Process automation
Programmable Logic Controllers (PLCs) Programming – part 2

BS UNI studies, Fall semester 2024/2025

Octavian M. Machidon

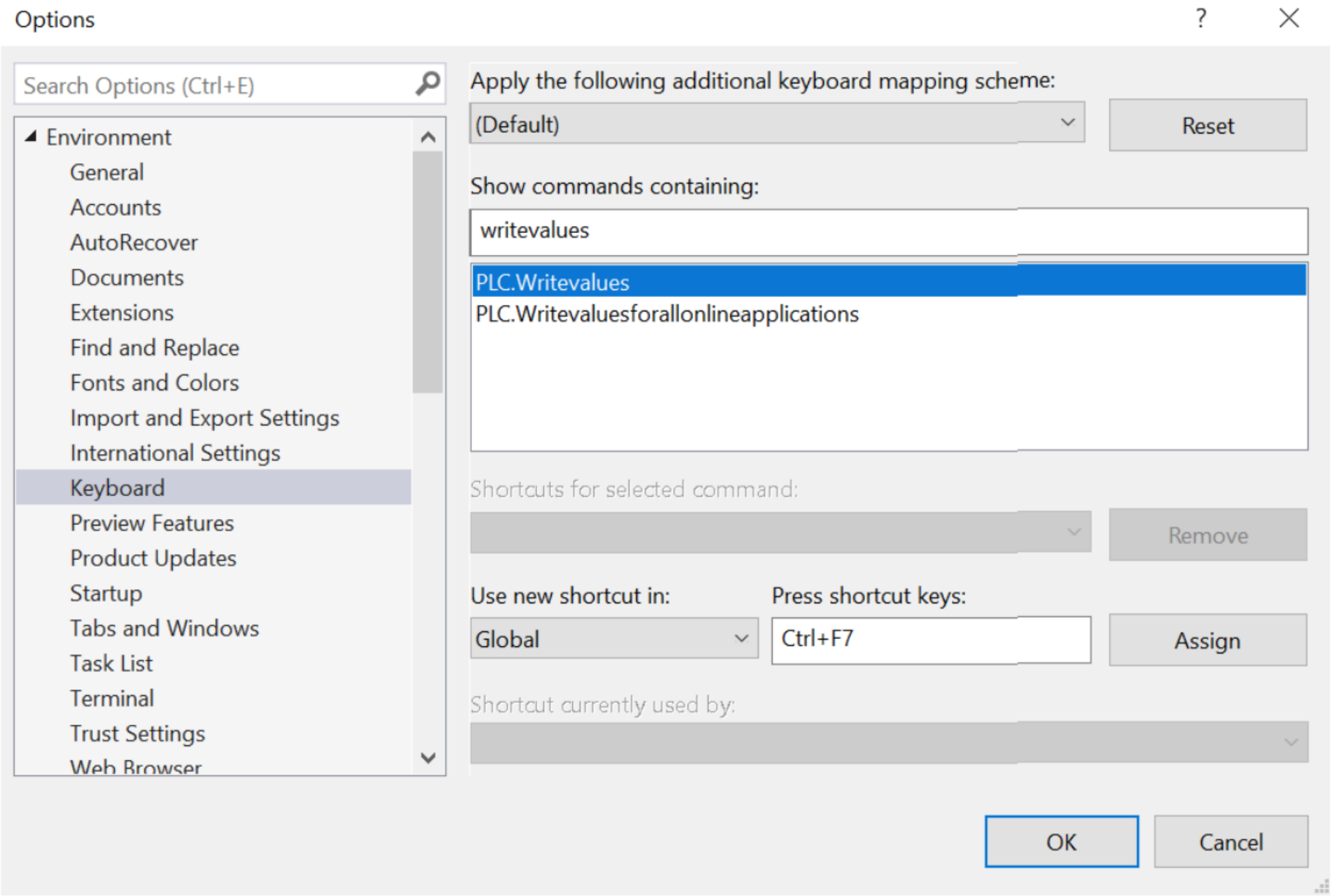
octavian.machidon@fri.uni-lj.si

Outline

- Function Block Diagrams (FBD)
- Sequential Function Charts (SFC)
- Instruction Lists (IL)
- Structured Text (ST)
- Programming Techniques

TwinCAT – shortcuts for ease of use

- Tools → Customize → Commands → Keyboard…



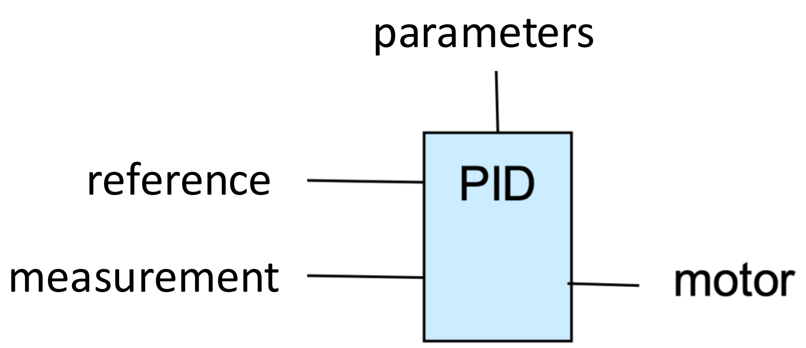
Function Block Diagram: Standard

- Function Block Diagram is one of the standard graphical languages
  - It visually displays the interconnection of functions and function blocks.
- It resembles electrical and block diagrams from analog and digital technologies
  - Each block has inputs and outputs.
  - The connections represent the flow of electrical current in proper circuits.
- Blocks usually represent combinatorial functions (decision logic) but can also have memory (sequential logic)
  - Some blocks are combinational, meaning they make decisions based on inputs. Others include memory, allowing for sequential operations.
- The standard also allows for feedback connections
  - The standard prescribes defining the order of execution for blocks with feedback but does not prescribe how.

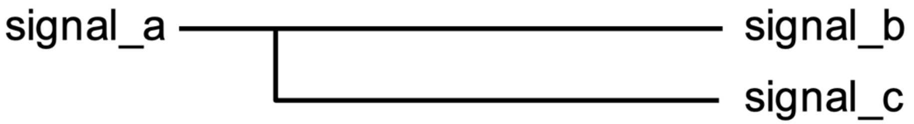
Function Block Diagrams: standard

- **A function block** is defined by:
  - Interface:
    - Specifies the number and types of inputs and outputs.
  - Black box functionality:
    - The operation of the block is described graphically, with a table, formula, or description.
- Connection Rules:
  - Every signal is connected to exactly one source.
  - The source, sink, and connection must all be of the same data type.

Example:

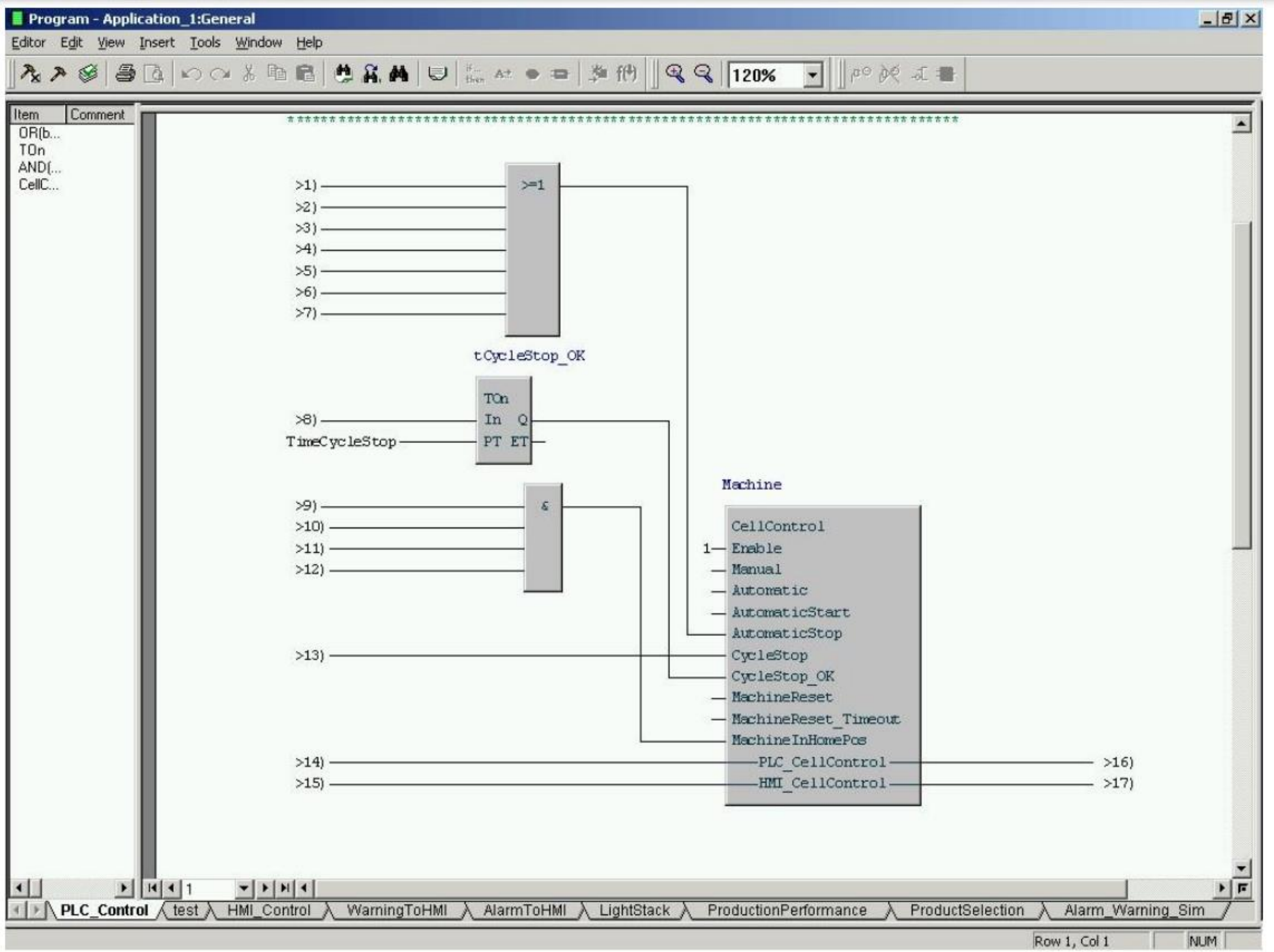


Example:



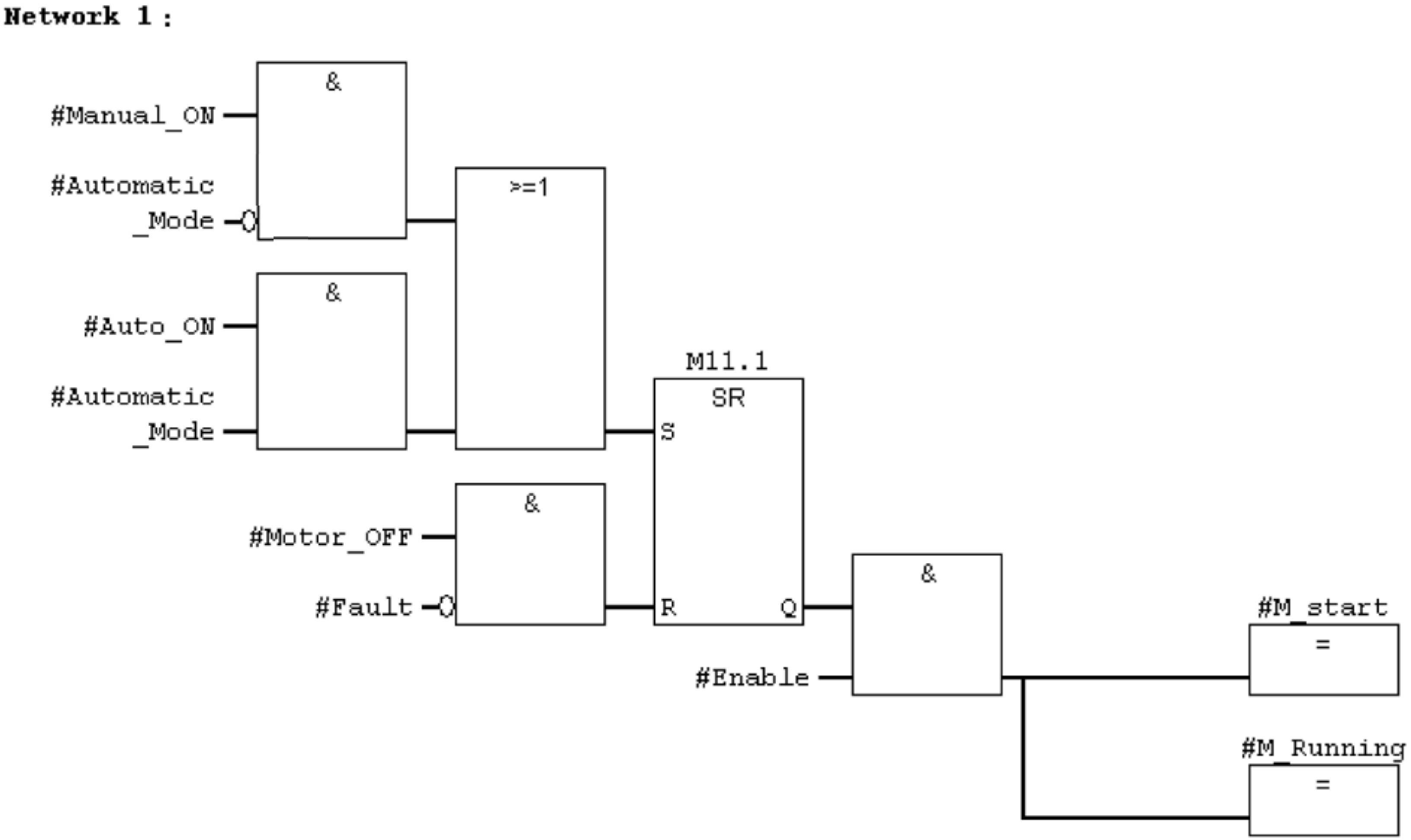
Function Block Diagrams: program

- Program Execution:
  - The program is executed in a specific order:
    - From top to bottom
    - From left to right
    - Exceptions are backward connections (e.g., feedback loops)
- Example: ABB development environment



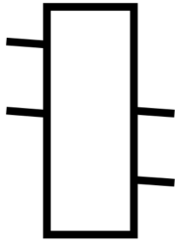
Function Block Diagrams: program

- Example: Motor control in Siemens TIA Portal.

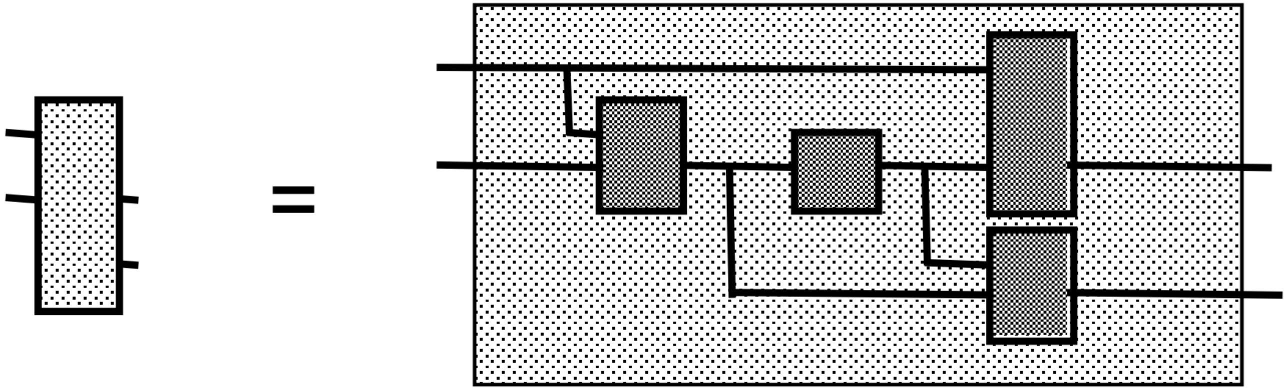


Function Block Diagrams: program

- Elementary block:
  - Microcode, Assembler, Structured Text (ST)

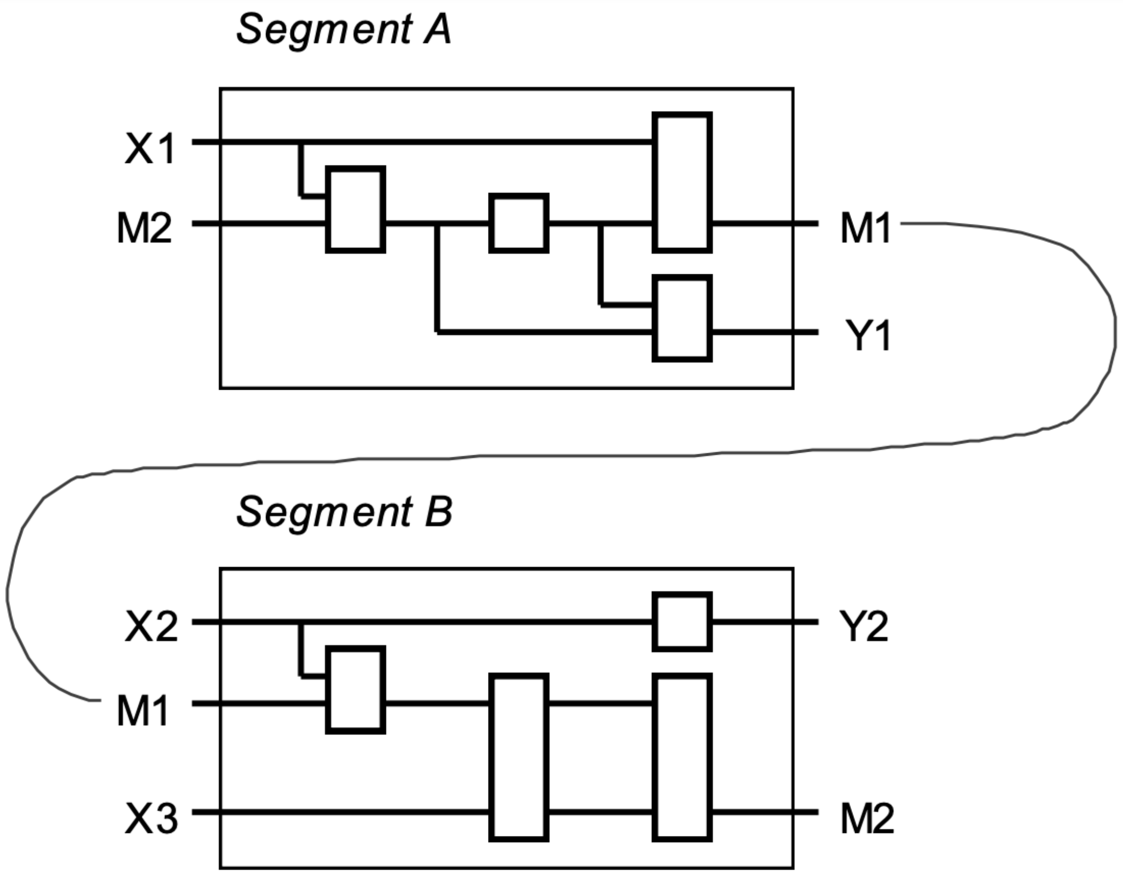


- Composite block:
  - Connects multiple elementary blocks into a whole.
  - Function block diagram (FBD) where multiple functions work together.



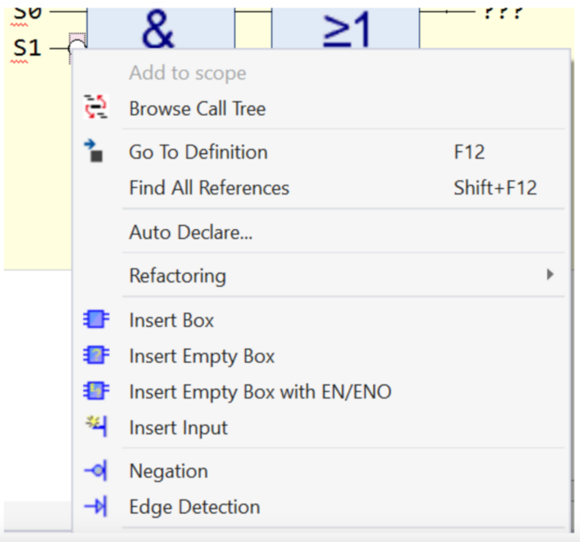
Function Block Diagrams: program

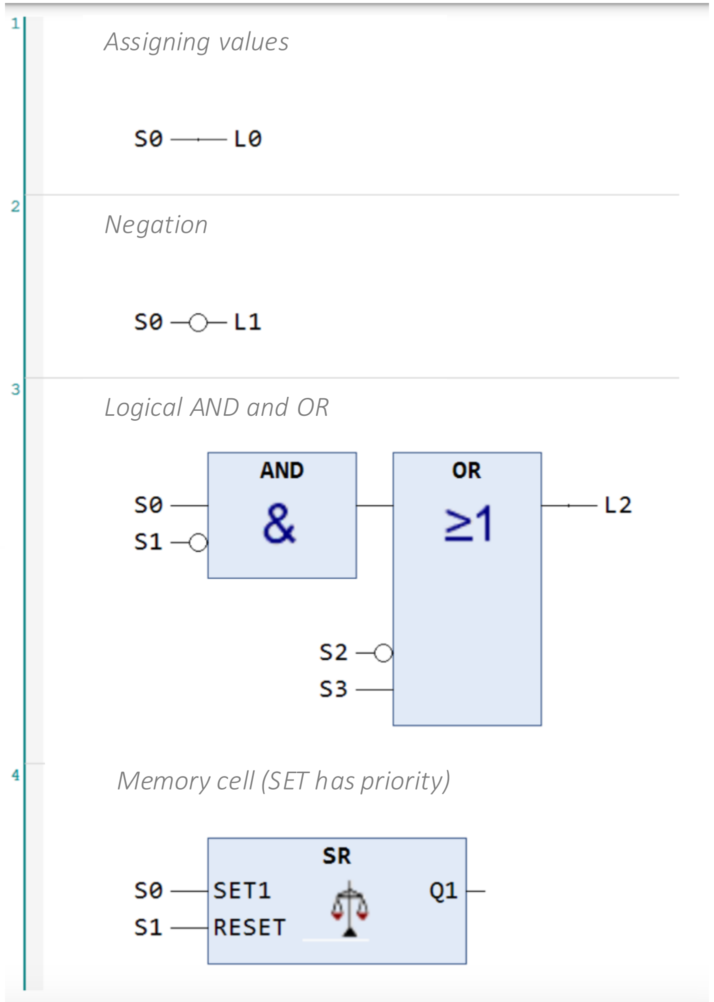
- Segmentation:
  - For better clarity, the functional plan is divided into several segments.
  - Inside each segment, the connections are represented graphically.
  - Temporary variables connect the segments to each other.



Function Block Diagrams: TwinCAT

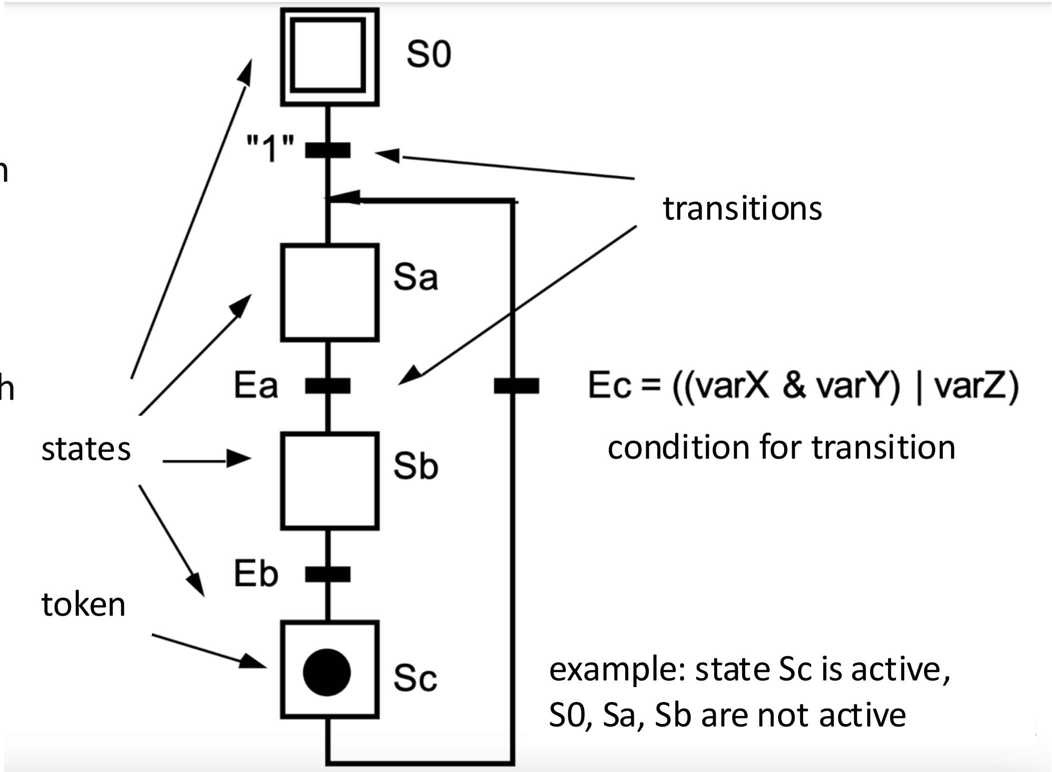
- Key Concepts:
  - Rungs (network):
  - The concept of “rungs” (networks) is borrowed from ladder diagrams.
  - The program is executed sequentially, one “rung” after another, from left to right.
- Best Practices:
  - Input Definition:
    - All inputs should be defined to avoid unwanted behavior.
  - Output Handling:
    - Each output (denoted as %Q) should only appear once in the program for clarity and control.
- Example: Basic Operations:
  - Negation can be accessed via the context menu (right-click) in the program.
- Switching Between Views:
  - Switching from LD to FBD: You can switch between Ladder Diagram (LD), Function Block Diagram (FBD), and Instruction List (IL) views through:
  - Extensions → FBD/LD/IL → View.





Sequential Function Chart (SFC): standard

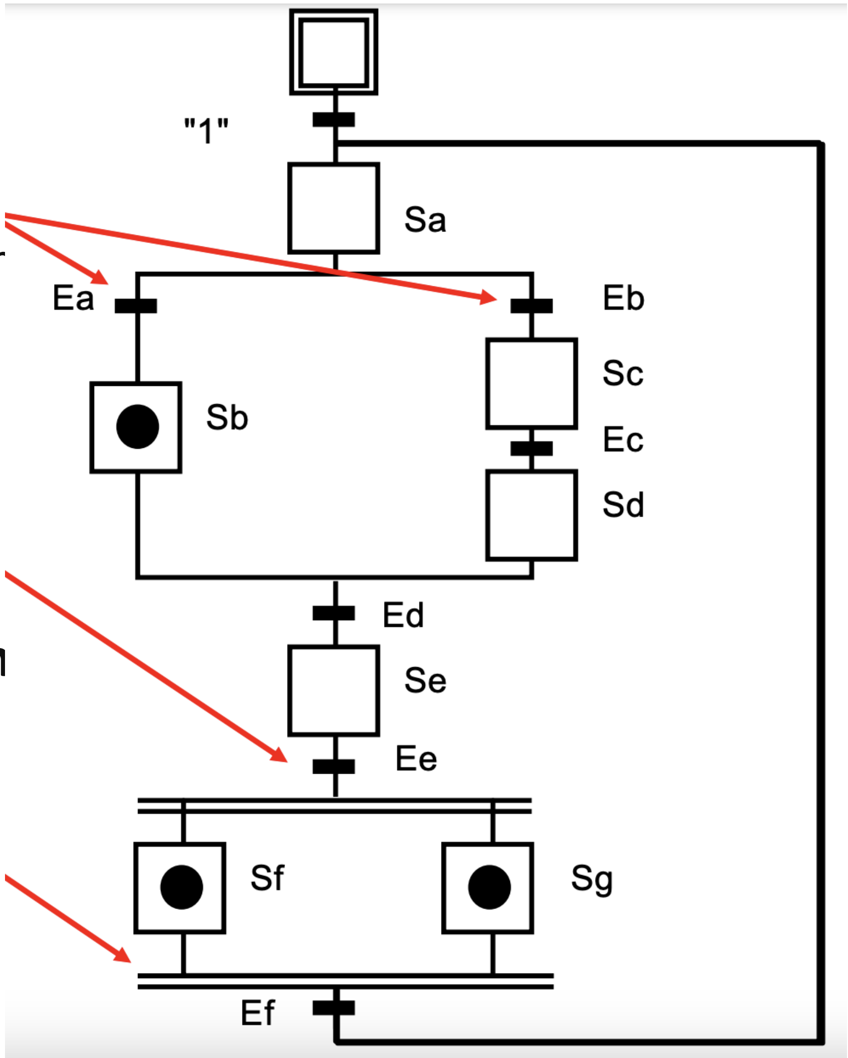
- The SFC is a method to represent the **sequence of operations** and interactions between parallel processes.
  - **Sequential operations:** It describes how various processes follow a sequence of operations.
  - **Parallel interactions:** It also illustrates how processes interact when multiple operations occur simultaneously.
  - **Mathematical basis:** The mathematical foundation of SFC is based on **Petri nets**, which are used to model and analyze systems where parallelism, concurrency, and synchronization are essential.
  - **States and Transitions:** In SFC, the system states are connected with **transitions** that move the process from one state to another.



- **Token:**
  - **Active State:** A state is considered active if it holds a **token**.
  - **Transition:** The token leaves the current state when the transition condition is met. This means the system moves to the next state based on specific conditions.
  - **One transition at a time:** Only one transition can occur at any given time.
  - **Initial token placement:** At the beginning of the program, the token is placed in the **initial state** to start the process.

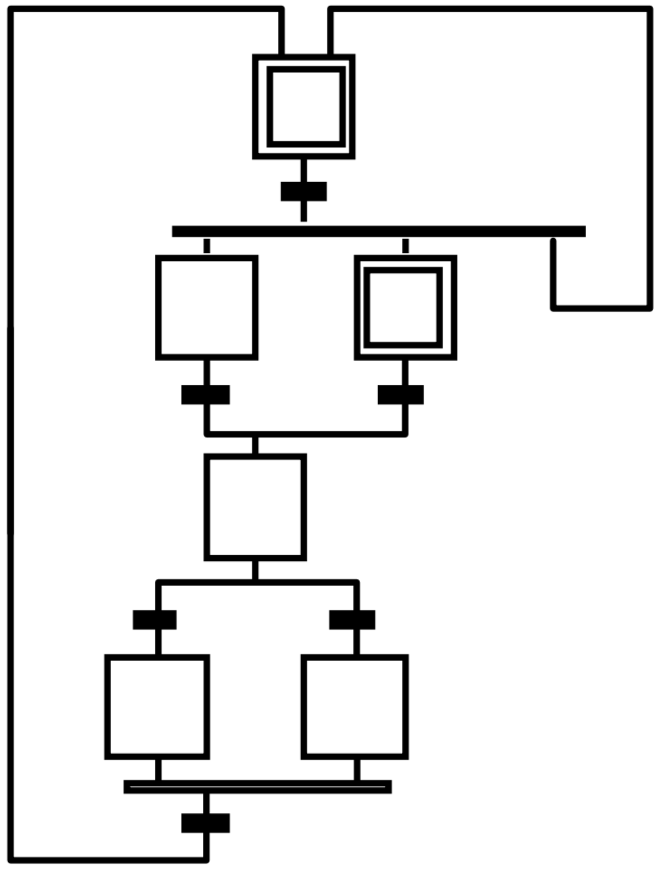
Sequential Function Chart: standard

- **Program Execution:**
  - **Token traverses the first active transition:**
    - The **token** moves through the **first active transition**. If both transitions **Ea** and **Eb** are active, the system will either follow a set priority or randomly choose between the two transitions.
  - **When condition Ee is met, the token is split across two states:**
    - Upon the fulfillment of the **Ee condition**, the token splits into two parts, moving into both connected states.
  - **When all divided tokens are present, and condition Ef is met, the token continues as one:**
    - Once the tokens reach all relevant states and **Ef condition** is satisfied, the tokens merge back into one, continuing in the flow.



Sequential Function Chart: standard

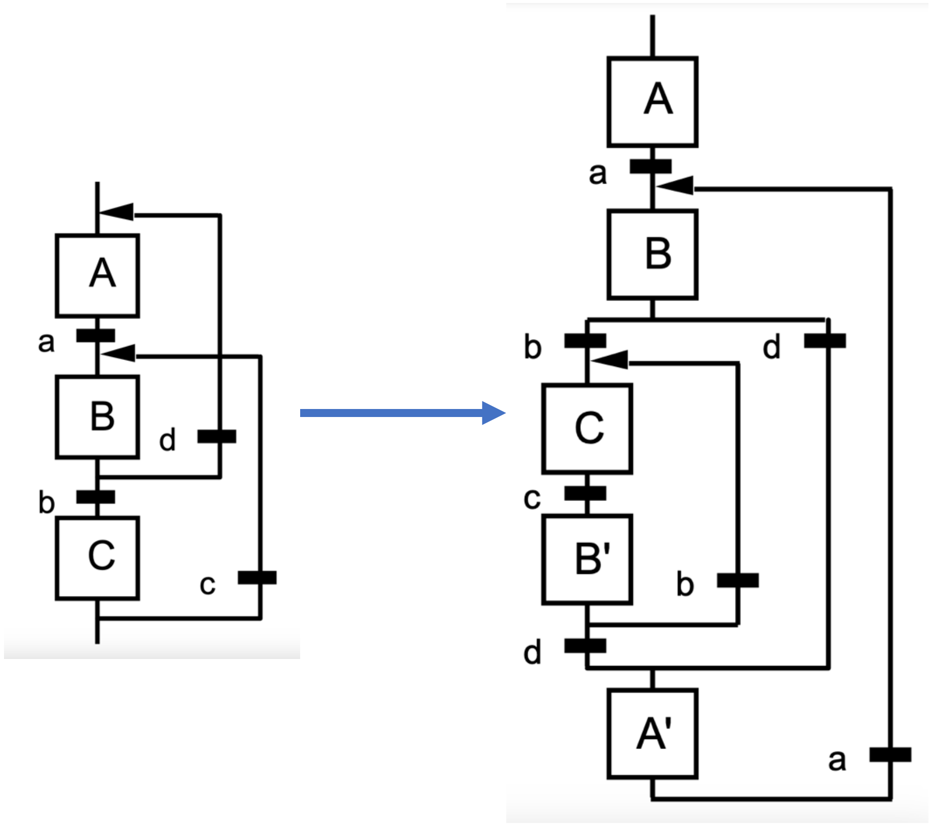
- Dangers of Complex Diagrams
  - Deadlock
  - Uncontrolled Handling of Tokens



- Solutions
  - Editor Restrictions
  - Editor Functions

Sequential Function Chart: standard

- Writing the diagram into a more transparent, structured form
  - Use duplication of states
- Work with exceptional events
  - Interlock
    - If the given condition is met, the actions in the state are interrupted
    - A transition to a new state is possible
  - Control errors
    - When an error occurs, transition to the next state is not possible
    - The automaton stops



Sequential Function Chart: comparison

Function Block Diagram, FBD:

- Continuous control, regulation

Sequential Function Chart, SFC:

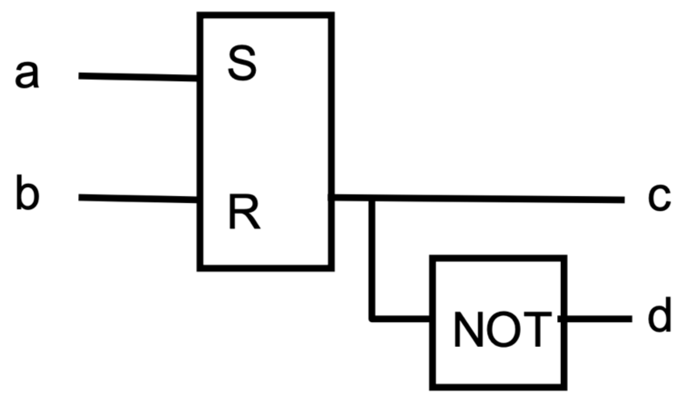
- Stepwise/sequential control, management

Often, the best choice is a combination of both, so communication between them must be possible: integration at the functional block level.

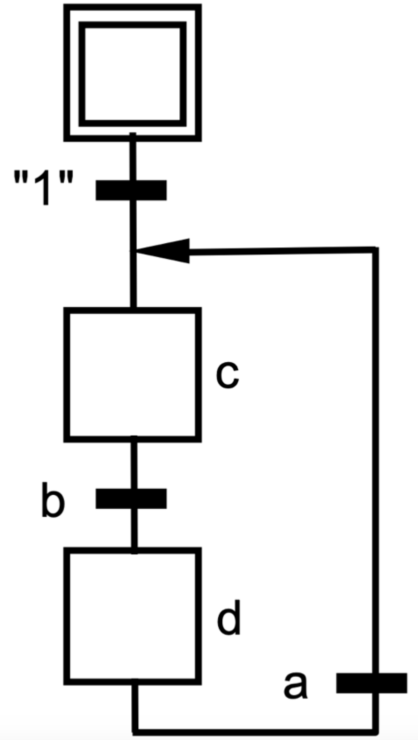
Sequential Function Chart: comparison

Example:

Functional Block Diagram



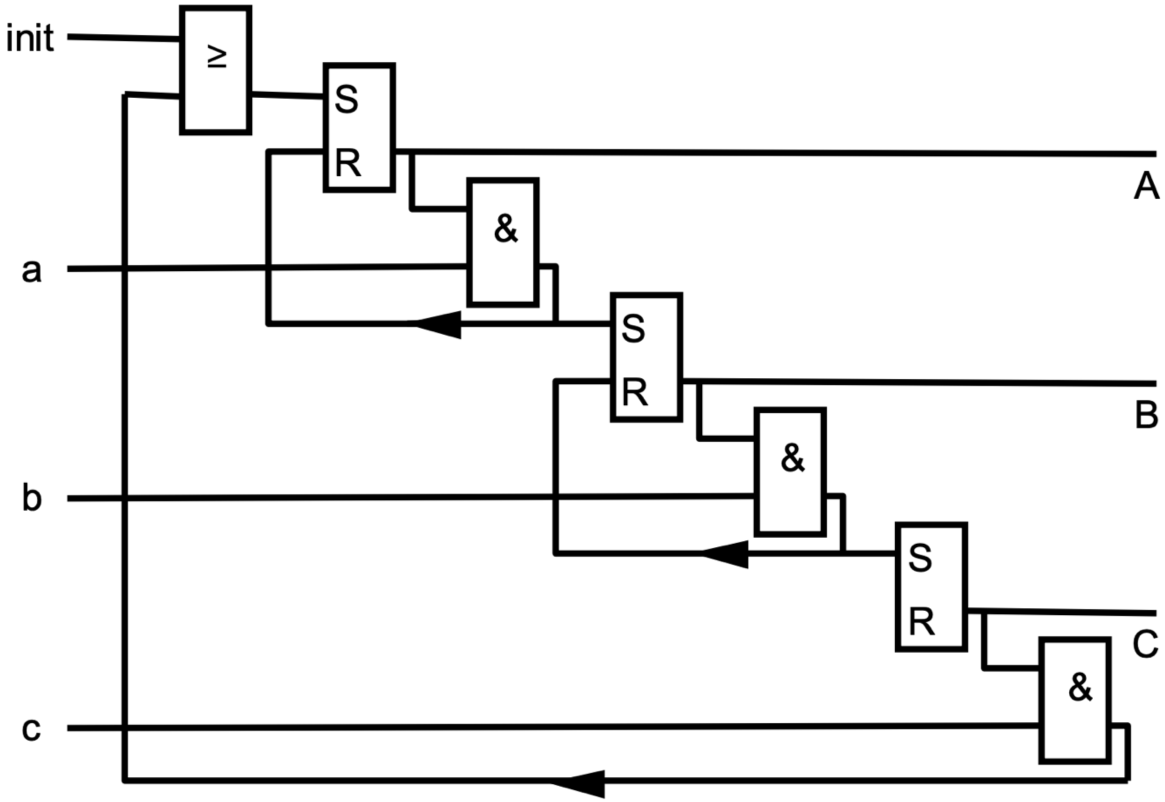
Sequential Function Chart



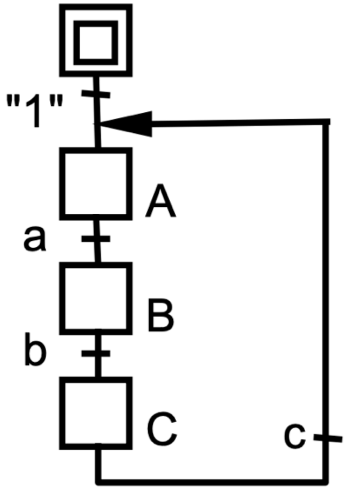
Sequential Function Chart: comparison

Example:

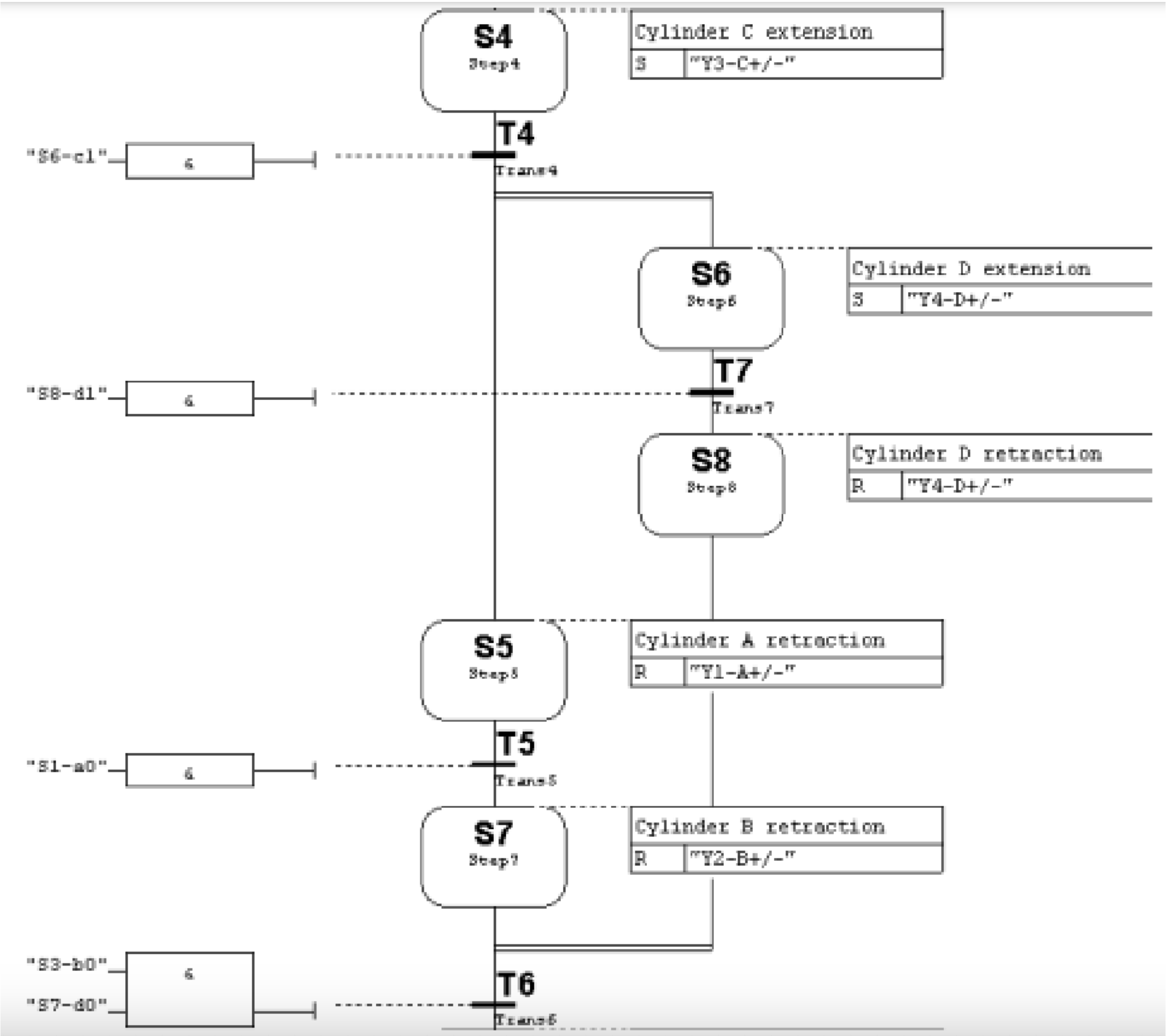
Functional Block Diagram



Sequential Function Chart



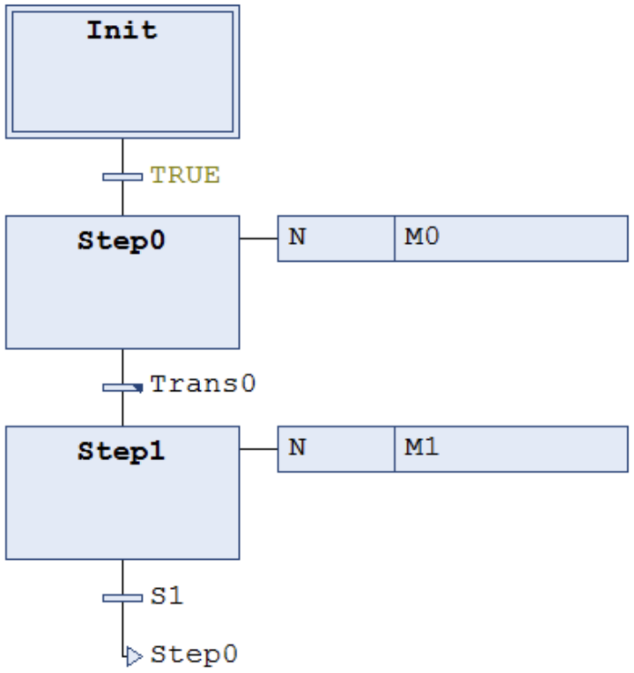
Sequential Function Chart: Siemens Graph example

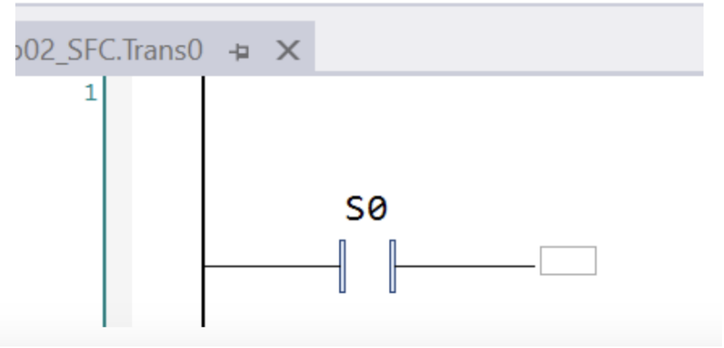


SFC: TwinCAT

- SFC can only be used in a program or function block, not in a function.
- Conditions for **transitions** (transition) can be written in any IEC language and included as calls or written directly **inline** in the ST language.

- **States**
  - Perform **actions** when the state is active:
    - by the IEC standard – action association
    - extension of the standard – step main action
    - On entry (extension – step entry action)
    - On exit (extension – step exit action)





SFC: qualifiers for actions in a state (IEC standard)

Qualifier	Name	Meaning
N	Non-stored	The action is active as long as the state is active.
R0	Overriding reset	The action is reset, meaning it is deactivated.
S0	Set/stored	The action is executed as soon as the state becomes active and continues executing even when the state is no longer active—until the action is reset.
L	Time-limited	The action begins executing as soon as the state is active and continues for the entire time the state is active or until the timer runs out.
D	Time-delayed	The action begins executing after a timer expires upon entering the state and continues as long as the state is active.
P	Pulse	The action is executed exactly twice: once when the state becomes active and once in the following program cycle.
SD	Stored and time-delayed	The action begins executing after a timer expires upon entering the state and continues until reset.
DS	Delayed and stored	The action begins after the timer expires and continues executing as long as the state remains active. It is active until reset.
SL	Stored and time-limited	The action begins as soon as the state becomes active and continues until the timer runs out or a reset occurs.

Instruction List: standard

- **Low-Level Programming Language**
  - **Similar to Assembly Language:**
    - The instruction list language is comparable to assembly language, which operates at a very low level, close to the hardware.
  - **User-Unfriendly:**
    - The code is unstructured, making it hard to follow or maintain.
    - It has weak semantics, meaning the meaning and behavior of the commands can be less clear compared to higher-level languages.
    - It is dependent on the specific programmable logic controller (PLC), which limits portability.
  - **Obsolescence:**
    - In the 2012 third edition of the **IEC 61131-3** standard, instruction list (IL) was considered obsolete. The argument presented by the committee was that assembly language is no longer suitable for modern development environments.
- **Majority Opinion**
  - **Basic Language for PLCs:** IL was historically considered the base language that should be supported by all programmable logic controllers.
  - **Undefined Standards:** The standard for the basic language (IL) has not been clearly defined, which leaves some ambiguity in its implementation.
- **Intended for Experienced Programmers**
  - **Creating Efficient Code:** It is targeted at experienced programmers for writing time- and space-efficient code.
  - **Translation of Higher-Level Languages:** All higher-level programming languages should theoretically be translatable into this language, although the standard does not mandate this.

Instruction List: standard

- Each instruction begins on a new line and contains the following:
  - **Label:**
    - Placed at the start of the line, ending with a colon (:).
  - **Operation Code (Operator / Mnemonic):**
    - The operation or instruction to be performed, such as arithmetic or logical operations.
  - **Operands:**
    - Operands follow the operation code and are separated by commas. These are the inputs or targets of the instruction.
  - **Comment:**
    - A comment can be placed at the end of the line to describe the instruction, which is helpful for documentation and understanding the code.
- **Additional Notes:**
  - Blank lines are allowed, or lines with just parentheses can also be used to enhance readability or structure in the instruction list.

Instruction List: standard

- **21 basic commands** are listed.
- The results of these operations are stored in the **RLO (Result of Logic Operation) register**.
- **Modifiers:**
  - **N** – Negation of the result.
  - **C** – Conditional execution.
  - **(** – Delayed result.

Nr.	Operator	Modifier	Operand	Definition
1	LD	N	Note 1	Sets the actual result of the operand
2	ST	N	Note 1	Stores the actual result in the operand address
3	S	Note 2	BOOL	Set Boolean operator to 1
	R	Note 2	BOOL	Reset Boolean operator to 0
4	AND	N, (	BOOL	Boolean AND
5	&	N, (	BOOL	Boolean AND
6	OR	N, (	BOOL	Boolean OR
7	XOR	N, (	BOOL	Boolean Exclusive-OR
8	ADD	(	Note 1	Addition
9	SUB	(	Note 1	Subtraction
10	MUL	(	Note. 1	Multiplication
11	DIV	(	Note 1	Division
12	GT	(	Note 1	Comparison: >
13	GE	(	Note 1	Comparison: >=
14	EQ	(	Note 1	Comparison: =
15	NE	(	Note 1	Comparison: <>
16	LE	(	Note 1	Comparison: <=
17	LT	(	Note 1	Comparison: <
18	JMP	C,N	MARK	Jump to the Mark
19	CAL	C,N	NAME	Call function block (Note 3)
20	RET	C,N		Return to a function or a function block
21	)			Processing reset operations

Note 1: The operations must be either loaded or given with a type.
The actual result and the operand must have the same type.
Note 2: The operations are only executed when the value of the actual result is a Boolean 1.
Note 3: A list of arguments in parenthesis follow the name of the function block

Instruction List: standard

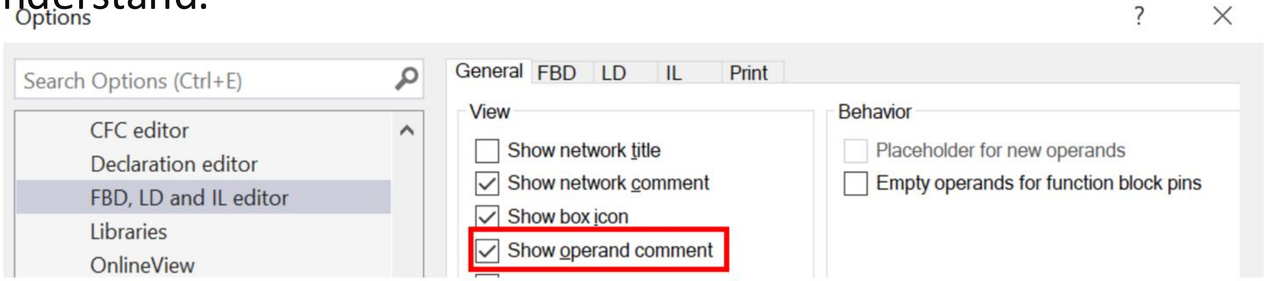
- Examples:

```
AND %IX1 (* Result := Result AND %IX1 *)

AND( %IX1
ORN %IX2
) (* Result := Result AND (%IX1 OR NOT %IX2) *)

LD 15
ST C10.PV
LD %IX10
ST C10.CU

CAL C10 (* Function call: CAL C10(CU:=%IX10, PV:=15) *)
```

Instruction List: TwinCAT

- **Enabling IL in TwinCAT**:
  - Navigate to: Tools → Options → TwinCAT → PLC Environment → FBD, LD and IL → IL
  - In the settings, check the box that says **“Enable IL”** to activate the Instruction List programming mode in TwinCAT.

- **Showing Operand Comments**:
  - If you want to display comments for each instruction line, go to:
    - General → Show operand comment
  - Check the option **“Show operand comment”** to display comments associated with operands, making the code easier to understand.



Instruction List: TwinCAT

- **Rung Concept**: The term “rung” refers to a section in a ladder logic diagram that executes specific operations, which TwinCAT has adopted from ladder diagrams.
- **Functional Block Example**: It shows how a functional block (FB) performs basic logical operations and how the FB is called from a program.

	Scope	Name	Address	Data type	Initialization	Comment	Attributes
1	VAR_INPUT	x1		BOOL			
2	VAR_INPUT	x2		BOOL			
3	VAR_OUTPUT	yNot		BOOL			
4	VAR_OUTPUT	yOr		BOOL			
5	VAR	yAnd		BOOL			

```
1
Basic logical operations in Instruction List (IL): Negation
LDN     x1        Load negated x1. RLO = NOT x1
ST      yNot      Store RLO into yNot

2
Logical AND
LD      x1        RLO = x1
AND     x2        RLO = RLO AND x2
ST      yAnd      Store yAnd = RLO

3
Logical OR
LD      x1        RLO = x1
OR      x2        RLO = RLO OR x2
ST      yOr       Store yOr = RLO
```

	Scope	Name	Address	Data type	Initialization	Comment	Attributes
1	VAR	S0		BOOL			
2	VAR	S1		BOOL			
3	VAR	S2		BOOL			
4	VAR	L0		BOOL			
5	VAR	L1		BOOL			
6	VAR	L2		BOOL			
7	VAR	logicalOperation		p03_IL_logOperation			

```
1
Conditional call of function block (only if S2 is false)
LD      S2                   Load S2 to RLO
JMPC    JUMP                 Conditional jump to JUMP (only if RLO is true, otherwise function call)
CAL     logicalOperation(    Call function "logicalOperation"
           x1:= S0,          Assign S0 to x1
           x2:= S1,          Assign S1 to x2
        yNot=> L0,           Assign result of yNot to L0
        yAnd=> L1,           Assign result of yAnd to L1
         yOr=> L2)           Assign result of yOr to L2

2
The jump occurs at the label (JUMP), which is assigned to a specific jump point in the program.
JUMP:
```

Structured text: standard

- A language similar to Pascal
- Suitable for complex data processing
- Expressions define values based on variable and constant values
- It is necessary to use the required data types
  - Conversion between types with functions
  - Example: REAL_TO_INT(…)
- An expression is composed of operators and operands
  - Calculations follow operator precedence
  - If precedence is equal, evaluation is from left to right
  - Example: X := (A + B - C) * ABS(D);

Nr.	Operacija	Oznaka	Prioriteta
1	Brackets	(expression)	Visoka
2	Function call	FunName(arguments) e.g., LN(A), MAX(X,Y)	
3	Exponentiation	EXPT	
4	Sign	-	
5	Negation	NOT	
6	Multiplication	*	
7	Division	/	
8	Modulo	MOD	
9	Addition	+	
10	Subtraction	-	
11	Comparison	<, >, <=, >=	
12	Equality	=	
13	Inequality	<>	
14	Logical AND	AND, AND_THEN	
15	Exclusive OR (XOR)	XOR	
16	Logical OR	OR, OR_THEN	Nizka

Structured text: standard - statements

Statement	Example
Assignment	`CV := CV + 1;` `C := SIN(X);`

Function block call	`CMD_TIMER(IN := %IX5,`
	`          PT := T#300ms);`
	`A := CMD_TIMER.Q;`

Exit from function or function block	`RETURN;`

IF statement	`IF D < 0.0 THEN`
	`       NROOTS := 0;`
	`ELSIF D = 0.0 THEN`
	`       NROOTS := 1;`
	`ELSE`
	`       NROOTS := 2;`
	`END_IF;`

CASE statement	`TW := BCD_TO_INT(THUMBWHEEL); CASE`
	`TW OF`
	`    1: DISPLAY := OVEN_TEMP;`
	`    2,3: DISPLAY := MOTOR_SPEED;`
	`ELSE`
	`    DISPLAY := 0;`
	`END_CASE;`

Statement	Example

FOR loop	`J := 101;`
	`FOR I := 1 TO 100 BY 2 DO`
	`     IF WORDS[I] = 'KEY' THEN`
	`          J := I;`
	`          EXIT;`
	`     END_IF;`
	`END_FOR;`

WHILE loop	`J := 1;`
	`WHILE J <= 100 DO`
	`     J := J + 2;`
	`END_WHILE;`

REPEAT loop	`J := -1;`
	`REPEAT`
	`     J := J + 2;`
	`UNTIL J = 101`
	`END_REPEAT;`

EXIT	`EXIT;`
Empty statement	`;`

Structured text: examples

```
// IF statement
IF switch = TRUE THEN
    light := TRUE;
END_IF;

// If the condition is false, the light does not change state, i.e., the state is retained. Explicitly written:
IF switch = TRUE THEN
    light := TRUE;
ELSE
    light := light;
END_IF;

// Is this what we wanted? Or should the light turn off when the switch is turned off?
IF switch = TRUE THEN
    light := TRUE;
ELSE
    light := FALSE;
END_IF;

// Do we even need a conditional statement for this? No...
light := switch;
```

Structured text: examples

```
// If possible, replace conditional statements with logical expressions
IF switch THEN
    IF button1 THEN
        light := TRUE;
    ELSE
        light := FALSE;
    END_IF
ELSE
    light := button2;
END_IF

// Same as:
light := (switch AND button1) OR (NOT switch AND button2);

// Positive edge detection on the button
IF button AND NOT oldButtonState THEN
    // On the positive edge, invert the light state
    light := NOT light;
    // We skip ELSE to avoid memorization issues
END_IF

// Update the previous state of the button
oldButtonState := button;
```

Structured text: examples

- **Functional block for controlling a motor with direction switching**
  - Block to prevent rapid switching of the rotation direction
  - Protecting relays (specific to educational models)

	Scope	Name	Address	Data type	Initialization	Comment	Attributes
1	VAR	**timerForward**		TOF			
4	VAR	**blockBackward**		BOOL			
2	VAR	**blockForward**		BOOL			
3	VAR	**timerBackward**		TOF			
5	VAR	**rotateForward**		BOOL			
6	VAR	**rotateBackward**		BOOL			
7	VAR_INPUT	**forward**		BOOL			
9	VAR_INPUT	**blockTime**		TIME			
8	VAR_INPUT	**backward**		BOOL			
10	VAR_OUTPUT	**movement**		BOOL			
11	VAR_OUTPUT	**direction**		BOOL			

```
// Call of the timer function block
timerForward(IN:=forward, PT:=blockTime, Q=>blockForward);
timerBackward(IN:=backward, PT:=blockTime, Q=>blockBackward);

// Calculation of whether the motor can rotate forward or backward
rotateForward := forward AND NOT blockForward;
rotateBackward := backward AND NOT blockBackward;

// Motor movement output
movement := rotateForward OR rotateBackward;

// Output direction - we want to save the relay for the direction,
// so we will remember the state of the direction.
IF rotateForward THEN
    direction := TRUE;
ELSIF rotateBackward THEN
    direction := FALSE;
ELSE
    direction := direction; // Retain current state
END_IF

// Shorter and simpler:
IF movement THEN
    direction := rotateBackward;
END_IF
```

Programming Techniques: Introduction of States

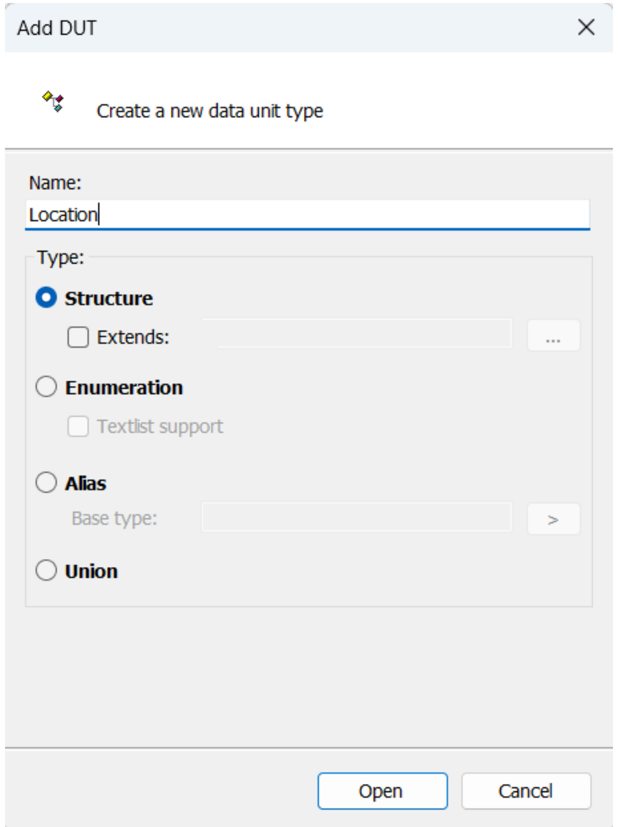
- **Reasons**
  - Certain parts of the program can only be executed under specific conditions
  - The need for locking rungs or parts of the program code
- **Dividing the program into logical states**
  - States and transitions between them must be clearly defined in both manual and automatic modes
    - States are determined based on system actions and values of measurement systems
  - Easier programming of complex systems
  - Easier reverse engineering
    - Code for each state is simpler
    - Conditions for transitions between states are much more apparent
    - Every programmer writes in their own style
- **Advantages**
  - Reducing system startup errors in the program by 85%
    - Mostly due to simpler conditions for locking rungs
    - In a typical ladder diagram, a large portion of the code is dedicated to locking rungs
      - 35% in process control (continuous processes, regulation)
      - 60% in sequential process

Programming Techniques: Introduction of States

- **Imitating concepts of the Sequential Function Chart (SFC) language:**
  - When the condition for the transition to a new state is met:
    - The corresponding variable (token) for the new state is activated (set).
    - The variable for the current state is deactivated (reset).
      - If multiple states can be active at the same time, care must be taken to deactivate all current states when transitioning to a new common state.
- **State Marking:**
  - Using bits: one bit corresponds to one state (known as "one hot encoding").
  - Numerical: using whole number variables (and comparators).
- **System Startup Logic:**
  - Identify the state in which the system was stopped.
    - Skipping states and not executing them sequentially can be very dangerous!
  - Set the system to its initial state or
  - Prevent its operation if it's not in the correct state, and raise an alarm (the easiest solution).

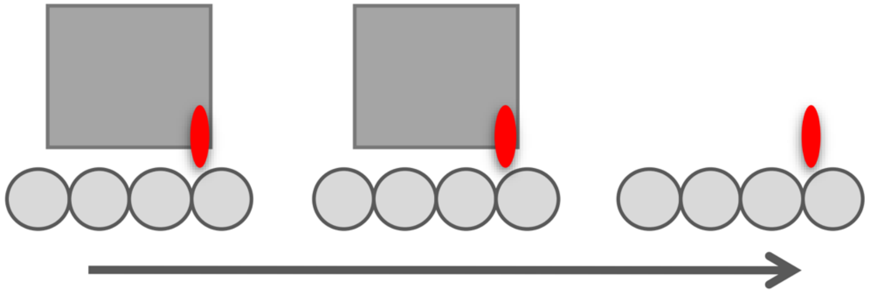
Programming Techniques: Material Tracking

- Creating a composite data type – structure (DUT, *data unit type*), which represents a logical image of a single workpiece/material:
  - Material data: barcode (ID), physical dimensions, defects, etc.
  - Target location
  - Processing instructions (recipe)
  - Functions at the current location: occupancy, movement, etc.
- **TwinCAT:**
  - PLC → <Project> → DUTs → Add → DUT...
  - We create a new **structure**
  - We can also use arrays (ARRAY) for help.



Programming Techniques: Material Tracking

- **Sequential Process**
- Each physical unit (location) also has its own logical representation.
- Each unit can perform only one task at a time.
  - Example: three conveyor belts
  - Simple communication: request, permission, action, confirmation, (alarm)



- **Overwriting Structures**
  - When conditions are met, the entire image is transferred from one location to another:
    - The material is physically in a new location.
    - Sensor image matches the future logical image (double check).
    - The transfer should NOT be linked to the front of a photocell.
- **Alarm activation** in case the logical and physical diagrams do not match after a set time.

Programming Techniques: Program Organization

1. **Reading sensors into structures**
   - Calibration of sensors, scaling of analog values, conversions **Triggering alarms**
2. **Managing triggered alarms**
   - Confirming and resetting alarms
3. **Preparing data for the human-machine interface**
   - Conversions, calculations, separate data structures (own FB) for better clarity
4. **Main program**
   - State transition automation
5. **Tracking material**
6. **Safety functions**
   - Protecting people and equipment
   - Due to safety independent of the main program
   - Blocking too fast direction changes in motor rotation
7. **Activation of execution systems**

Programming Techniques: Learning Models

Line with Two Devices and Pneumatic System:

- Definition of locations where the sensor is placed (photocell or limit switch); "virtual" locations (pusher, rotating table, entry onto the conveyor)
- Automaton for each location or executive element
- Dependencies (previous, next position)
- Rotating table (control of position, material on tables)
- Material tracking:
  - Barcode (ID)
  - Task/recipe
  - Input of values at the first location through IDE

Robotic hand:

- Hierarchy of automatons:
  - Automaton for each axis (rotation, lift, extension, grip)
  - Automaton connecting all four axes:
    - Move to location
    - Move to location and pick up
    - Move to location and place down
  - Automaton executing the "program" of movements
- Material tracking:
  - Barcode (ID)
  - Locations/positions of objects