4) binary digits are combined to create a hexadecimal digit.

The base number of the hexadecimal system is sixteen (16).

Each number in the hexadecimal number system is expressed as multiple of a power of "16".

Figure: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), F (15)

Base: 16

Value: Power of 16 (Base)

Example: 4 A 7 F

F	x 16^0 =	15	x 1 =	15
7	x 16^1 =	7	x 16 =	112
A	x 16^2 =	10	x 256 =	2 560
4	x 16^3 =	4	x 4 096 =	16 384
			Sum :	19 071

The link between binary numbers and hexadecimal numbers

Power of 2 (Base 2)	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary Number	0	1	0	1	1	0	1	1	1	1	1	0	0	1	1	1
Power of 16 (Base 16)	16^3				16^2				16^1				16^0			
Hexadecimal Number	5				B				E				7			
Decimal Number	23 527															

Hexadecimal Numbers

Four (4) bits are required to represent a single hexadecimal number.

Hexadecimal - Numbers				
	8	4	2	1
Decimal				
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A (10)	1	0	1	0
B (11)	1	0	1	1
C (12)	1	1	0	0
D (13)	1	1	0	1
E (14)	1	1	1	0
F (15)	1	1	1	1

BCD numbers

The BCD numbering system is based on the hexadecimal number system with the agreement that only the numbers, which are present in the decimal number system, are used.

Due to this rule the decimal number system and the BCD number system have a base of ten (10).

The BCD number system as well as the hexadecimal numbering system uses four (4) bits for the representation of each BCD digit.

Each number in the BCD number system is expressed as multiple of a power of "10". Only the numbers 0 to 9 are used.

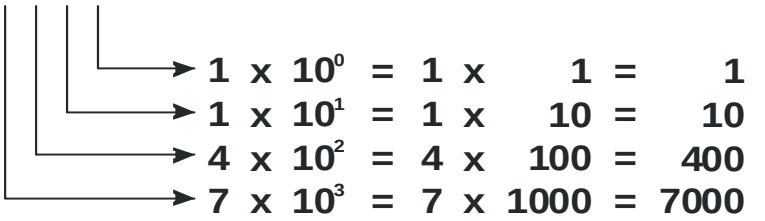
The advantage of the BCD numbering system is that the represented numbers are easier to read. However the disadvantage of the BCD numbering system is that it requires substantial conversion and memory power.

Figure: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

Base: 10

Value: Power of 10 (Base 10)

Example: 7 4 1 1



$1 \times 10^0 = 1 \times 1 = 1$

$1 \times 10^1 = 1 \times 10 = 10$

$4 \times 10^2 = 4 \times 100 = 400$

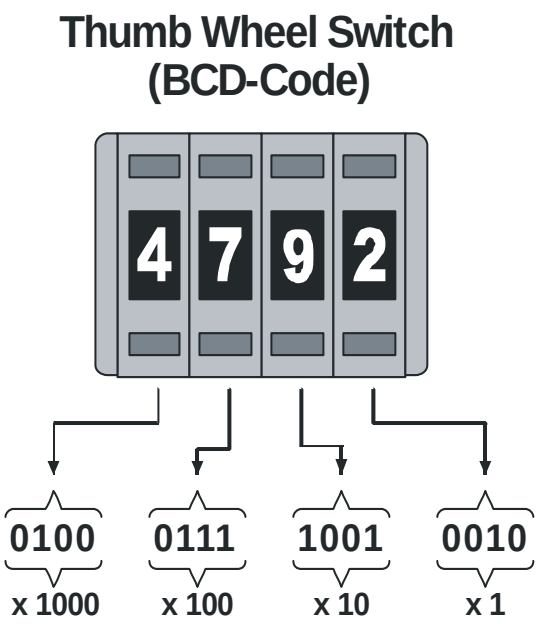
$7 \times 10^3 = 7 \times 1000 = 7000$

Sum : 7411

The link between binary, BCD, and hexadecimal numbers

Power of 2 (Base 2)	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary Number	0	1	0	1	1	0	0	1	0	1	1	0	0	1	1	1
Power of 10 (Base 10)	10^3				10^2				10^1				10^0			
BCD Number	5				9				6				7			
Decimal Number	5 967															

BCD Example: Thumb wheel switch



BCD - Numbers

Decimal	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

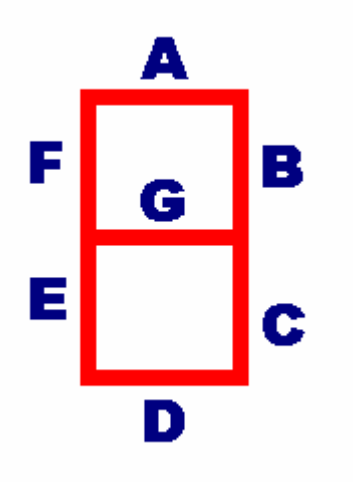
Practice Exercise 3–9; Seven Segment Display

The four switches S0, S1, S2, and S3 will be used to control a seven-segment display.

A Program Block (PB10) should be programmed to display the number.

The switches have the following values:

Switch	Value	PLC Operand
S0	2^0	I0.0
S1	2^1	I0.1
S2	2^2	I0.2
S3	2^3	I0.3



Write a PLC program with the S5 Blocks PB10 and OB1.

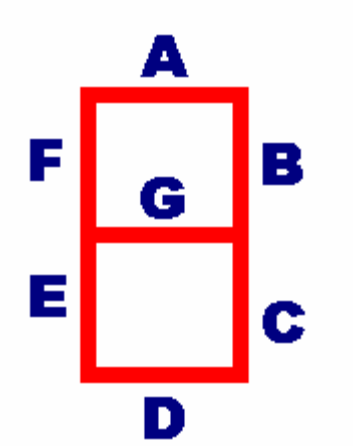
Transfer of the program into the S5 TEST PLC.

Test the PLC program.

Switch Decoding

Number	I 0.0 one (1)	I 0.1 two (2)	I 0.2 four (4)	I 0.3 eight (8)	Flag	Display FW 2
0	0	0	0	0	F3.0	1
1	1	0	0	0	F3.1	2
2	0	1	0	0	F3.2	4
3	1	1	0	0	F3.3	8
4					F3.4	16
5					F3.5	32
6					F3.6	64
7					F3.7	128
8					F2.0	256
9					F2.1	512
A					F2.2	1024
B					F2.3	2048
C					F2.4	4096
D					F2.5	8192
E					F2.6	16384
F					F2.7	-32768

7 Segment Decoding



Number	Flag	Segment A	Segment B	Segment C	Segment D	Segment E	Segment F	Segment G
		Q0.0	Q0.1	Q0.2	Q0.3	Q0.4	Q0.5	Q0.6
0	F3.0	X	X	X	X	X	X	
1	F3.1		X	X				
2	F3.2	X	X		X	X		X
3	F3.3	X	X	X	X			X
4	F3.4		X	X			X	X
5	F3.5	X		X	X		X	X
6	F3.6	X		X	X	X	X	X
7	F3.7	X	X	X				
8	F2.0	X	X	X	X	X	X	X
9	F2.1	X	X	X			X	X
A	F2.2	X	X	X		X	X	X
B	F2.3			X	X	X	X	X
C	F2.4	X			X	X	X	
D	F2.5		X	X	X	X		X
E	F2.6	X			X	X	X	X
F	F2.7	X				X	X	X

X **Output (Q..) must be "ON"**

3.3 Setting / Resetting Bit Addresses

The value of **"1" (true)** or the value of **"0" (false)** can be assigned to an Address (memory location, operand) when the RLO has a value of "1" (true) by using the instructions **"S"** (Set) or **"R"** (Reset).

S – Set instruction

Format: S <Bit>

Address	Data type	Memory area
<Bit>	BOOL	I, Q, M, L, D

S (set bit) places a "1" (true) in the addressed bit if RLO = 1. The "S" instruction is an RLO delimiting Instruction.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	x	-	0

Example

STL Program	Relay Logic
A I 0.0 S Q 0.4 A I 0.1 R Q 0.4 Signal State Diagrams I 0.0 I 0.1 Q 0.4	L1 I 0.0 NO Contact Q 0.4 NO Contact I 0.1 NO Contact NC Contact Q 0.4 Coil Coil N

R – Reset instruction

Format: R <Bit>

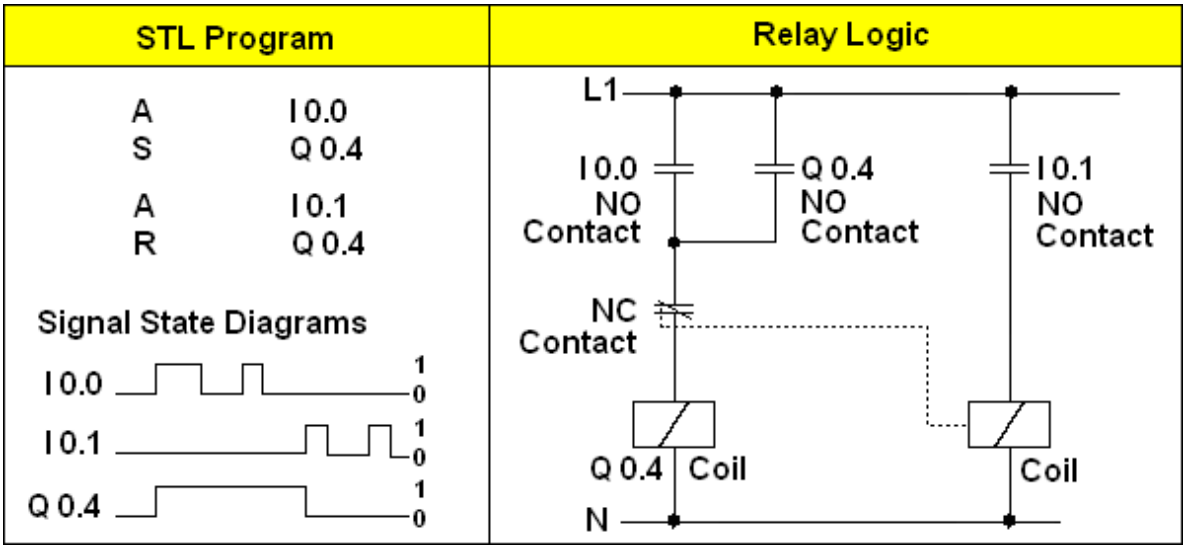
Address	Data type	Memory area
<Bit>	BOOL	I, Q, M, L, D

R (reset bit) places a "0" in the addressed bit if RLO = 1. The “R” instruction is an RLO delimiting Instruction.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	x	-	0

Example



The output Q 0.4 is set to “1” as soon as the input I 0.0 has a status of “1” (actually the RLO must be “1”). If the input I 0.0 goes back to “0” the output Q 0.4 remains set.

The output Q 0.4 is reset to “0” as soon as the input I 0.1 has a status of “1” (actually the RLO must be “1”).

In the example it is possible that both inputs (I 0.0 and I 0.1) are “1” (true). In this case the output Q 0.4 is set to “1” and immediately reset to “0”. The output Q 0.4 therefore remains reset (false) because the “R” (reset instruction) follows the “S” (set instruction).

Note:

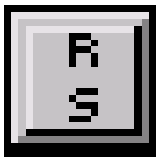
Set Dominant

If a latch (RS Flip Flop) should remain set (“1” – true) when both inputs (set and reset) of the latch are “1” (true), the “R” (reset instruction) must be programmed prior the “S” (set instruction).

Reset Dominant

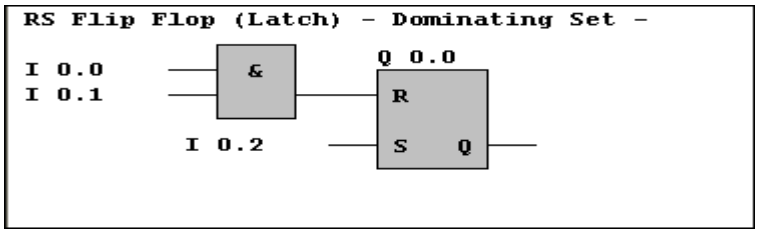
If a latch (SR Flip Flop) should be reset (“0” – false) when both inputs (set and reset) of the latch are “1” (true), the “S” (set instruction) must be programmed prior the “R” (set instruction).

RS Flip Flop

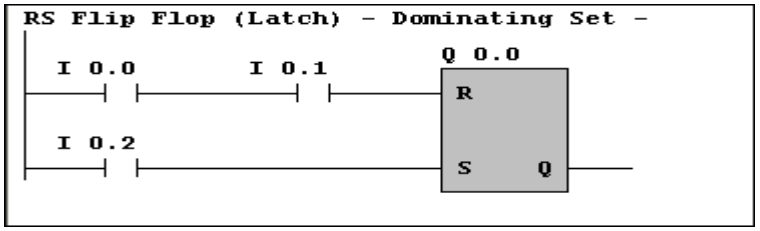


With the button “RS” a RS Flip Flop with a dominating set input is inserted.

CSF Presentation



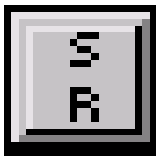
LAD Presentation



STL Presentation

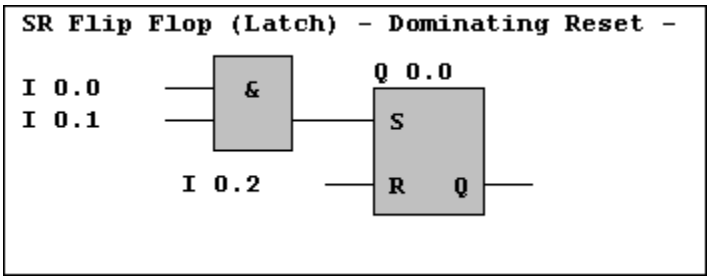
Tag	Instruction	Operand	Comment
; RS Flip Flop (Latch) - Dominating Set -			
	A	I 0.0	
	A	I 0.1	
	R	Q 0.0	
	A	I 0.2	
	S	Q 0.0	
	NOP	0	

SR Flip Flop

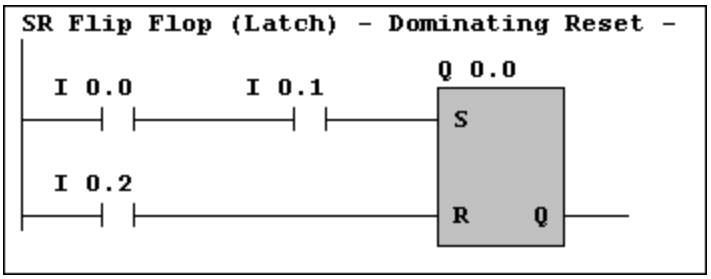


With the button “SR” a SR Flip Flop with a dominating reset input is inserted.

CSF Presentation



LAD Presentation



STL Presentation

Tag	Instruction	Operand	Comment
; SR Flip Flop (Latch) - Dominating Reset -			
	A	I 0.0	
	A	I 0.1	
	S	Q 0.0	
	A	I 0.2	
	R	Q 0.0	
	NOP	0	

The instruction “NOP 0” is only required to convert the STL presentation into LAD or CSF presentation.

Practice Exercise 3–10; Latch

A light needs to be switched on and off from three (3) different locations.

If an ON switch and an OFF switch are operated at the same time the light should stay on.

Device	PLC Operand
S1: Switch ON (Location 1)	I 0.0
S2: Switch OFF (Location 1)	I 0.1
S3: Switch ON (Location 2)	I 0.2
S4: Switch OFF (Location 2)	I 0.3
S5: Switch ON (Location 3)	I 0.4
S6: Switch OFF (Location 3)	I 0.5
H1: Light	Q 0.0

Write a PLC program with the S5 Blocks PB10 and OB1.

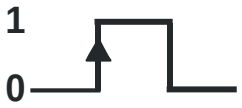
Transfer of the program into the S5 TEST PLC.

Test the PLC program.

3.4 Edge Detection

STEP® 5 does not provide Instructions to detect the rising and falling edge (flange detection) of a signal (RLO).

Positive Flange (positive edge)



Negative Flange (negative edge)

The “Edge Detection” can be programmed using bit operations. The instructions require “Flange Memory”. This “Flange Memory” must be a bit address that fulfills the following requirements:

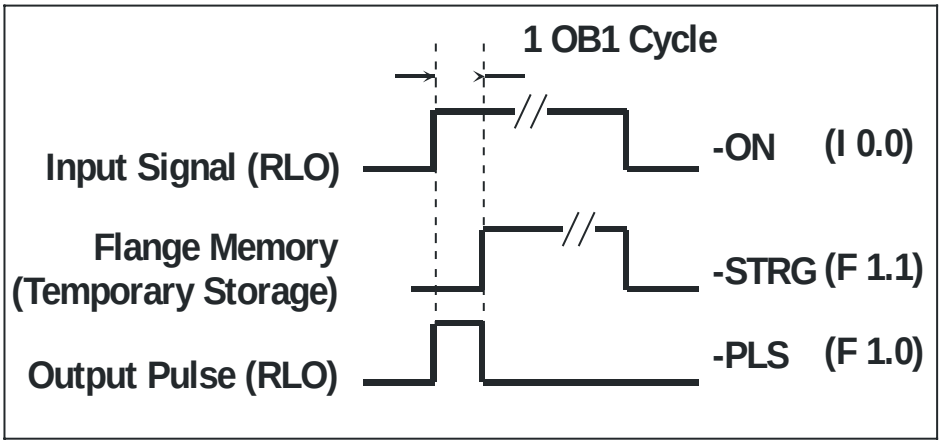
The “Flange Memory” may not be modified at any other location within the PLC program.

The status of the “Flange Memory” must be available in the next OB1 cycle.

The following “Flange Memory” operands (Bit) fulfill these requirements:

Flag	for instance:	F10.0, F15.1 etc.
Data Bit in a Data Block	for instance:	D3.1, D2.14

Positive Edge Detection



If a positive edge of the Input Signal “-ON” (I 0.0) is detected, the Output Pulse “-PLS” F1.0) is “1” (true) for one (1) OB 1 scan cycle. The previous RLO state is stored in “Flange Memory”.

Positive Edge Detection (STL – absolute operands)

```
Tag        Instruction   Operand     Comment
;Positive Flange (Edge) detection

           A             I  0.0      Signal to detect the Flange
           AN            F  1.1      Edge Flag
           =             F  1.0      Edge Pulse

           A             I  0.0      Signal to detect the Flange
           =             F  1.1      Edge Flag
```

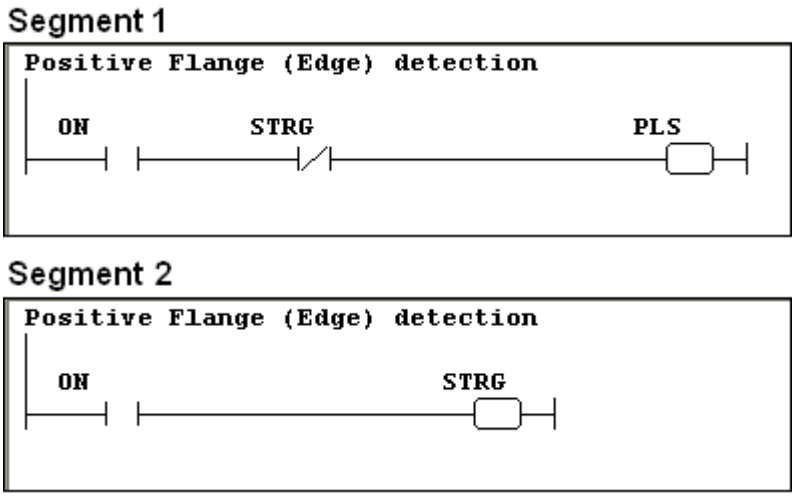
During each program scan cycle, the signal state of the Input Signal (RLO) bit is compared with the previous cycle to see if there has been a state change.

The previous RLO state must be stored in the “Flange Memory” (Bit) to make the comparison. If there is a difference between current and previous "0" state (detection of rising edge), the Output Pulse (RLO) bit will be "1".

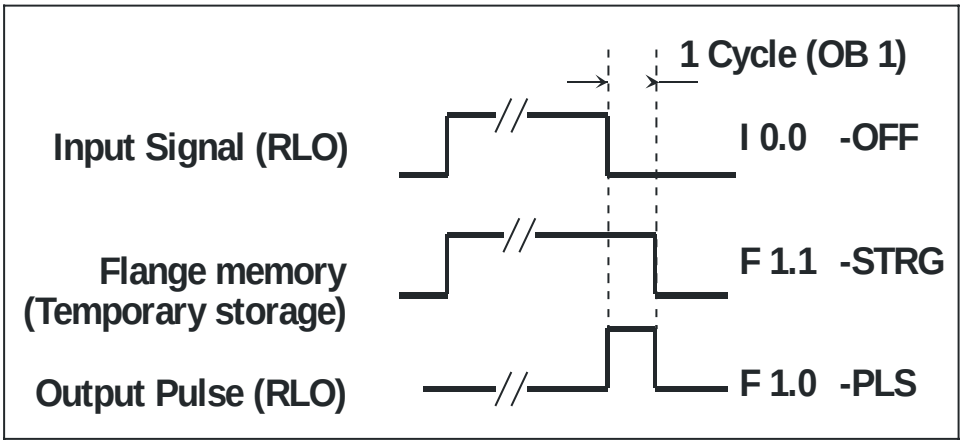
Positive Edge Detection (STL – symbolic operands)

```
Tag        Instruction   Operand     Comment
;Positive Flange (Edge) detection

           A             -ON         Signal to detect the Flange
           AN            -STRG       Edge Flag
           =             -PLS        Edge Pulse

           A             -ON         Signal to detect the Flange
           =             -STRG       Edge Flag
```

Positive Edge Detection (LAD – two segments)



Negative Edge Detection



If a negative edge of the Input Signal “-OFF” (I 0.0) is detected, the Output Pulse “-PLS” F1.0) is “1” (true) for one (1) OB 1 scan cycle. The previous RLO state is stored in “Flange Memory”.

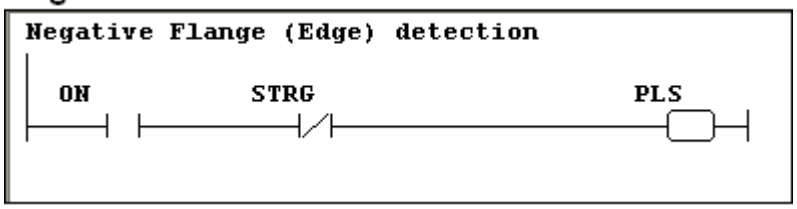
Negative Edge Detection (STL – absolute operands)

```
;Negative Flange (Edge) detection

      A          I  1.0     Signal to detect the Flange
      AN         F  1.1     Edge Flag
      =          F  2.0     Edge Pulse

      A          I  1.0     Signal to detect the Flange
      =          F  2.1     Flag Memory
```

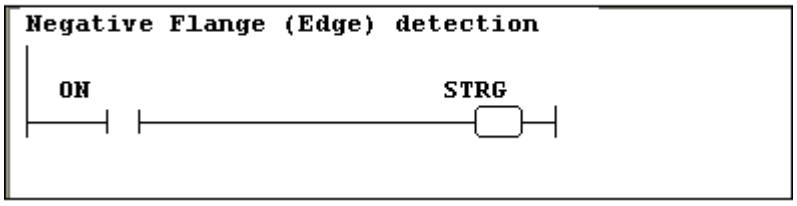
Negative Edge Detection (STL – symbolic operands)

```
;Negative Flange (Edge) detection

        A           -OFF        Signal to detect the Flange
        AN          -STRG       Edge Flag
        =           -PLS_N      Edge Pulse

        A           -OFF        Signal to detect the Flange
        =           -FLGM       Flag Memory
```

Segment 1



Segment 2



During each program scan cycle, the signal state of the Input Signal (RLO) bit is compared with the previous cycle to see if there has been a state change.

The previous RLO state must be stored in the “Flange Memory” (Bit) to make the comparison. If there is a difference between current and previous "0" state (detection of rising edge), the Output Pulse (RLO) bit will be "1".

Practice Exercise 3–11; Motor ON/OFF, Edge Detection with Latch

Two (2) push buttons are used to switch a Motor (Latch) ON and OFF.

To be independent of the activation of the push buttons, edge detection should be used.

For safety reasons the OFF push button should have a NO contact (activating the OFF push button puts a “0” at the input).
If a “wire brake” occurs at the OFF push button the motor should be switched off and it should not be possible to start the motor with the ON push button.

Device	PLC Operand
Pushbutton ON	I 0.0
Pushbutton OFF	I 0.1
Motor Relay (Latch)	Q 0.0
Flange Memory (ON – , Pos.)	F 10.1
Flange Memory (OFF – ,Neg.)	F 10.3
Pulse Positive (not required)	F 10.0
Pulse Negative (not required	F 10.2

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

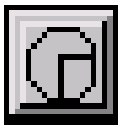
4 Timing Functions (Timer) and Counters

The Timing functions and Counters are similarly constructed software functions, using 16-bit word operands.

4.1 Timing Functions (Timer)

Timing functions are used to implement waiting periods and monitoring times in the PLC program

The timing functions are constructed using 16-bit word operands.



With the command ***Timer…*** a dialog box is opened to select one Timer from a choice of five timer functions.

Timer signals overview

SP, SE, SR, SS, SF	Start timer			
TV	**Time Constant KT** The Time Constant occupies a 16 bit word. The Time constant is entered as a 3 digit BCD number, the Time value followed by a decimal point and a multiplication factor. min. KT 1.0 = 10ms max. KT 999.3 = 2h, 46m, 25s			
	The following multiplication factors are available:			
	Time Base	Accuracy	Example	Time
	0 = 0.01s	10ms	KT 500.0	5 Seconds
	1 = 0.1s	100ms	KT 50.1	5 Seconds
	2 = 1s	1s	KT 5.2	5 Seconds
	3 = 10s	10s	KT 100.3	1000 Seconds
R	Reset			
BI	Current counter value (Binary)			
DE	Current counter value (BCD)			
Q	Output			

Area in Memory

Timers have an area reserved for them in the memory of your CPU. This memory area reserves one 16-bit word for each timer address.

The following functions have access to the timer memory area:

Timer instructions

Updating of timer words by means of clock timing. This function of your CPU in the RUN mode decrements a given time value by one unit at the interval designated by the time base until the time value is equal to zero.

Time Value

Bits 0 through 9 of the timer word contain the time value in binary code. The time value specifies a number of units. Time updating, decrements the time value by one unit at an interval designated by the time base. Decrementing continues until the time value is equal to zero. The time value is loaded into the accumulator 1 in the following format.

KT xyz.t

- Where **t** = the time base (that is, the time interval or resolution)
- Where **xyz** = the time value in binary coded decimal format

The maximum time value that you can enter is 9,990 seconds, or 2H_46M_30S.

Time Base

Bits 12 and 13 of the timer word contain the time base in binary code. The time base defines the interval at which the time value is decremented by one unit. The smallest time base is 10 ms; the largest is 10 s.

Time Base	Binary Code for the Time Base
10 ms	00
100 ms	01
1 s	10
10 s	11

Values that exceed 2h46m30s are not accepted. A value whose resolution is too high for the range limits (for example, 2h10ms) is

truncated down to a valid resolution. The general format for S5TIME has limits to range and resolution as shown below:

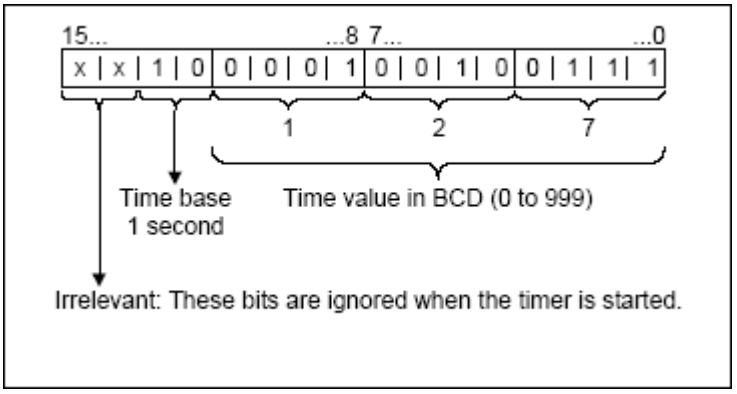
Resolution	Range	Time Constant
0.01 second	10MS to 9S_990MS	KT 1.0 – KT 999.0
0.1 second	100MS to 1M_39S_900MS	KT 1.1 – KT 999.1
1 second	1S to 16M_39S	KT 1.2 – KT 999.2
10 seconds	10S to 2H_46M_30S	KT 1.3 – KT 999.3

Bit Configuration in ACCU 1

When a timer is started, the contents of ACCU1 are used as the time value.

Bits 0 through 11 of the ACCU1-L hold the time value in binary coded decimal format (BCD format: each set of four bits contains the binary code for one decimal value). Bits 12 and 13 hold the time base in binary code.

The following figure shows the contents of ACCU1-L loaded with timer value 127 and a time base of 1 second:



Starting a Timer

A timer is started as soon as signal (RLO) at the start input (LAD, CSF) or at the start operation (STL) changes its state as indicated in the table below.

This change of the signal state of the RLO is compulsory for starting a timer

For starting a timer the time base from accumulator 1 is used. In accumulator 1 are the BCD values for time value and time basis.

For starting a timer different functions can be used:

Name	Timer function	Start with the change of the
SP	Pulse timer	Signal State from "0" to "1"
SE	Extended pulse timer	Signal State from "0" to "1"
SD	On-delay timer	Signal State from "0" to "1"
SS	Retentive on-delay timer	Signal State from "0" to "1"
SF	Off-delay timer	Signal State from "1" to "0"

Reset Timer (R)

Format: R <timer>

Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

R <timer> stops the current timing function and clears the timer value and the time base of the addressed timer word if the RLO transitions from 0 to 1.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

Tag	Instruction	Operand	Comment
; Example Timer Reset			
	A	I 0.0	; Check the signal state of input I 0.0
	R	T 1	; The timer is reset if the RLO changes from "0" to "1"
	BE		

Enable Timer – FR (Free)

Format: FR <timer>

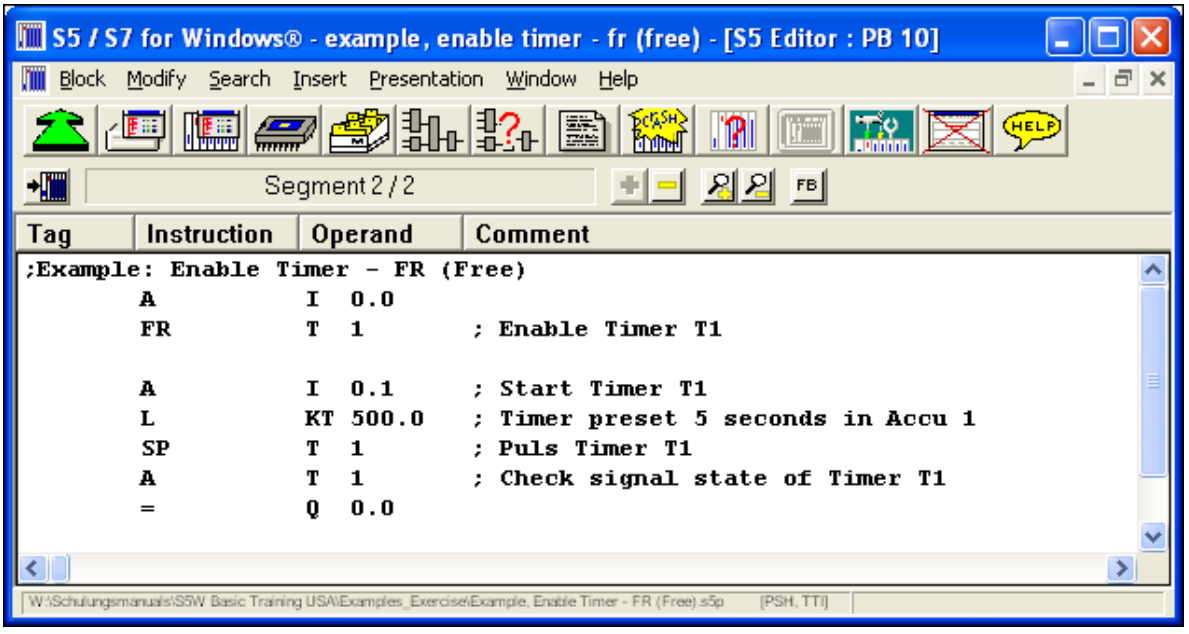
Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

When the RLO transitions from "0" to "1", FR <timer> clears the edge-detecting flag that is used for starting the addressed timer. A change in the RLO bit from 0 to 1 in front of an enable instruction (FR) enables a timer.

Timer enable is not required to start a timer, nor is it required for normal timer instruction. An enable is used only to re-trigger a running timer, that is, to restart a timer. The restarting is possible only when the start instruction continues to be processed with RLO = 1.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0



Note:

The instruction ***Enable Timer – FR*** (Free) is only available in STL – Presentation.

Pulse Timer (SP)

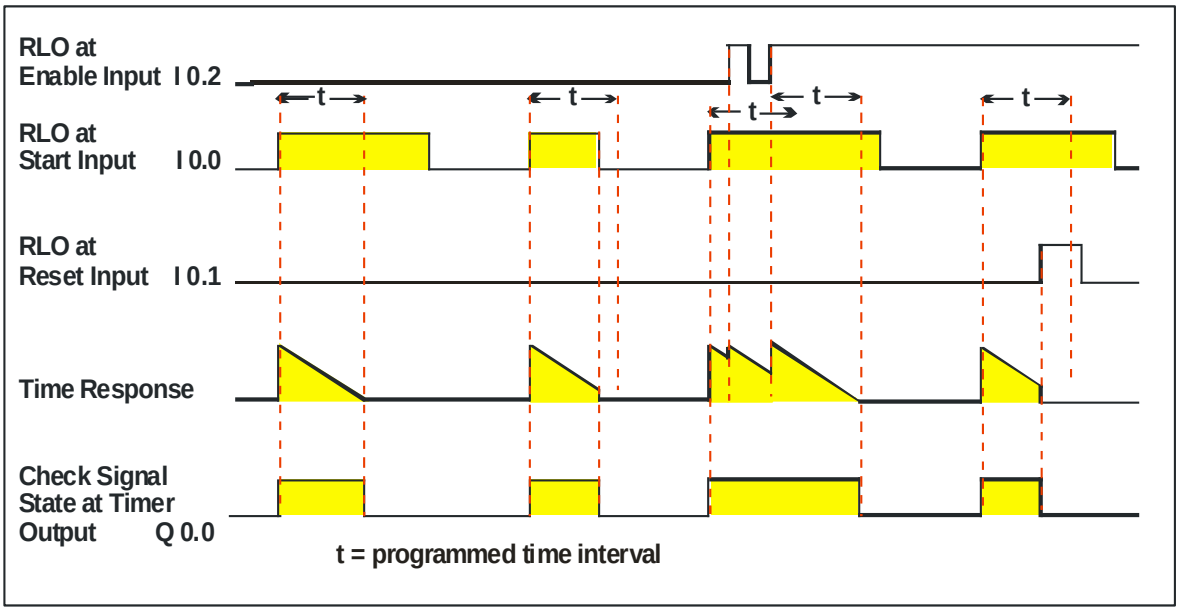
Format: SP <timer>

Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

SP <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time elapses as long as RLO = 1. The timer is stopped if, the RLO transitions to "0" before the programmed time interval has expired. This timer start command expects the time value and the time base to be stored as a BCD number in ACCU 1-L.

Status word

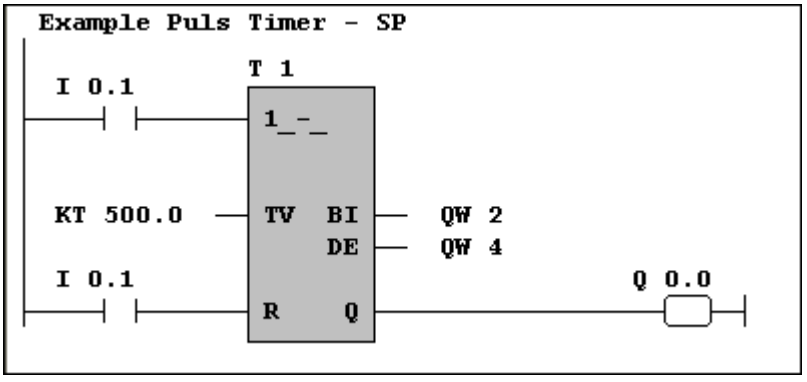
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

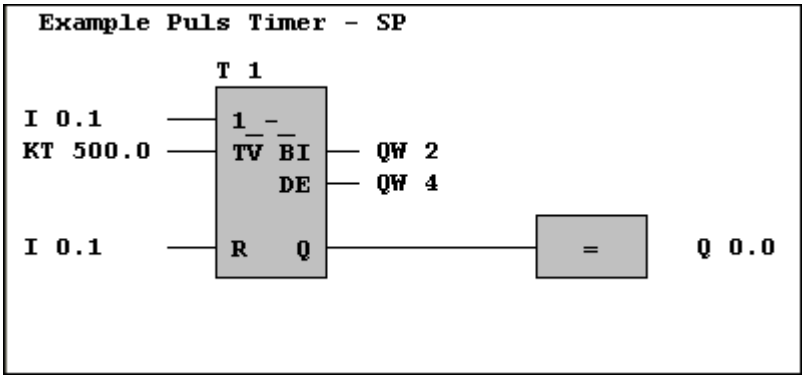


The maximum time that the output signal remains at 1 is the same as the programmed time value t. The output signal stays at 1 for a shorter period if the input signal changes to 0.

Pulse Timer (SP) (continued)

Tag	Instruction	Operand	Comment
; Example Puls Timer - SP			
	A	I 0.0	
	FR	T 1	; Enable Timer T1
	A	I 0.1	; Start Timer T1
	L	KT 500.0	; Timer preset 5 seconds in Accu 1
	SP	T 1	; Puls Timer T1
	A	I 0.1	; Reset signal
	R	T 1	; Reset Timer T1
	L	T 1	; Load current Timer value (T1) into ACCU1 in binary format
	T	QW 2	; Transfer to Output word
	LC	T 1	; Load current Timer value (T1) into ACCU1 in BCD format
	T	QW 4	; Transfer to Output word
	A	T 1	; Check signal state of Timer T1
	=	Q 0.0	





Extended Pulse Timer (SE)

Format: SE <timer>

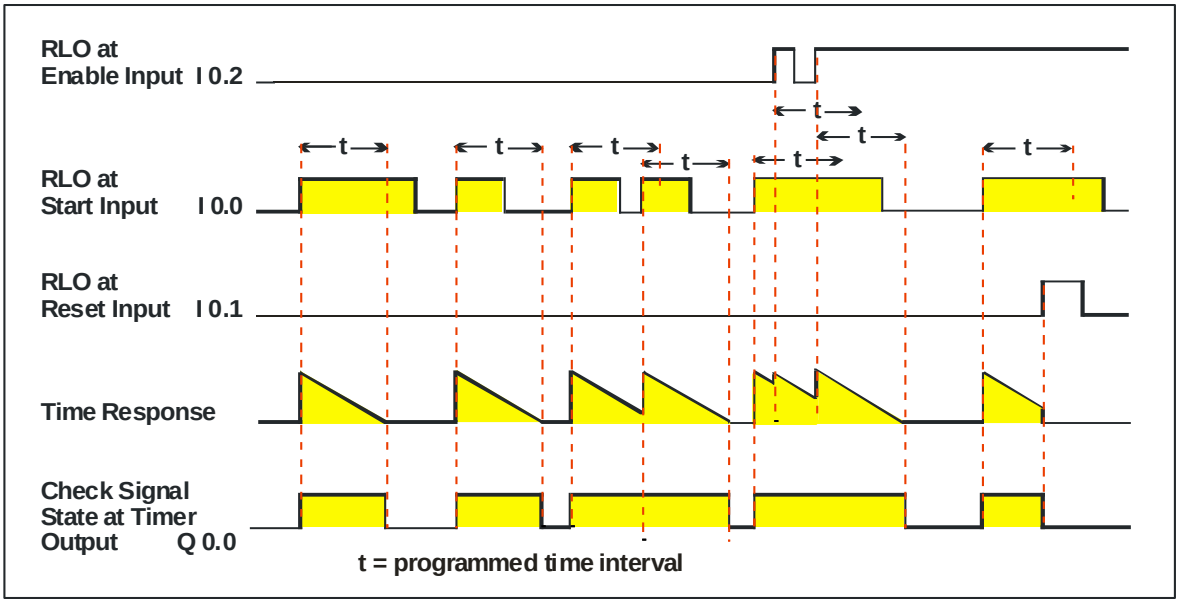
Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

SE <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time interval elapses, even if the RLO transitions to "0" in the meantime.

The programmed time interval is started again if, the RLO transitions from "0" to "1" before the programmed time has expired. This timer start command expects the time value and the time base to be stored as a BCD number in ACCU 1-L.

Status word

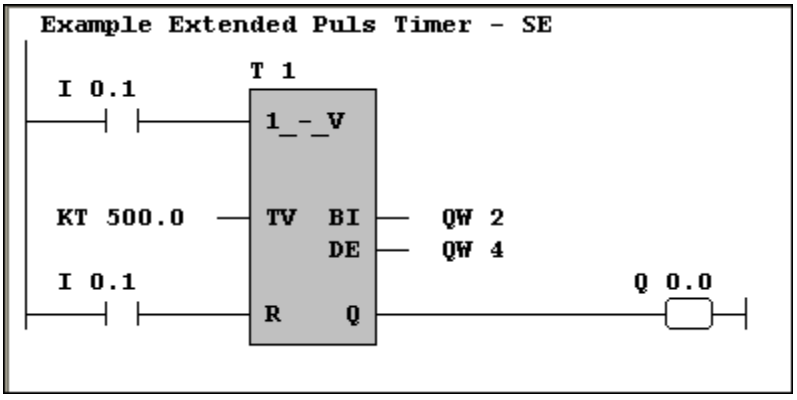
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

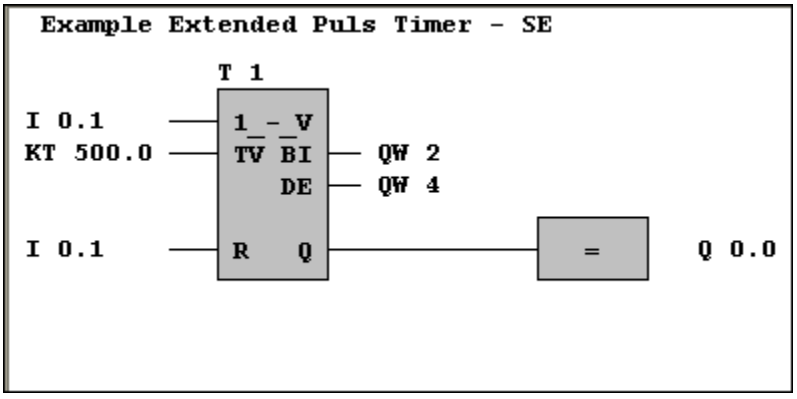


The output signal remains at 1 for the programmed length of time, regardless of how long the input signal stays at 1.

Extended Pulse Timer (SE) (continued)

Tag	Instruction	Operand	Comment
; Example Extended Puls Timer - SE			
	A	I 0.0	
	FR	T 1	; Enable Timer T1
	A	I 0.1	; Start Timer T1
	L	KT 500.0	; Timer preset 5 seconds in Accu 1
	SE	T 1	; Puls Timer T1
	A	I 0.1	; Reset signal
	R	T 1	; Reset Timer T1
	L	T 1	; Load current Timer value (T1) into ACCU1 in binary format
	T	QW 2	; Transfer to Output word
	LC	T 1	; Load current Timer value (T1) into ACCU1 in BCD format
	T	QW 4	; Transfer to Output word
	A	T 1	; Check signal state of Timer T1
	=	Q 0.0	





On-Delay Timer (SD)

Format: SD <timer>

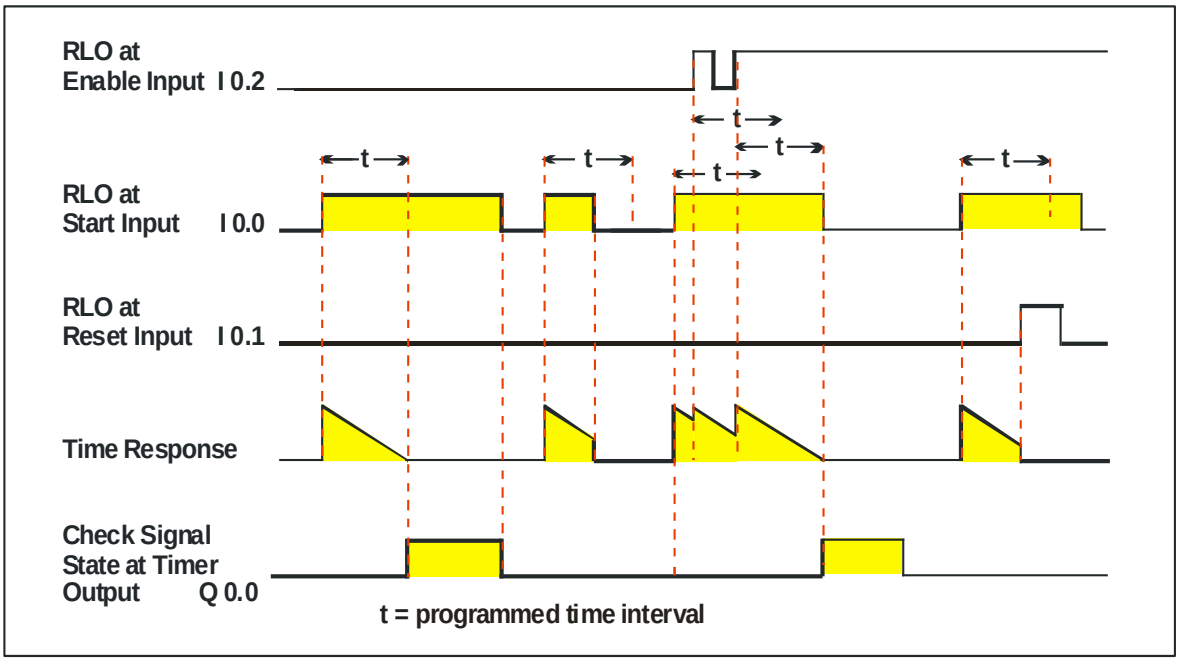
Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

SD <timer> starts the addressed timer when the RLO transitions from "0" to "1". The programmed time interval elapses as long as RLO = 1. The time is stopped if, the RLO transitions to "0" before the programmed time interval has expired.

This timer start instruction expects the time value and the time base to be stored as a BCD number in ACCU 1-L.

Status word

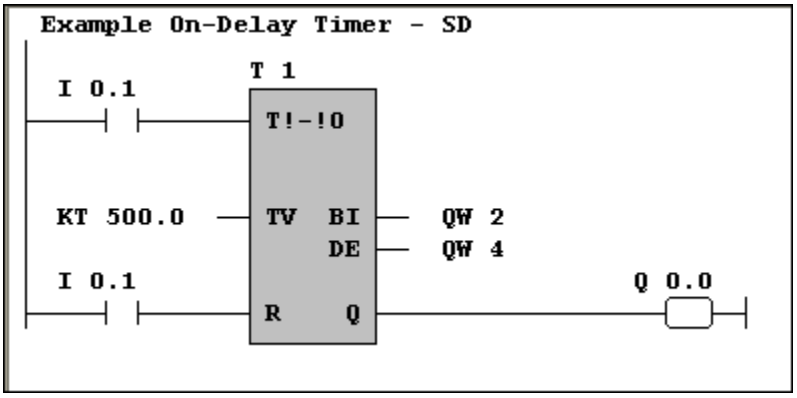
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

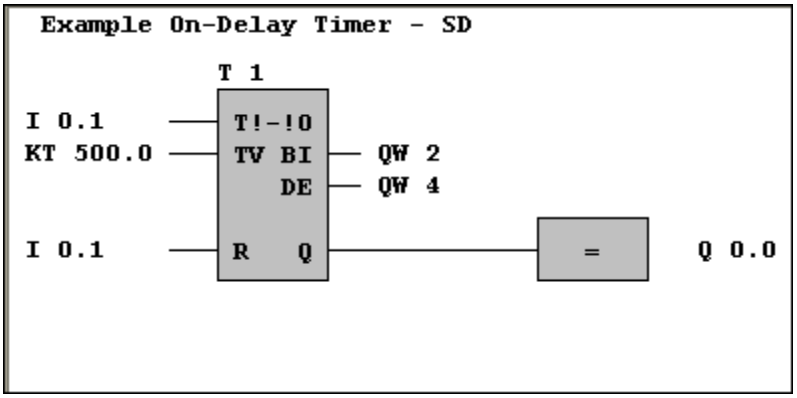


The output signal changes to 1 only when the programmed time has elapsed and the input signal is still 1.

On-Delay Timer (SD) (continued)

Tag	Instruction	Operand	Comment
; Example On-Delay Timer - SD			
	A	I 0.0	
	FR	T 1	; Enable Timer T1
	A	I 0.1	; Start Timer T1
	L	KT 500.0	; Timer preset 5 seconds in Accu 1
	SD	T 1	; Puls Timer T1
	A	I 0.1	; Reset signal
	R	T 1	; Reset Timer T1
	L	T 1	; Load current Timer value (T1) into ACCU1 in binary format
	T	QW 2	; Transfer to Output word
	LC	T 1	; Load current Timer value (T1) into ACCU1 in BCD format
	T	QW 4	; Transfer to Output word
	A	T 1	; Check signal state of Timer T1
	=	Q 0.0	





Retentive On-Delay Timer (SS)

Format: SS <timer>

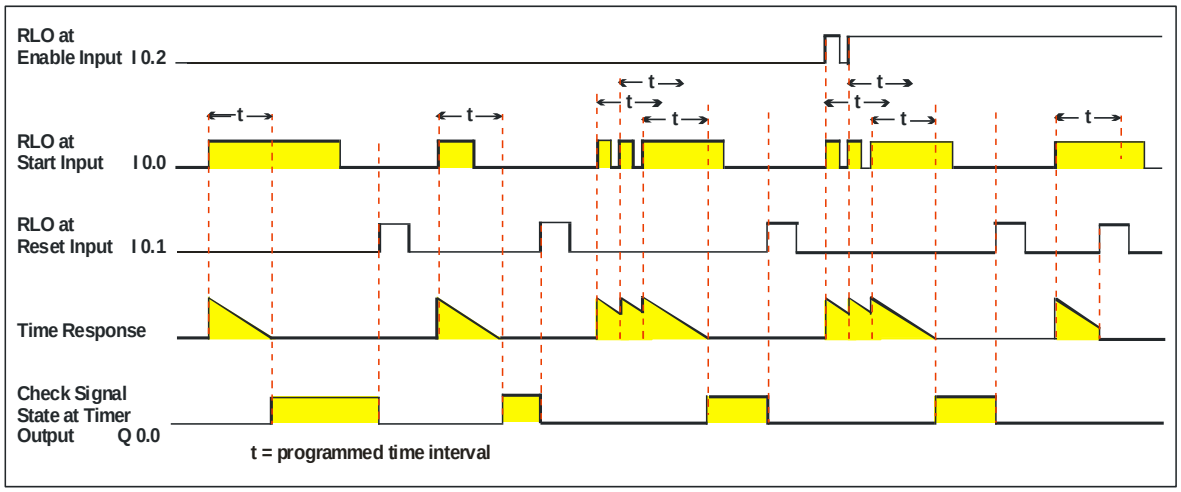
Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

SS <timer> (start timer as a retentive ON-delay timer) starts the addressed timer when the RLO transitions from "0" to "1". The full programmed time interval elapses, even if the RLO transitions to "0" in the meantime.

The programmed time interval is re-triggered (started again) if the RLO transitions from "0" to "1" before the programmed time has expired. This timer start command expects the time value and the time base to be stored as a BCD number in ACCU 1-L.

Status word

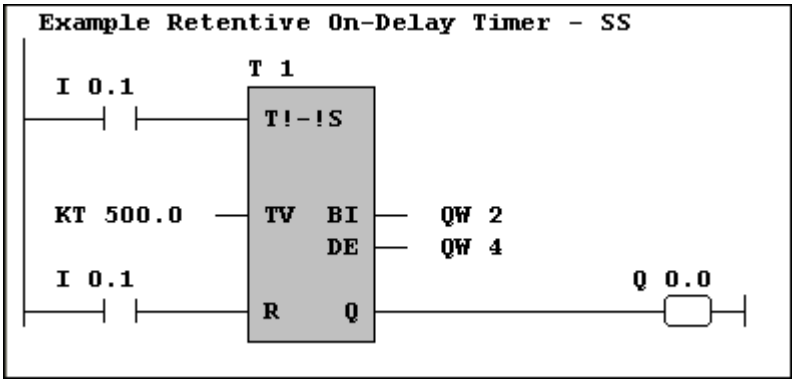
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

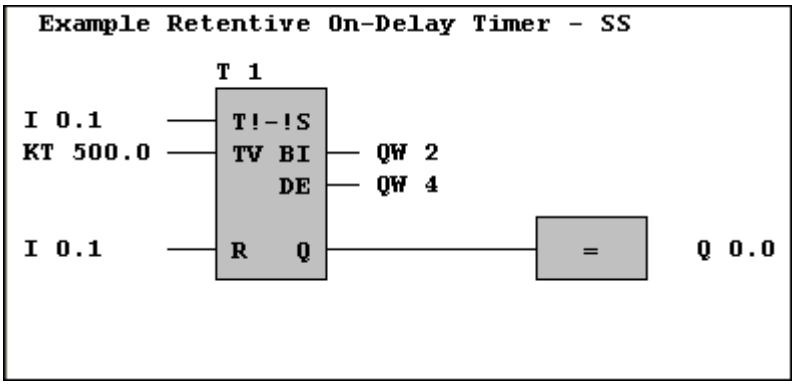


The output signal changes from 0 to 1 only when the programmed time has elapsed, regardless of how long the input signal stays at 1.

Retentive On-Delay Timer (SS) (continued)

Tag	Instruction	Operand	Comment
; Example Retentive On-Delay Timer - SS			
	A	I 0.0	
	FR	T 1	; Enable Timer T1
	A	I 0.1	; Start Timer T1
	L	KT 500.0	; Timer preset 5 seconds in Accu 1
	SS	T 1	; Puls Timer T1
	A	I 0.1	; Reset signal
	R	T 1	; Reset Timer T1
	L	T 1	; Load current Timer value (T1) into ACCU1 in binary format
	T	QW 2	; Transfer to Output word
	LC	T 1	; Load current Timer value (T1) into ACCU1 in BCD format
	T	QW 4	; Transfer to Output word
	A	T 1	; Check signal state of Timer T1
	=	Q 0.0	





Off-Delay Timer (SF)

Format: SF <timer>

Address	Data type	Memory area	Description
<timer>	TIMER	T	Timer number, range depends on CPU

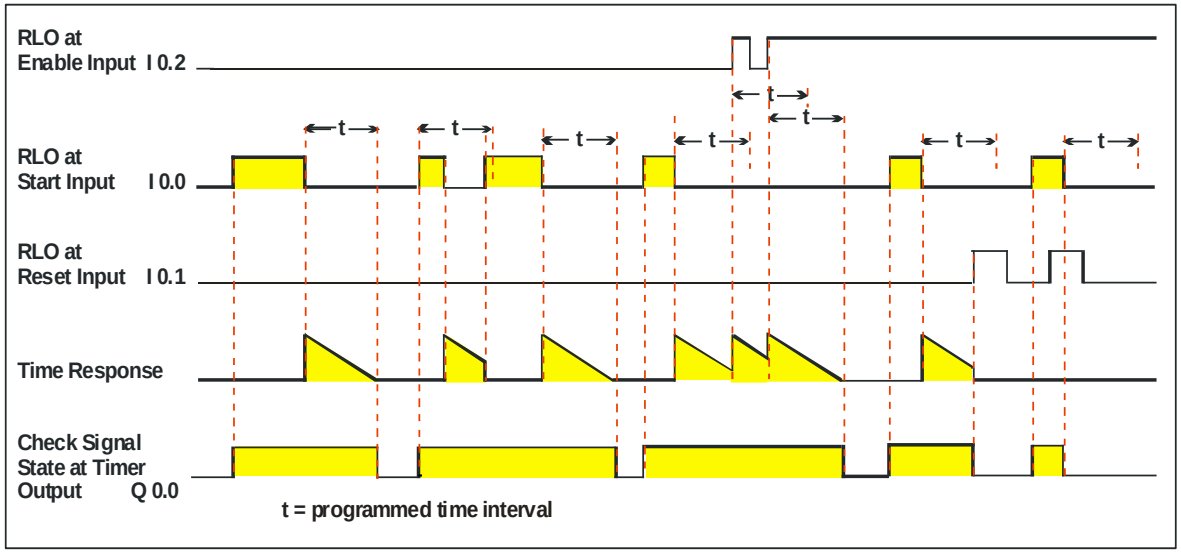
SF <timer> starts the addressed timer when the RLO transitions from "1" to "0". The programmed time elapses as long as RLO = 0.

The time is stopped if, the RLO transitions to "1" before the programmed time interval has expired.

This timer start command expects the time value and the time base to be stored as a BCD number in ACCU 1-L.

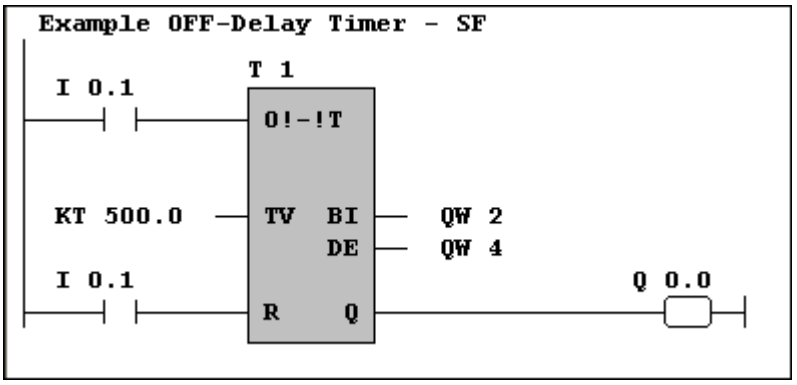
Status word

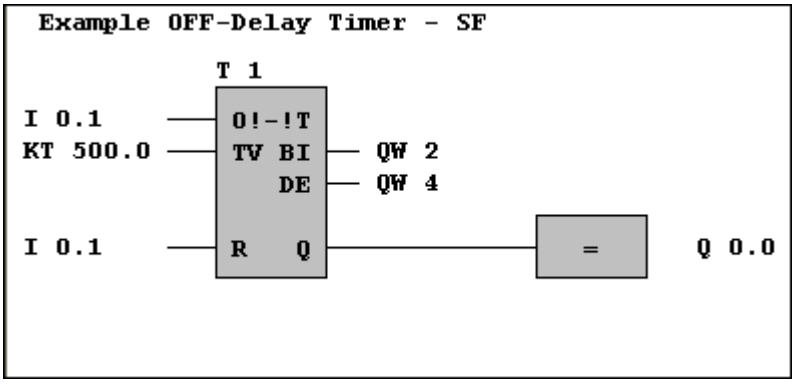
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0



Off-Delay Timer (SF) (continued)

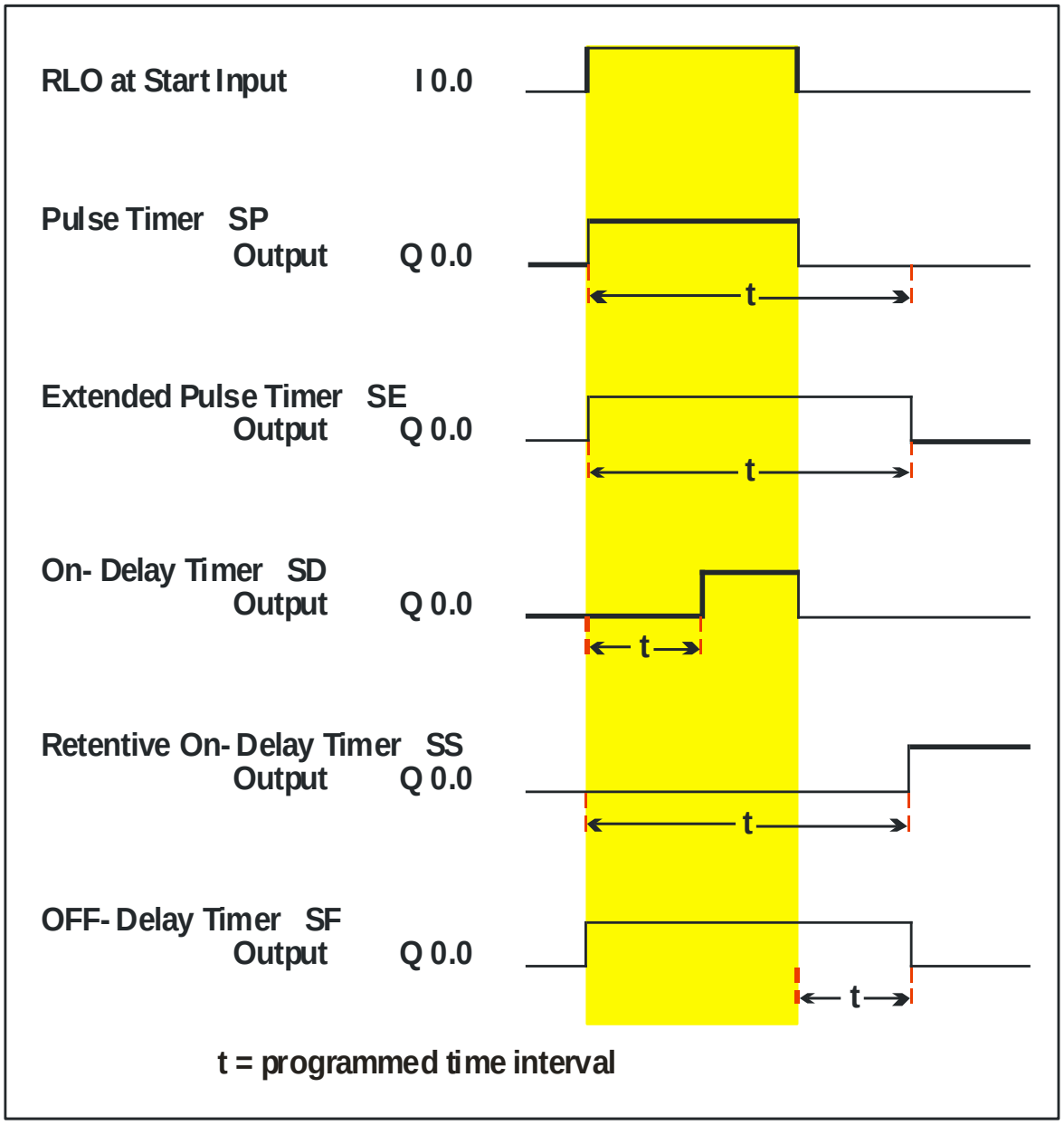
Tag	Instruction	Operand	Comment
; Example OFF-Delay Timer - SF			
	A	I 0.0	
	FR	T 1	; Enable Timer T1
	A	I 0.1	; Start Timer T1
	L	KT 500.0	; Timer preset 5 seconds in Accu 1
	SF	T 1	; Puls Timer T1
	A	I 0.1	; Reset signal
	R	T 1	; Reset Timer T1
	L	T 1	; Load current Timer value (T1) into ACCU1 in binary format
	T	QW 2	; Transfer to Output word
	LC	T 1	; Load current Timer value (T1) into ACCU1 in BCD format
	T	QW 4	; Transfer to Output word
	A	T 1	; Check signal state of Timer T1
	=	Q 0.0	





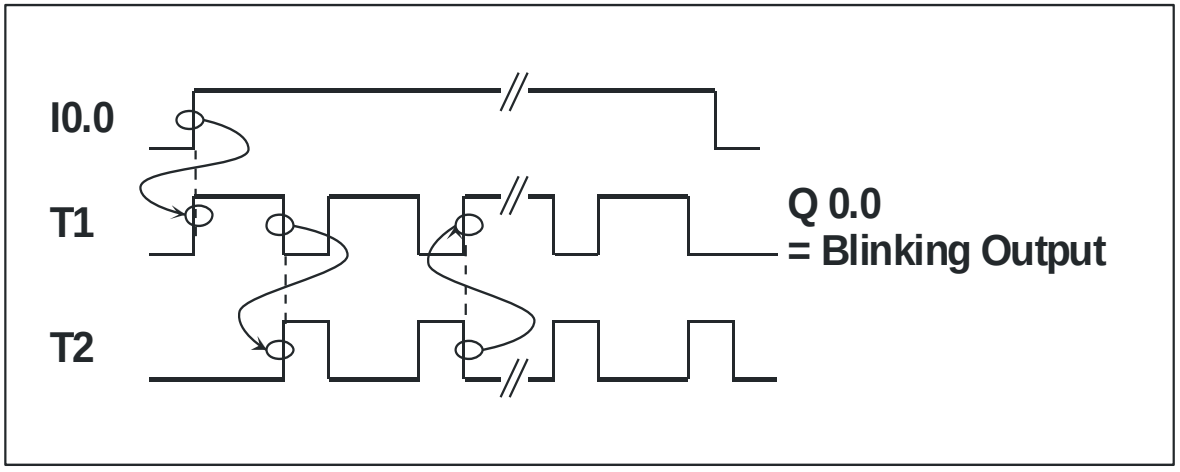
Selecting the right Timer

The following overview should help you to select the right timer for your timing application.



Practice Exercise 4–1; Flashing Light

A flashing light with an ON time of 1 second and an OFF time of 0.5 second needs to be programmed. The ON time and the OFF time should be separately adjustable.



Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

Practice Exercise 4–2; Traffic Light

A pedestrian crossing light needs to be controlled. If a pedestrian pushes the "Walk" button, the traffic light should be switched to "Red" for the cars and "Green" for the pedestrian crossing light.

The yellow phase for the automobiles should be 3 seconds and the red phase 8 seconds. The pedestrians have a green phase of the 5 seconds. The automobiles should have a green phase from at least 4 seconds.

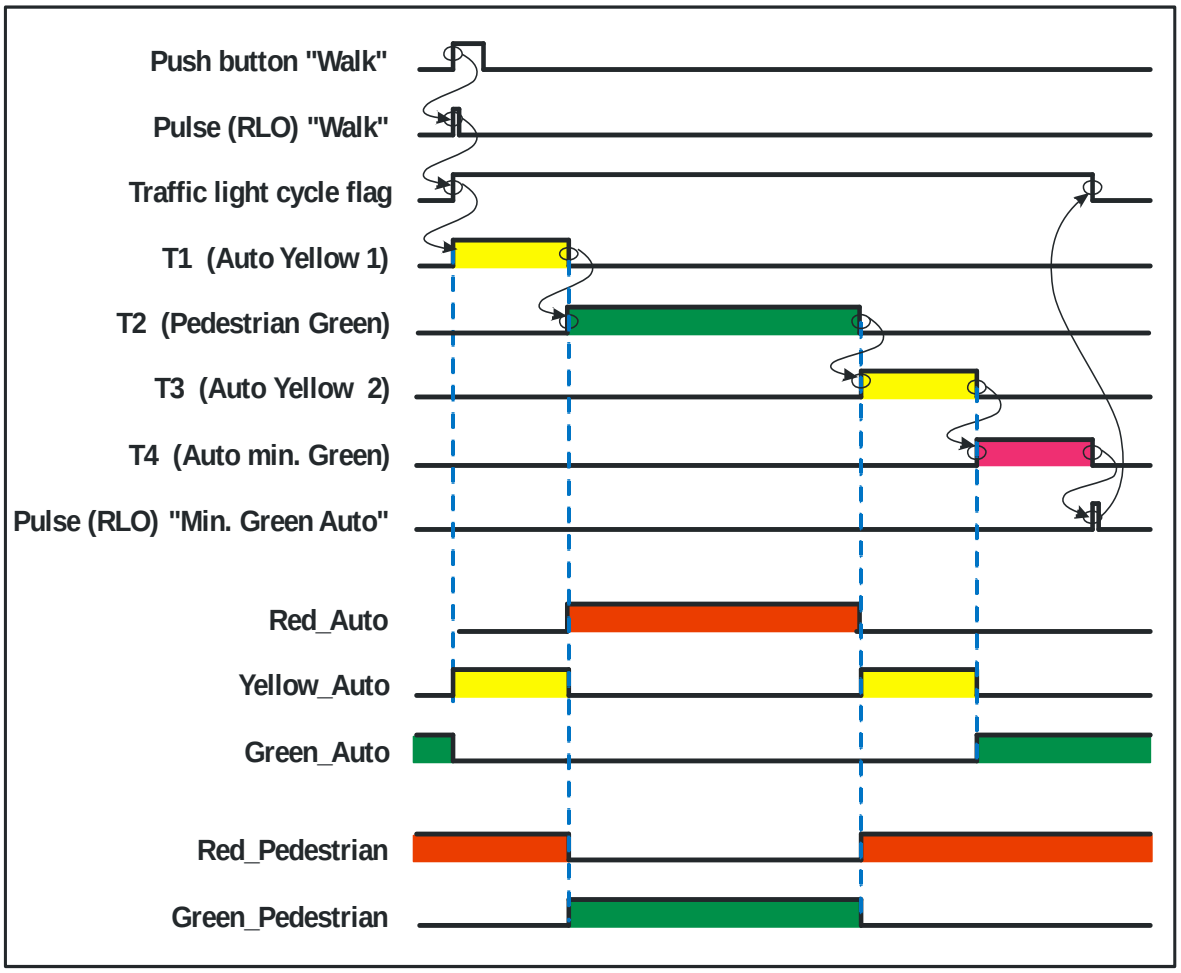
Comment	Operands
Push button "Walk"	I 0.0
Red light Automobiles	Q 0.0
Green light Automobiles	Q 0.2
Yellow light Automobiles	Q 0.1
Red light Pedestrian	Q 0.3
Green light Pedestrian	Q 0.4
Time Value for Timer T1	3 seconds
Time Value for Timer T2	5 seconds
Time Value for Timer T3	3 seconds
Time Value for Timer T4	4 seconds
Traffic light cycle flag	F10.0
Edge Flag for “Walk”	F10.1
Edge Pulse “Walk”	F10.2
Edge Flag for “Output T4”	F10.3
Edge Pulse “Output T4”	F10.4

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

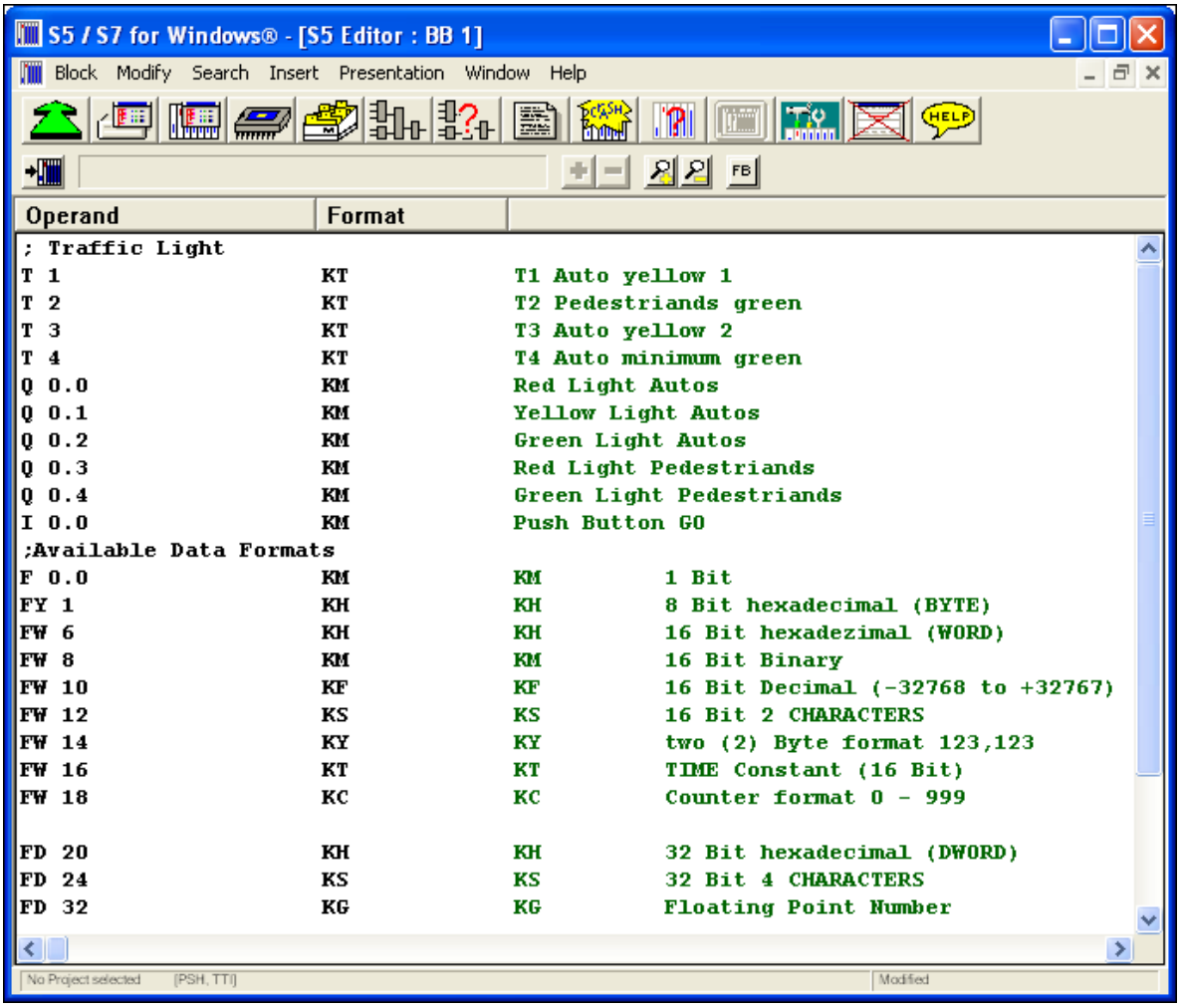
Test the PLC program.

Traffic Light Control



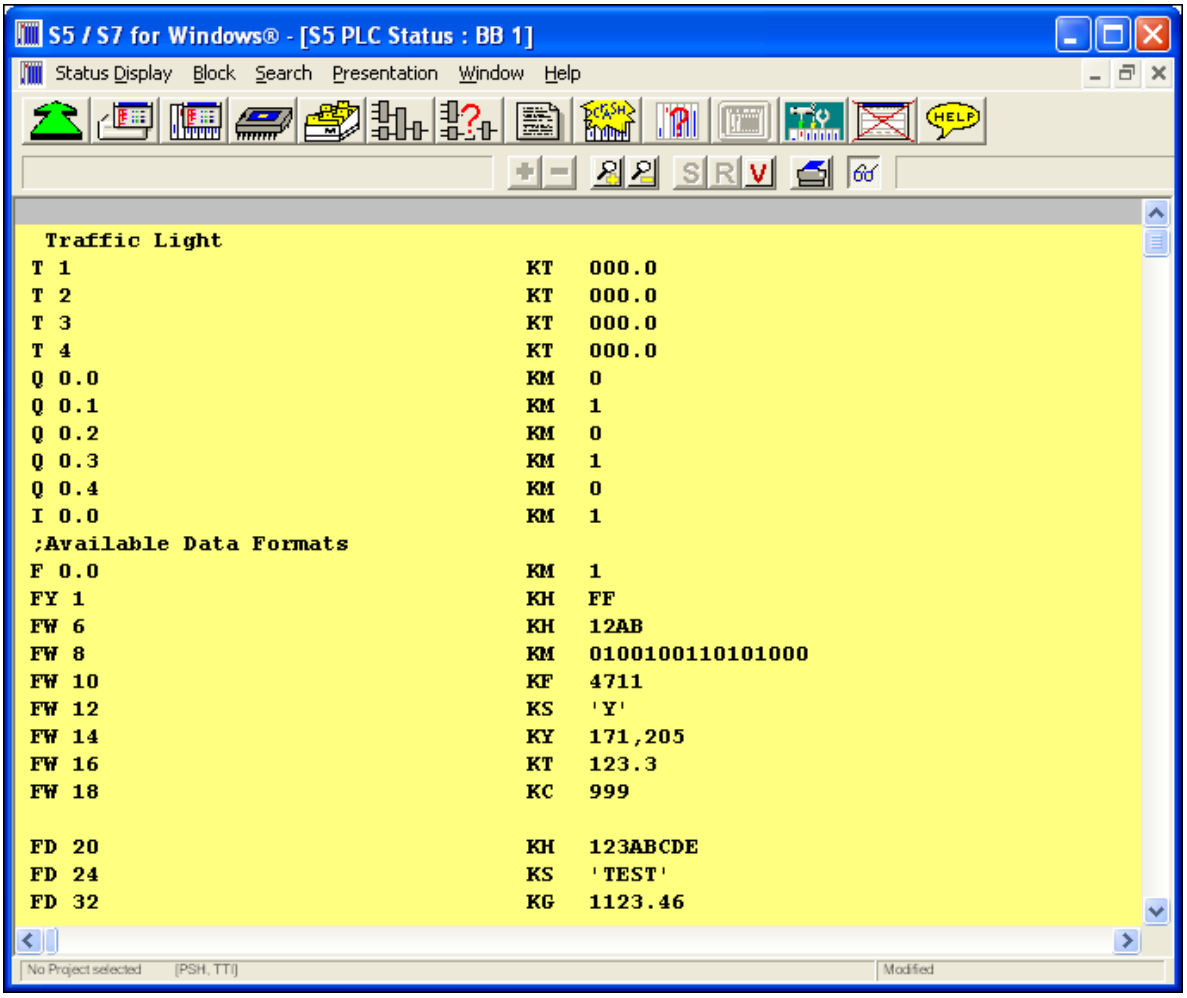
Picture Block; Editor

Available Data Formats:



Picture Block; Status Display

Available Data Formats:



4.2 Counter Instructions

A counter is a function element of the STEP® 5 programming language that counts. Up and Down counting is possible. Counters have an area reserved for them in the memory of your CPU. This memory area reserves one 16-bit word for each counter.

Counter instructions are the only functions with access to the memory area. You can vary the count value within this range by using the following Counter instructions:

- FR Enable Counter (Free)
- L Load Current Counter Value into ACCU 1
- LC Load Current Counter Value into ACCU 1 as BCD
- R Reset Counter
- S Set Counter Preset Value
- CU Counter Up
- CD Counter Down

Enable Counter FR (Free)

Format: FR <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter, range depends on CPU

When RLO transitions from "0" to "1", FR <counter> clears the edge-detecting flag that is used for setting and selecting the counting direction (up or down) of the addressed counter.

Enable counter is not required to set a counter or for normal counting. This means that in spite of a constant RLO of 1 for the Set Counter Preset Value, Counter Up, or Counter Down, these instructions are not executed again after the enable.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

Example: Enable Counter FR (Free)

Tag	Instruction	Operand	Comment
;Enable Counter			
	A	I 0.0	; Check signal state for input I 0.0
	FR	C 1	; Enable counter C1 when RLO transitions from 0 to 1

Reset Counter R

Format: R <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter to be reset, range depends on CPU

R <counter> loads the addressed counter with "0" if RLO = 1.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

Example:

Tag	Instruction	Operand	Comment
; Reset Counter			
	A	I 0.0	; Check signal state for input I 0.0
	R	C 1	; Reset counter C1 to a value of 0 if RLO = 1

Set Counter S (Preset Counter)

Format: S <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter to be preset, range depends on CPU

S <counter> loads the count from ACCU 1-L into the addressed counter when the RLO transitions from "0" to "1". The count in ACCU 1 must be a BCD number between "0" and "999".

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

Example:

Tag	Instruction	Operand	Comment
; Set Counter Preset Value			
	A	I 0.0	; Check signal state for input I 0.0
	L	KC 123	; Load the preset value (123) into Accu1 (in BCD)
	S	C 1	; Enable counter C1 when RLO transitions from 0 to 1

Load Current Counter Value (L) into ACCU 1 in Binary Form

Format: L <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter range depends on CPU

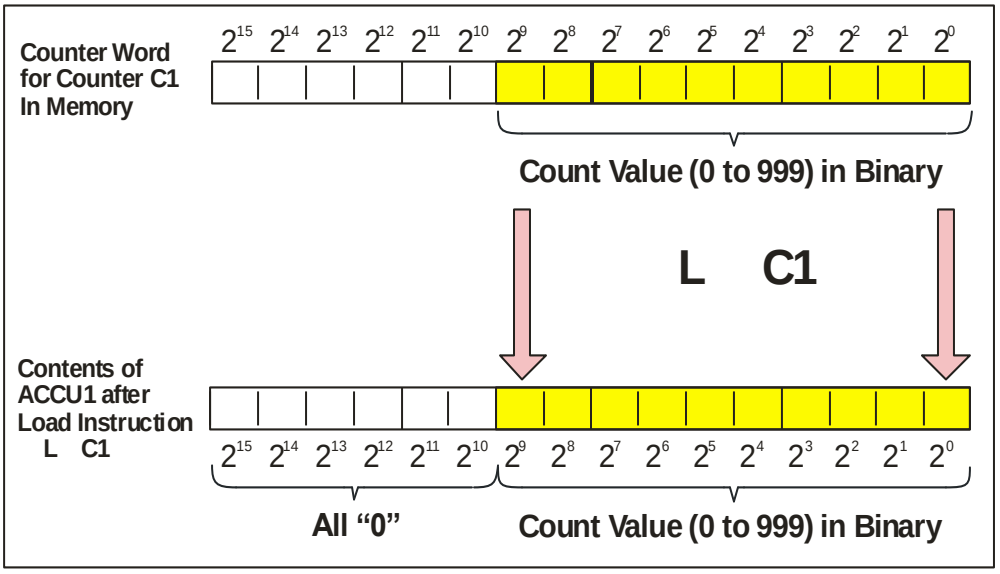
L <counter> loads the current count of the addressed counter as a binary number into ACCU 1-L (between "0" and "999").

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	-	-	-	-

Example:

Tag	Instruction	Operand	Comment
; Load Counter Value into Accu 1			
	L	C 1	; Load the momentary value of counter C1 into Accu 1 (Binary)



Load Current Counter Value (LC) into ACCU 1 in BCD Form

Format: LC <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter range depends on CPU

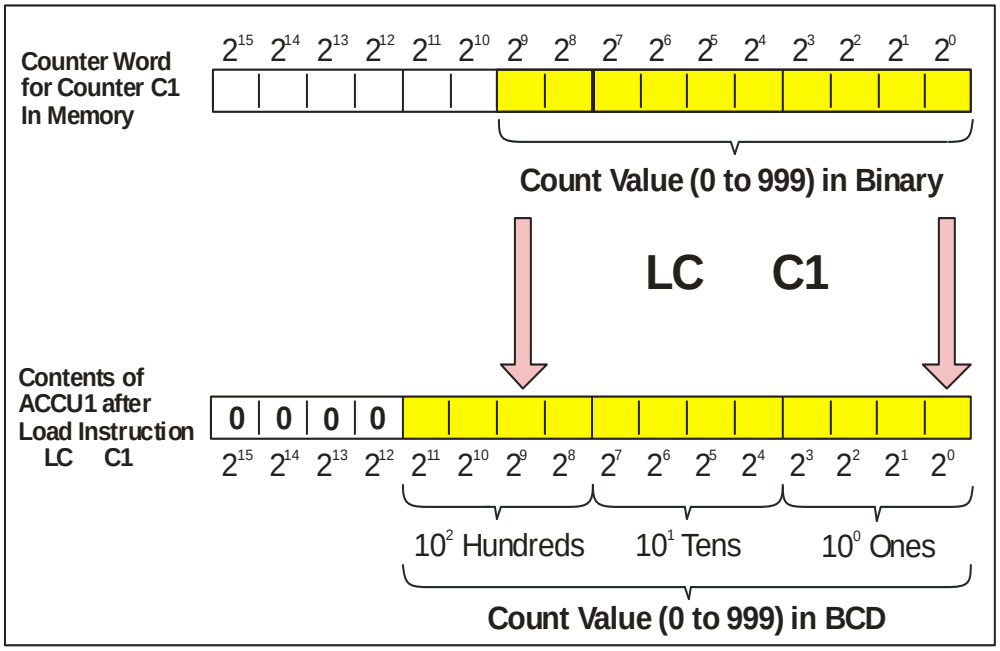
LC <counter> loads the current count of the addressed counter as a BCD number into ACCU 1-L (between "0" and "999").

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	-	-	-	-

Example:

```
Tag         Instruction   Operand     Comment
; Load Counter Value into Accu 1

            L             C  1        ; Load the momentary value of counter C1 into Accu 1 (BCD)
```



Counter Up (CU)

Format: CU <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter range depends on CPU

CU <counter> increments the count of the addressed counter by 1 when RLO transitions from "0" to "1" and the count is less than "999". When the count reaches its upper limit of "999", incrementing stops. Additional transitions of RLO have no effect and overflow OV bit is not set.

Status word

	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

Example:

Tag	Instruction	Operand	Comment
; Set Counter Preset Value			
	A	I 0.0	; Check signal state for input I 0.0
	CU	C 1	;Counter is incremented by 1 when RLO transitions from 0 to 1

Counter Down (CD)

Format: CD <counter>

Address	Data type	Memory area	Description
<Counter>	COUNTER	C	Counter range depends on CPU

CD <counter> decrements the count of the addressed counter by 1 when RLO transitions from "0" to "1" and the count is greater than "0". When the count reaches its lower limit of "0", decrementing stops. Additional transitions of RLO have no effect as the counter will not count with negative values.

Status word

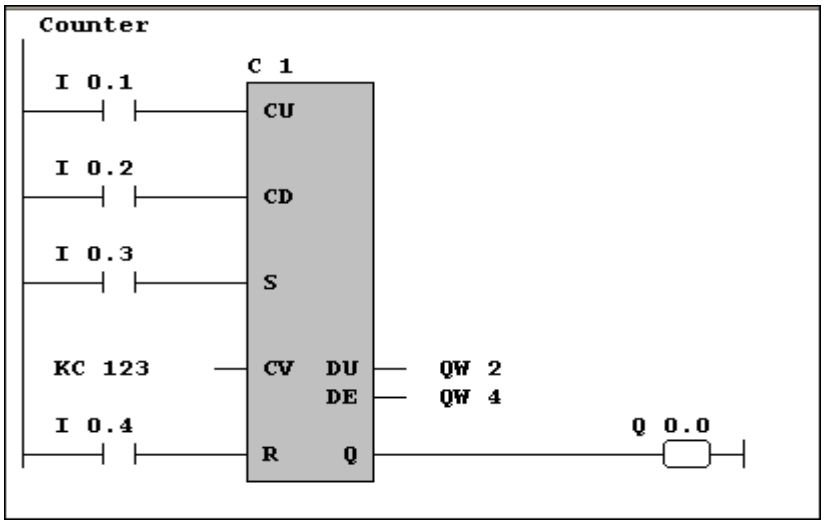
	CC 1	CC 0	OV	OS	OR	STA	RLO	/FC
writes:	-	-	-	-	0	-	-	0

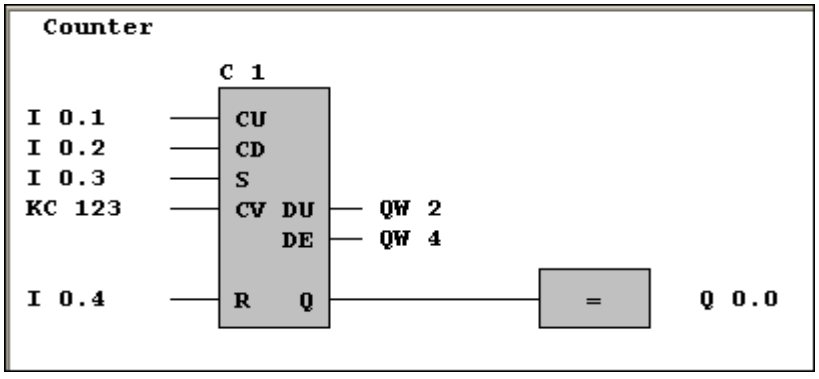
Example:

Tag	Instruction	Operand	Comment
; Set Counter Preset Value			
	A	I 0.0	; Check signal state for input I 0.0
	CD	C 1	;Counter is decremented by 1 when RLO transitions from 0 to 1

Counter (continued)

```
; Counter

        A           I  0.0      ; Check signal state for input I 0.0
        FR          C  1        ; Enable counter C1 when RLO transitions from 0 to 1

        A           I  0.1      ; Check signal state for input I 0.1
        CU          C  1        ;Counter is incremented by 1 when RLO transitions from 0 to 1

        A           I  0.2      ; Check signal state for input I 0.2
        CD          C  1        ;Counter is decremented by 1 when RLO transitions from 0 to 1

        A           I  0.3      ; Check signal state for input I 0.3
        L           KC 123      ; Load the preset value (123) into Accu1 (in BCD)
        S           C  1        ; Enable counter C1 when RLO transitions from 0 to 1

        A           I  0.4      ; Check signal state for input I 0.4
        R           C  1        ; Reset counter C1 to a value of 0 if RLO = 1

        L           C  1        ; Load the momentary value of counter C1 into Accu 1 (Binary)
        T           QW 2        ; save momentary value of counter C1

        LC          C  1        ; Load the momentary value of counter C1 into Accu 1 (BCD)
        T           QW 4        ; save momentary value of counter C1

        A           C  1        ; Check signal state for conter C1 output
        =           Q  0.0      ; Assign RLO to output Q 0.0
```





Practice Exercise 4–3; Counter

The number of times a compressor is switched on needs to be monitored. If the compressor is switched on, a counter should be incremented. The input signal
I 0.0 is used to monitor the on stage of the compressor.

The number, how many times the compressor has been switched on should be displayed in the output word QW2.

A light should indicate if the compressor was switched on at least once (output Q 0.0).

The input "I 0.1" resets the counter.

To understand the function of the counter better, the down count input (I 0.2), the set input (I 0.3) with the set value (IW 2) should be used.

Also the second output giving the momentary value of the counter should be displayed at an output word (QW 4).

Write a PLC program with the S5 Blocks PB10 and OB1.

Transfer of the program into the S5 TEST PLC.

Test the PLC program.

5 Function Blocks (FB; FX) and Data Blocks (DB; DX)

Advanced Step 5 programming requires the use of Function Blocks in conjunction with Data Blocks. A lot of Step 5 operations can only be performed in Function Blocks.

5.1 Programming Function Blocks

As a rule, program blocks contain the largest portion of an application program.

Only basic operations, however, may be programmed in these blocks.

Function blocks must be used to implement control tasks which require supplementary operations. Function blocks are also used when a control function (for an individual control element, for example) occurs frequently in a program. In such cases, it is possible to make use of one of the biggest advantages proffered by function blocks: the fact that they can be assigned runtime parameters, i.e. when a function block is invoked, the user may specify the operands with which it is to execute.

This can be done each time the block is called, thus enabling a block that is present in memory only once to be used repeatedly for the same function, but with different operands each time.

Essentially, the following characteristics distinguish function blocks from program blocks:

- They can be programmed using the CPU's full operations set.
- Function blocks can be assigned different parameters each time they are called.
- Function blocks have names.

These characteristics make it possible to utilize the CPU's full capabilities. On the other hand, function blocks are not as easy to program as, for example, program blocks.

All information included in this section applies both to "normal" function blocks (FBs) and extended function blocks (FXs)

Programming Function Blocks (continued).

Function Blocks (FB, FX) are made up of STEP® 5 instructions. The PLC program or parts of the program are stored in FB's. Especially complex or recurring program sequences are accomplished within FB's. Comments may be added. The instructions may be edited and displayed in STL, CSF, and LAD (optional).

The first segment, with the name and the identifiers, must be programmed using STL presentation.

Note:

A Function Block (FB; FX) always have a name. The name can have up to eight characters (A – Z and 0 – 9). The only special character allowed in the colon "**:**".

Function blocks can be roughly divided into two categories: those with and those without block parameters.

Function Blocks Without Block Parameters

Function blocks without block parameters are programmed in essentially the same way as program blocks. The user has to enter the name of the function block (which may comprise up to eight characters). Programming can be continued in the "normal" way following entry of the function block name, including statements from the supplementary operations set.

The function block name is stored in the block header; the programmer thus has the name at its disposal at all times. A function block header is therefore longer than the headers of other blocks.

Function Blocks With Block Parameters

If Function Blocks are to be assigned block parameters, these must be specified with name, parameter type and data type following entry of the block name (see next subsection). Once all block parameters have been entered, the user must program the control function.

It stands to reason that the block parameters (and the program) should be carefully defined before beginning programming.

Function Blocks With Block Parameters (continued).

Although it is possible to delete or insert a block parameter once a function block has been programmed, insertions and deletions result in renumbering of the block parameters.

Since the (MC 5) program contains only the numbers (not the names!) of the block parameters, it is necessary, after changing the block parameter list, to check the entire function block program to see whether the parameter assignments are still correct. This can involve considerable overhead.

Modifications in the parameter type and data type of block parameters also normally necessitate a full program check.

Function block programs may also include the “Substitution Statements”.

The programmer automatically stores the block parameter specifications in the block header, behind the function block name. The block header thus contains all the information the programmer requires to

- display the names of the block parameters for the purpose of operator guidance and
- carry out an operand check during programming and initialization of the function block.

As many as 40 block parameters may be programmed. In practice, block parameters are normally restricted to approximately a dozen for purposes of manageability and clarity.

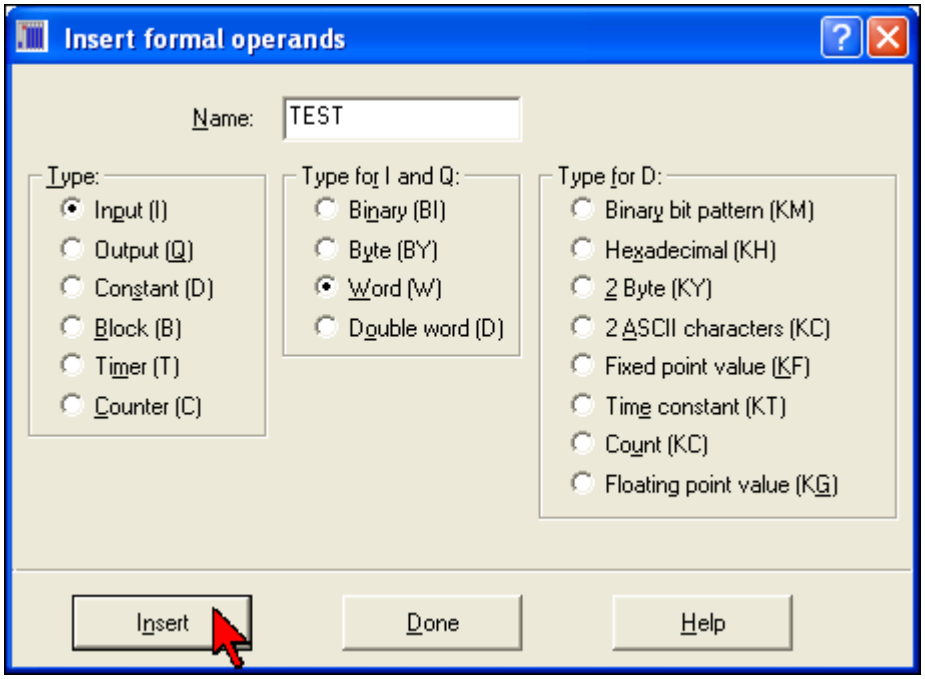
Block Parameters

A block parameter is classified by its name (identifier), its parameter type and its data type, all of which must be entered.

S5 for Windows® provides a dialog box to insert the ***FB / FX Formal Operands*** (Block Parameters).

With this dialog box you can easily insert a formal operand parameter by name (Declaration **- DECL:**), its type and its data configuration.

Insert Formal Operand dialog box



Name

In the text field enter the name of the block parameter. The name may be up to four (4) characters long and must start with a letter. The name is automatically entered in capital letters. The block parameter name is identical to the formal operand specified in the program in place of the actual operand.

Type

A marked button identifies the block parameter type. Input parameter, output parameter and parameters representing a constant, need further definitions.

Type for I and Q

Input and output parameters need a further definition. With the buttons you may define if an input or output parameter represents a bit (BI), a byte (BY), a word (W), or a double word (D).

Type for D

A parameter representing a constant needs further definition. The value may be presented in different forms. Mark the button to select the required form.

Insert button

If you activate the “Insert” button the defined parameter will be entered into the function block. The dialog box stays open and is ready to define the next parameter. Up to forty block parameters may be defined per function block.

Done button

If you activate the “Done” button the defined parameter will be entered into the function block and the dialog box will be closed.

Note:

Prior to opening the dialog box **Insert Formal Operand** you must position the insertion mark in a separate line directly below the line defining the name.

Parameter type

A block parameter may be of type "I", "Q", "D", "B", "T" or "C".

- I = Input parameter
- Q = Output parameter
- D = Data
- B = Block
- T = Timer
- C = Counter

In graphic representation, parameters of type "I", "D", "B", "T" and "C" appear at the left, parameters of type "Q" at the right of the function symbol.

Data type

The programmer checks the data type when the function block call is initialized.

The following data types are permitted for parameters of type "I" or "Q":

- BI for an operand with bit address
- BY for an operand with byte address
- W for an operand with word address
- D for an operand with doubleword address

The following actual operands are permissible for data type "BI":

- I n.m Input
- Q n.m Output
- F n.m Flag

The following actual operands are permissible for data type "BY":

- IB n Input byte
- QB n Output byte
- FY n Flag byte
- DL n Left (i. e. high-order) data byte
- DR n Right (i. e. low-order) data byte
- PY n Peripheral byte
- OB n Extended peripheral byte

The following actual operands are permissible for data type "W":

- IW n Input word
- QW n Output word
- FW n Flag word
- DW n Data word
- PW n Peripheral word
- OW n Extended peripheral word
- RS n System data word
- RT n Extended system data word
- RI n System transfer data word
- RJ n Extended system transfer data word

System data areas RS, RT, RI and RJ can only be output if the function block call is itself within a function block. System data areas RT and RJ are only available with all CPUs.

The following actual operands are permissible for data type "D":

- ID n Input double word
- QD n Output double word
- FD n Flag double word
- DD n Data double word

The following data types are permissible for block parameters of type “D”:

- KM Binary constant (16 digits)
- KH Hexadecimal constant (max. 4 digits)
- KY Two one-byte absolute values, each in the range 0 to 255, separated from one another by a comma
- KS Character constant (max. 2 alphanumeric characters)
- KF Fixed-point number in the range -32 768 to +32 767
- KT Time constant (BCD) with time base 1.0...999.3
- KC Count constant (BCD) in the range 0...999
- KG Floating-point number in the range $\pm$ 1.701411 x$10^{\pm 38}$

No data type specification is permitted for block parameters of type “B” The following actual operands are permissible:

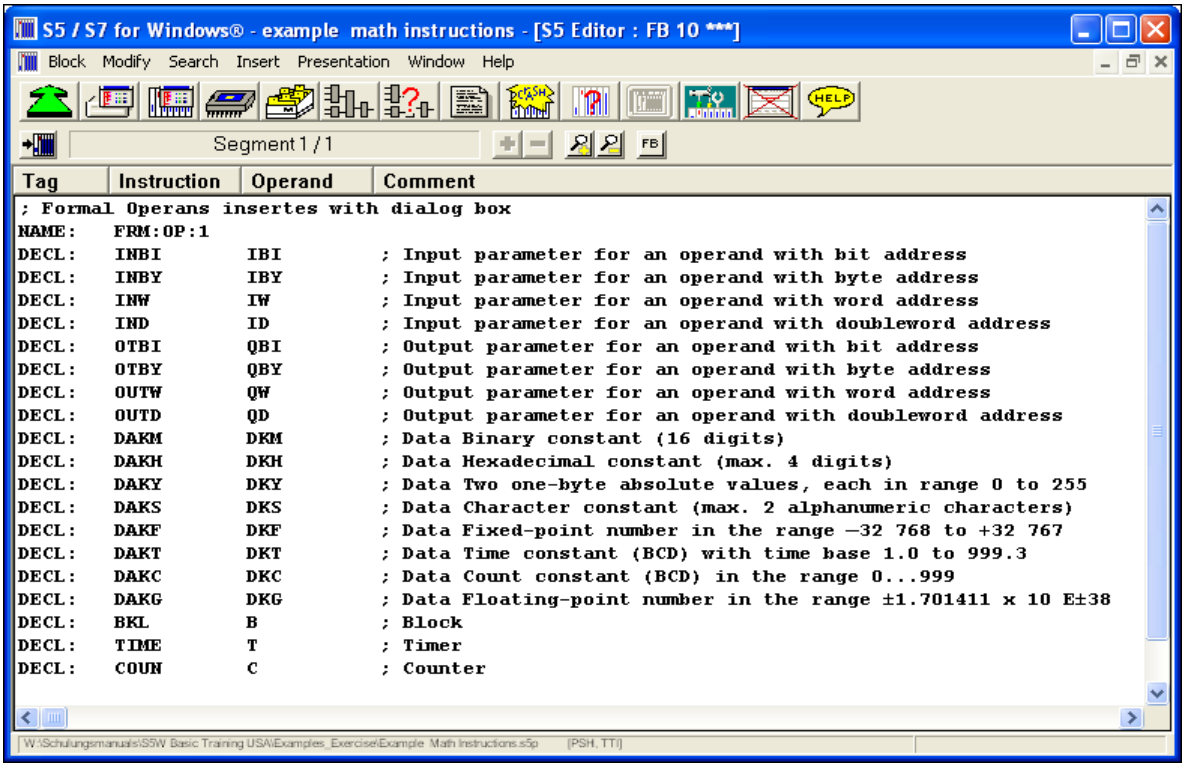
- DB n Data blocks; the C DBn statement is executed
- FB n Function blocks (without parameter list)
- PB n Program blocks
- SB n Sequence blocks

All blocks are called unconditionally (JU...n).

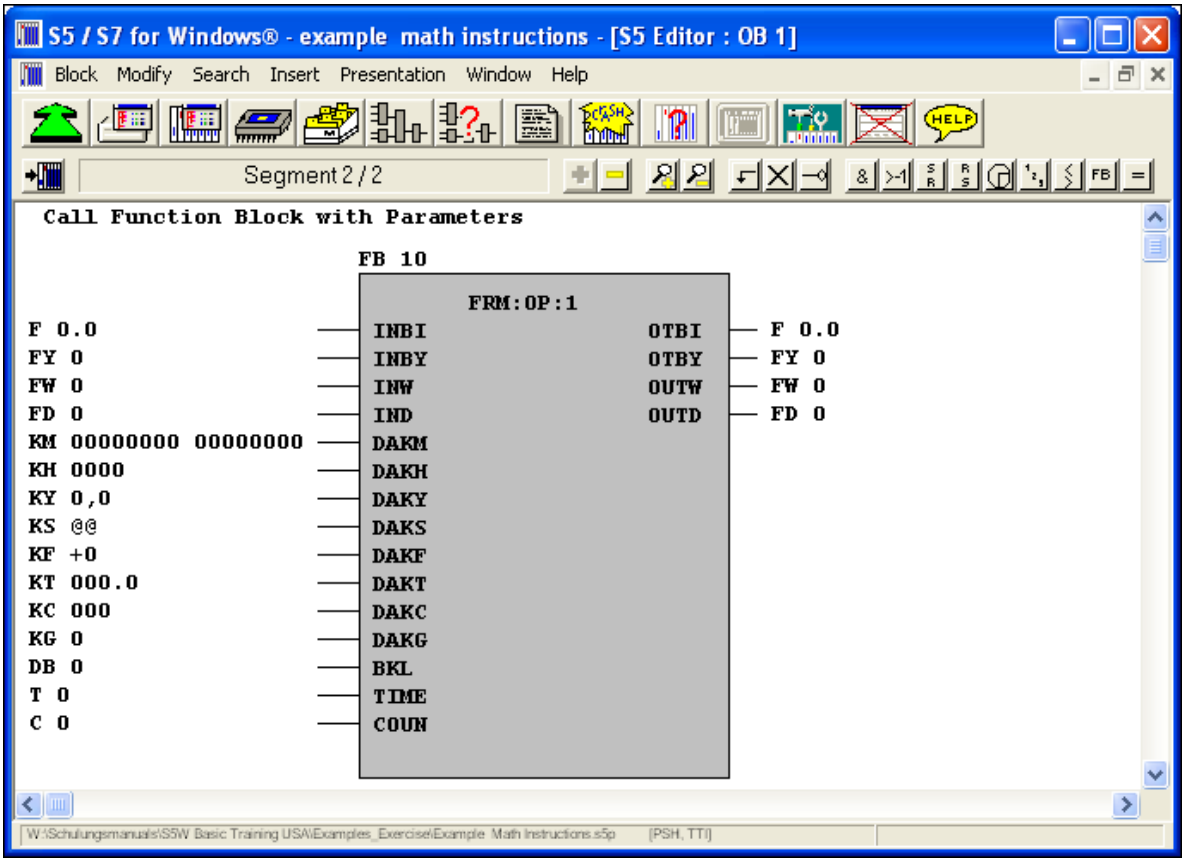
No data type specification is permitted for block parameters of type “T”. Only the T (Timer) operand is allowed when initializing called function blocks.

No data type specification is permitted for block parameters of type “C”. Only the C (Counter) operand is allowed when initializing called function blocks.

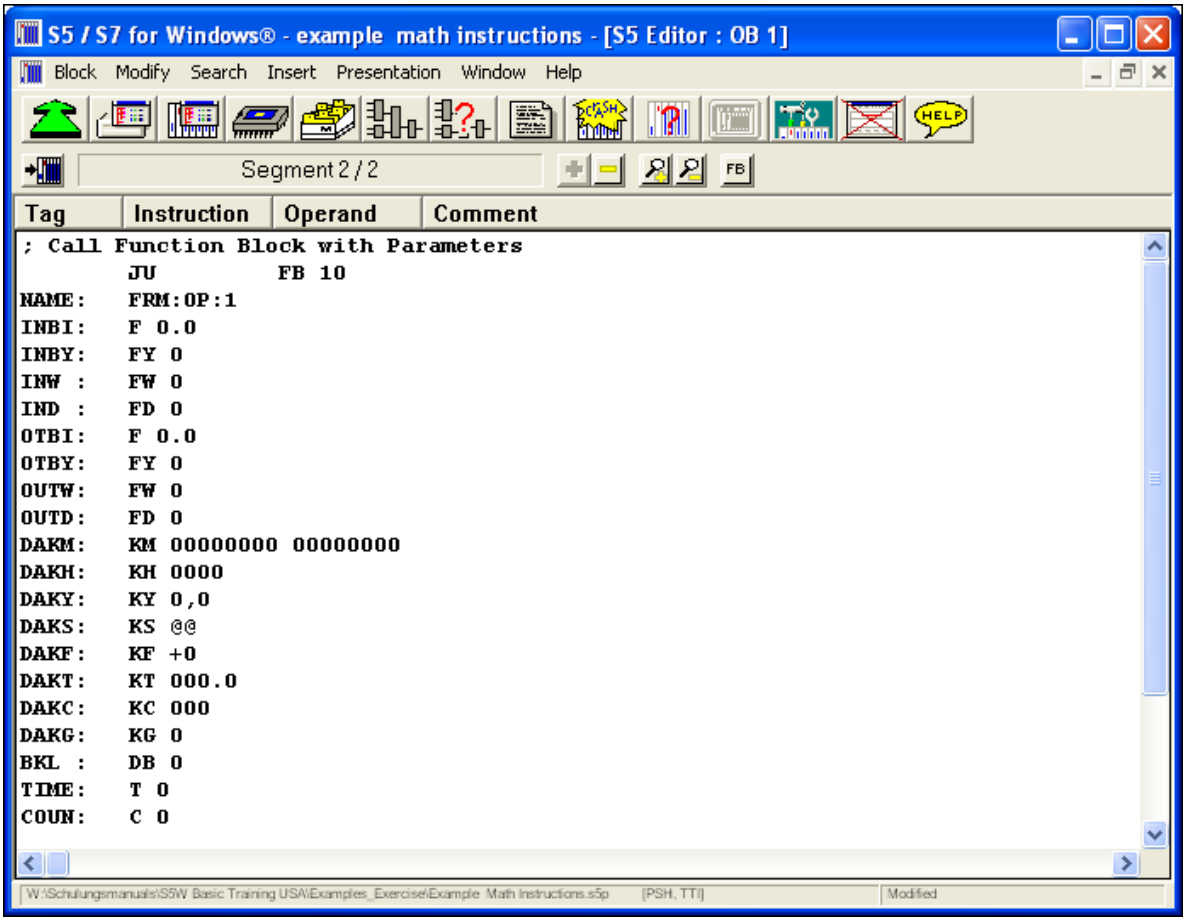
Block Parameters (Formal Operands) defined in a FB



Calling a Function Block with Parameters (graphic presentation)



Calling a Function Block with Parameters (STL)



The ***Place holder*** must be replaced with the **Actual Parameters** to be used in the called Function Block.

5.2 Data Blocks

The data blocks contain the data for the user program. A data block comprises 256 data words. Should this prove insufficient, the data block is changed and a new data block invoked. All operations with operand identifier D then access the “new” data block.

An unconditional data block call (C DB or CX DX) is executed without regard to any conditions whatsoever. All data subsequently addressed refer to this data block. Cyclic program scanning is not interrupted, and neither the RLO nor the contents of the accumulators are affected.

All data blocks must be “generated” before they can be used (i.e. before data can be read from or written to them), that is to say, space must be reserved for the data. Data blocks can be generated over the programmer or with the G DB or GX DX operations. Attempts to access non-existent data blocks may produce an undefined state or result in flagging of a “transfer error” (TRAP) (“Load/Transfer Errors (OB 32”).

Before data can be used, the relevant data block must be called. A data block remains “valid” until another data block is called. If the data block is changed within an invoked (“lower level”) block, the “new” data block remains valid until exited, at which point the “old” data block becomes valid again (in the “higher-level” block).

Note:

Like all other software blocks, data blocks can be up to 2048 and/or 4096 words long depending on the CPU version; however, the area that can be addressed direct with STEP 5 operations is limited to the first 256 data words.

Calling Data Blocks

Data blocks may only be called from segments in STL presentation.

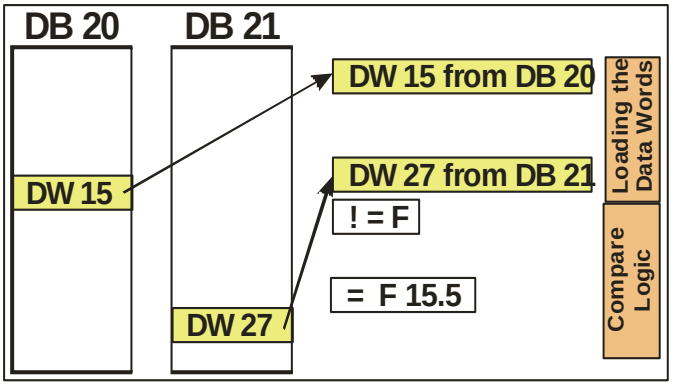
- Create a new segment by activating the commands ***Add New Segment*** or ***Insert Segment*** from the modify menu. You may also create a separate block.
- Select STL presentation.
- Enter **C DB**nn or **CX DX**nn.

Example:

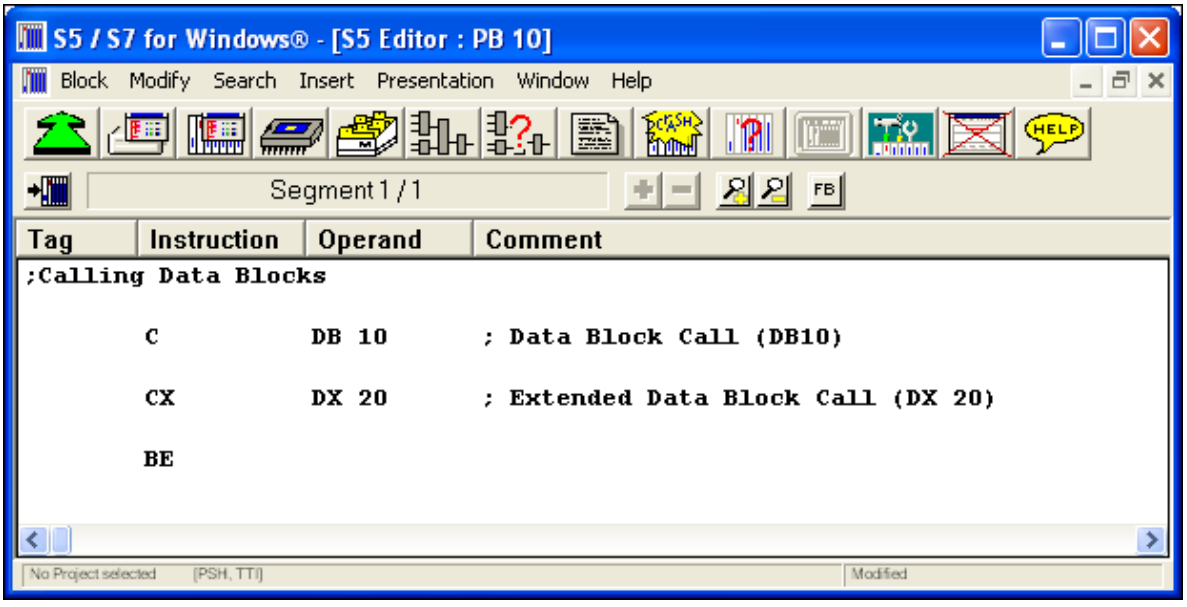
Comparing the value in data word DW 15 in data block DB 20 and the value in data word DW 27 in data block DB 12 and setting flag F 15.5 if they are identical (i.e. equal).

C	DB 20	Call data block DB. 20
L	DW 15	Load the value in data word DW 15, data block DB 20
C	DB 21	Call data block DB 21
L	DW 27	Load the value in data word DW 27, data block DB 21
!=F		Compare the two data words for “equal”
=	F 15.5	Set flag F 15.5 to “1” if the comparison is true

Calling the Data Blocks DB 20 and DB 21



Data Block Call (DB, DX)



Opening another Data Block in a called Block

Data block DB 5 is called in program block PB 5.

The program subsequently processes the data in this data block.

When program block PB 10 is called, both the jump address and the data area valid at this address (in this case DB 5) are pushed onto the stack.

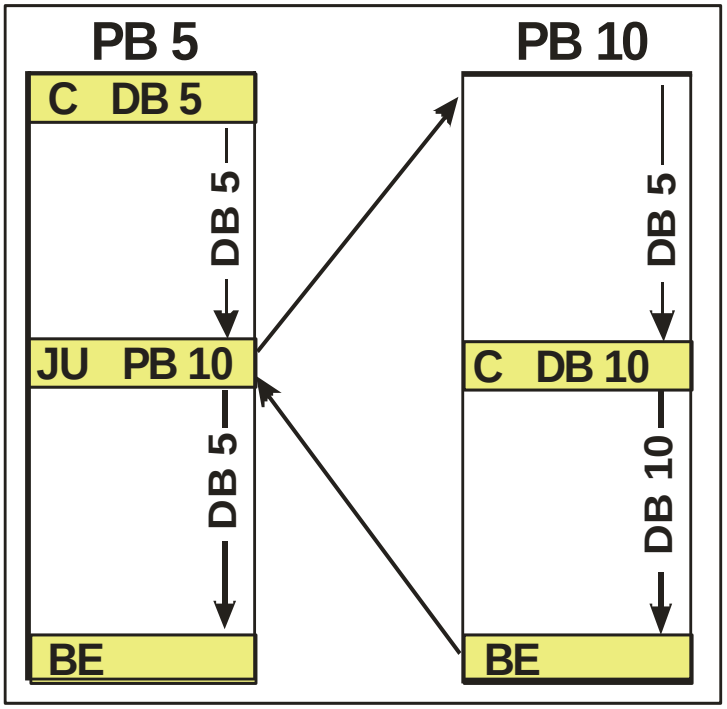
Program block PB 10 is then executed.

Data block DB 5 is still valid at this point, and remains so until data block DB 10 is invoked.

Data block DB 10 then remains valid until the final statement in program block PB 10 has been processed.

Upon return to program block PB 5, both the jump address and the address of data block DB 5 are popped from the stack, and PB 5 resumes execution using the data in DB 5.

Data block DB 10 was thus "local" to program block PB 10.



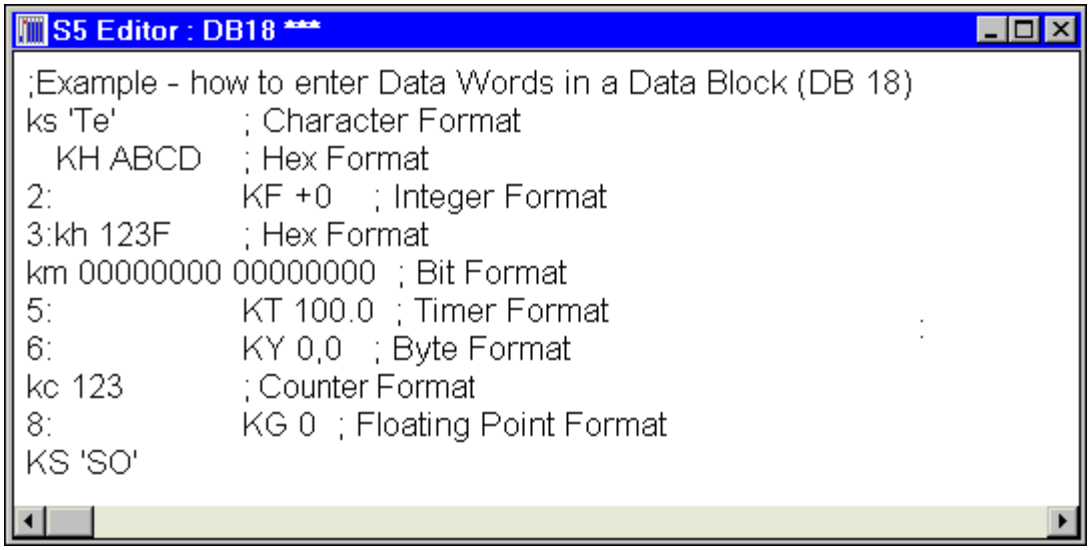
Creating a Data Block (DB, DX)

To create a data block, perform the following steps.

- Open a data block by activating the ***New Block*** command from the block menu (PC or PLC block list window).

When you open a new data block, STL presentation is automatically selected from the presentation menu (editor window).

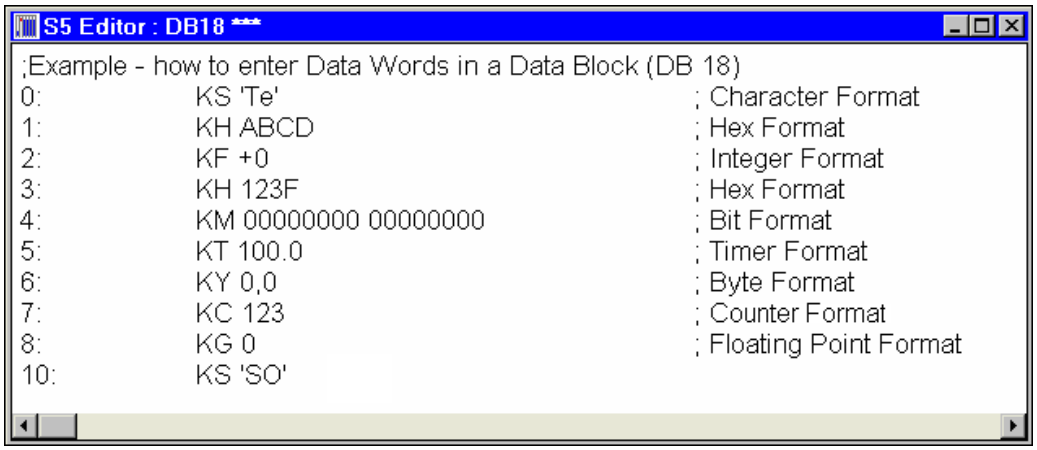
- A data block is not divided into segments. The icons to select the previous or next segment are not active and no block end mark (**BE**) is shown.
- Enter the text as shown below (Data Block (DB, DX) prior formatting)



To enter the data words you must follow a defined syntax.

- Spaces within data type declaration (**K H** is not permitted) or within numbers (**123 456**) are not permitted.
- A comment after a data word that is separated by a semicolon (**;**) is permitted.
- A separate line comment is not permitted.
- The data words are automatically (using the format command) numbered starting with data word zero (0). If you enter numbers (e.g. **5:**) they are ignored.
- Format the data block (key F9) and save the block.

The formatted data will have the following form (Data Block (DB, DX) formatted).

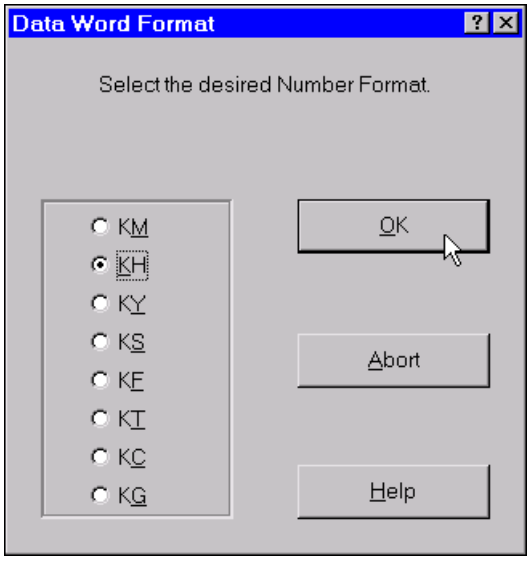


Changing the Data Word Format

A dialog box is provided to modify the format of a data word.

- Mark a data word.

- Click ***Change Type*** in the modify menu.
- Select the new data word format by activating the desired button and confirm the selection (**OK** button).

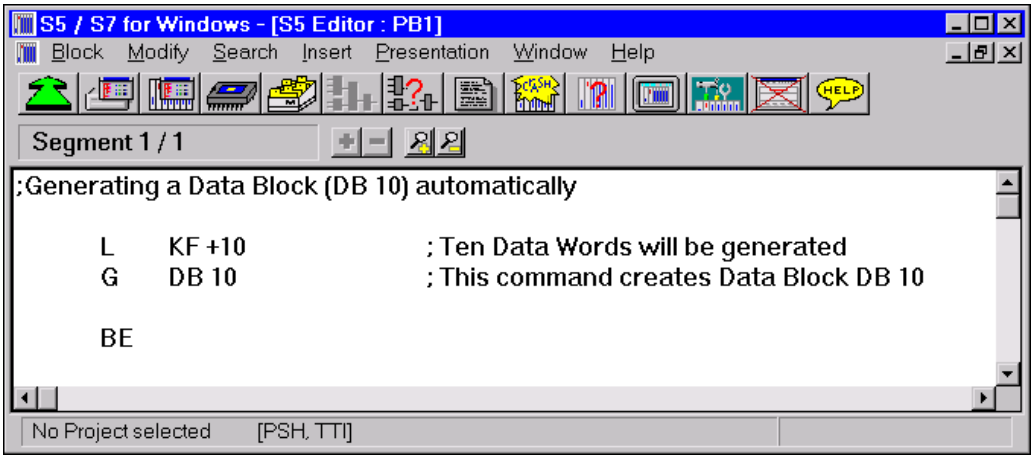


Possible Data Word Formats (constants).

Format	Limits		Explanation
	lower	upper	
KM	00000000.00000000	11111111.11111111	arbitrary bit pattern (16 bit)
KH	0000	FFFF	hexadecimal code
KY	000.000	255.255	two (2) byte (address)
KS	two ASCII characters, max. 24 chr. per line		text format
KF	- 32768	+ 32767	integer (fixed point number)
KT	000.0	999.3	time value with multiplier
KC	0	999	count
KG	- 1469368 - 38	+ 17014112 + 39	floating point value

Creating a Data Block (DB, DX) automatically

The generate data block statement may also be used to create a data block. To do so you must write a segment as shown in the following picture.

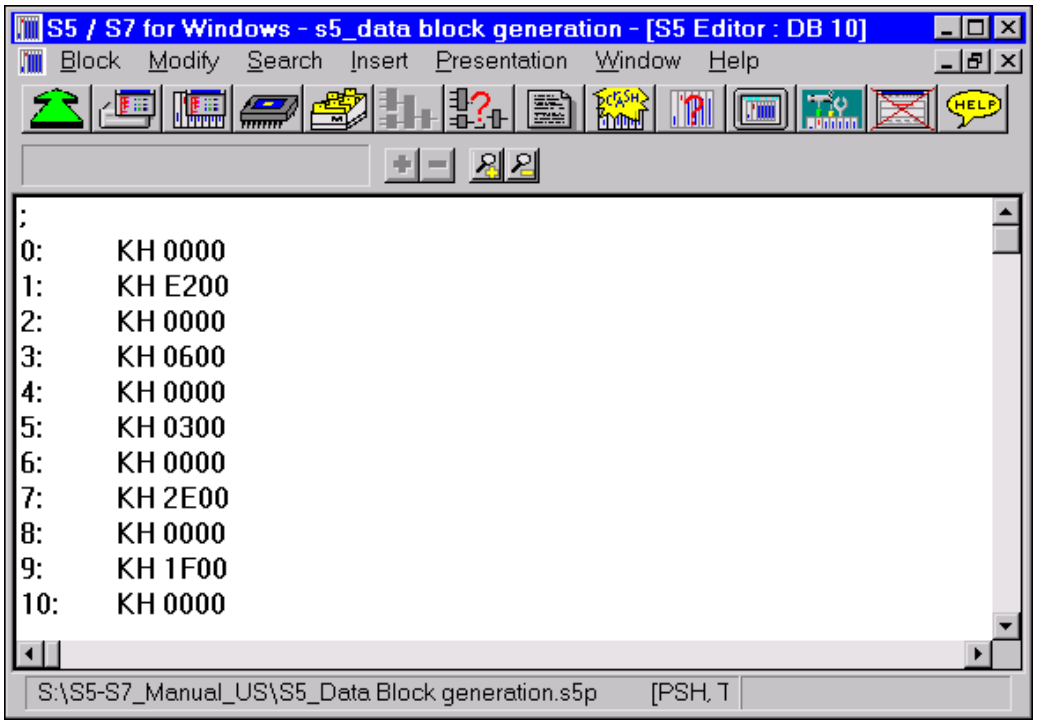


The data block is generated in the internal data block area of the PLC CPU.

Prior to the generate instruction (**G DBnn** or **G DXnn**) the number of data words must be defined (**L KF +xx**).

The maximum number of data words that can be generated depends on the CPU type (max KB 253 = 254 Data Words).

After executing the PLC program the data block is created. You may edit the value or change the data type format.



Function Block (FB) with Data Block (DB)

Example: (Maximum Value)

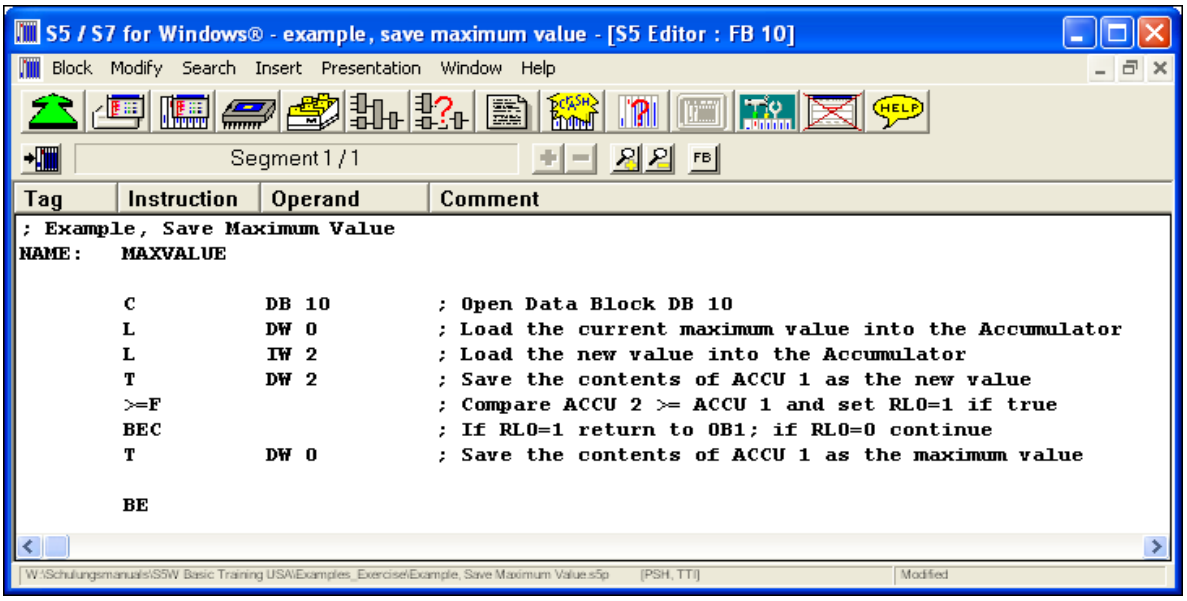
The maximum value entered into the PLC via an input module should be saved. Other PLC Blocks should have access to that value any time. The easiest way to accomplish this task is to save the “Maximum Value” in a Data Block.

The value coming from the input module (IW 2) is compared with the value stored in the Data Block (DB 10) (“Maximum Value”).

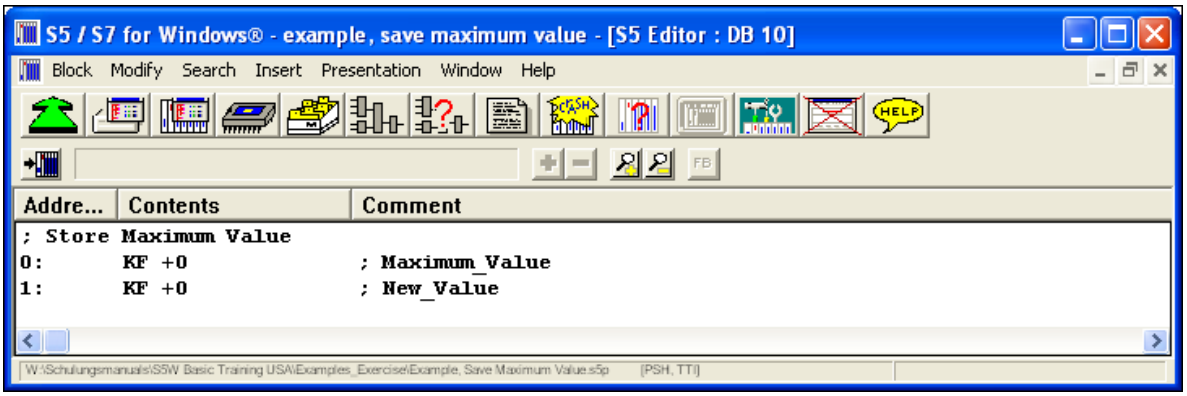
If the “New Value” is higher than the stored “Maximum Value” the “New Value” is saved by replacing the previous “Maximum Value”

The current “Maximum Value” and “New Value” should be monitored outside the Function Block (in OB1).

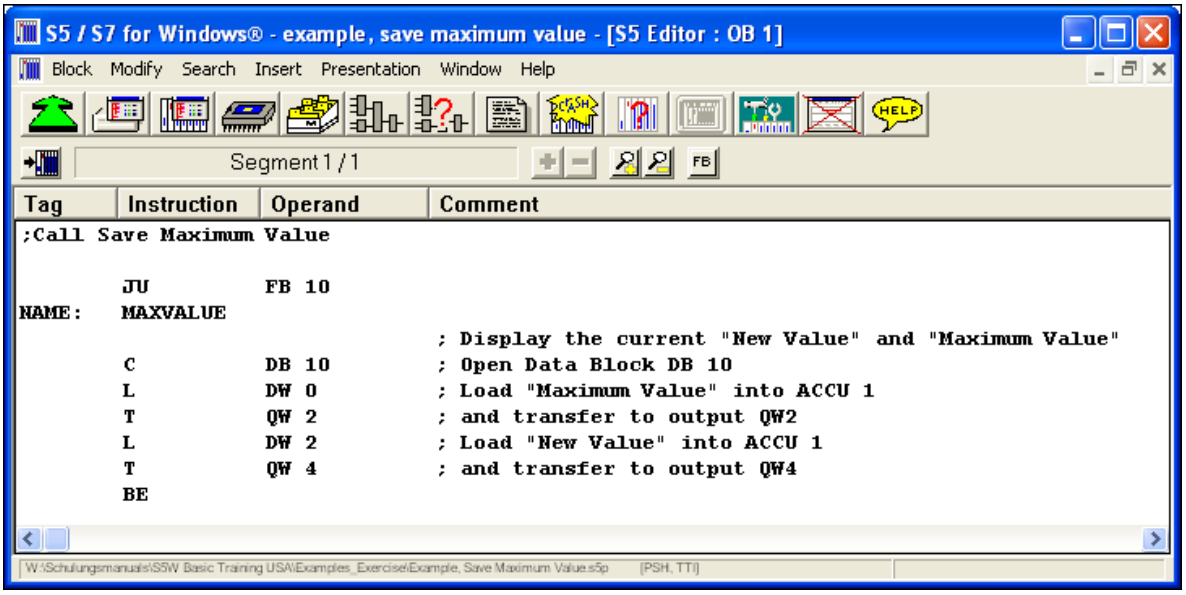
Function Block FB 10



Data Block DB 10



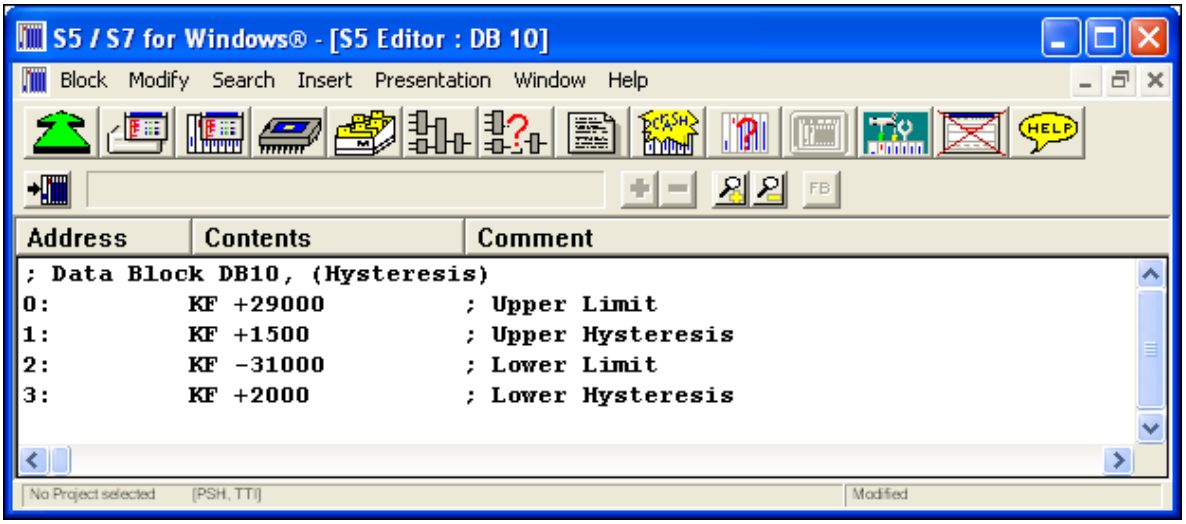
Organization Block OB 1



Practice Exercise 5–1; Hysteresis, Function Block with Data Block

A “Value” coming from an input module should be compared against an “Upper Limit” and a “Lower Limit”. If the “Value” exceeds the limits, lights should be turned on. The lights should be turned off if the “Value” is away from the limits by the defined offset (Hysteresis).

1. Start a new project.
2. Declare the following parameters in the Data Block DB10:



3. Make a new S5 Block (Function Block FB10) and program the logic in the Function Block.
4. Transfer of the program into the S5 TEST PLC.
5. Test the PLC program.