efesotomasyon.com

SIMATIC Instructions

9

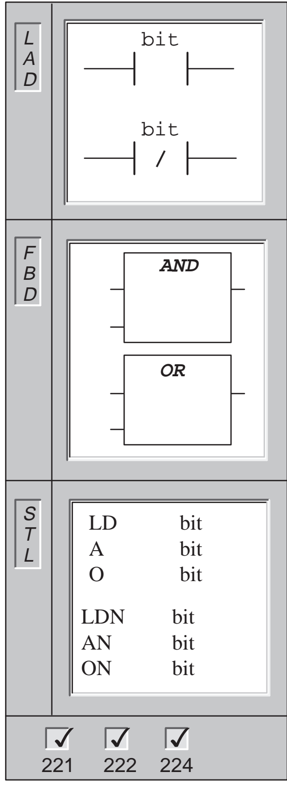
This chapter describes the SIMATIC instruction set for the S7-200.

Chapter Overview

Section	Description	Page
9.1	SIMATIC Bit Logic Instructions	9-2
9.2	SIMATIC Compare Instructions	9-10
9.3	SIMATIC Timer Instructions	9-15
9.4	SIMATIC Counter Instructions	9-23
9.5	SIMATIC High-Speed Counter Instructions	9-27
9.6	SIMATIC Pulse Output Instructions	9-49
9.7	SIMATIC Clock Instructions	9-70
9.8	SIMATIC Integer Math Instructions	9-72
9.9	SIMATIC Real Math Instructions	9-81
9.10	SIMATIC Move Instructions	9-99
9.11	SIMATIC Table Instructions	9-104
9.12	SIMATIC Logical Operations Instructions	9-110
9.13	SIMATIC Shift and Rotate Instructions	9-116
9.14	SIMATIC Conversion Instructions	9-126
9.15	SIMATIC Program Control Instructions	9-141
9.16	SIMATIC Interrupt and Communications Instructions	9-165
9.17	SIMATIC Logic Stack Instructions	9-192

9.1 SIMATIC Bit Logic Instructions

Standard Contacts



These instructions obtain the referenced value from the memory or process-image register if the data type is I or Q. You can use a maximum of seven inputs to both the AND and the OR boxes.

The **Normally Open** contact is closed (on) when the bit is equal to 1.

The **Normally Closed** contact is closed (on) when the bit is equal to 0.

In LAD, normally open and normally closed instructions are represented by contacts.

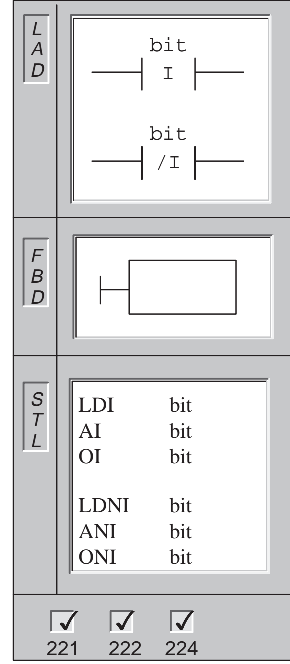
In FBD, normally open instructions are represented by AND/OR boxes. These instructions can be used to manipulate Boolean signals in the same manner as ladder contacts. Normally closed instructions are also represented by boxes. A normally closed instruction is constructed by placing the negation symbol on the stem of the input signal.

In STL, the Normally Open contact is represented by the **Load**, **And**, and **Or** instructions. These instructions Load, AND, or OR the bit value of the address bit to the top of the stack.

In STL, the Normally closed contact is represented by the **Load Not**, **And Not**, and **Or Not** instructions. These instructions Load, AND, or OR the logical Not of the bit value of the address bit to the top of the stack.

Inputs/Outputs	Operands	Data Types
bit (LAD, STL)	I, Q, M, SM, T, C, V, S, L	BOOL
Input (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL
Output (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Immediate Contacts



The immediate instruction obtains the physical input value when the instruction is executed, but the process-image register is not updated.

The **Normally Open Immediate** contact is closed (on) when the physical input point (bit) is 1.

The **Normally Closed Immediate** contact is closed (on) when the physical input point (bit) is 0.

In LAD, normally open and normally closed immediate instructions are represented by contacts.

In FBD, normally open immediate instructions are represented by the immediate indicator in front of the operand tic. The immediate indicator may not be present when power flow is used. The instruction can be used to manipulate physical signals in the same manner as ladder contacts.

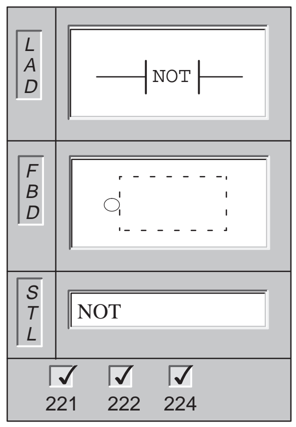
In FBD, normally closed immediate instructions are also represented by the immediate indicator and negation symbol in front of the operand tic. The immediate indicator cannot be present when power flow is used. The normally closed instruction is constructed by placing the negation symbol on the stem of the input signal.

In STL, the Normally Open Immediate contact is represented by the **Load Immediate**, **And Immediate**, and **Or Immediate** instructions. These instructions Load, AND, or OR the physical input value to the top of the stack immediately.

In STL, the Normally Closed Immediate contact is represented by the **Load Not Immediate**, **And Not Immediate**, and **Or Not Immediate** instructions. These instructions immediately Load, AND, or OR the logical Not of the value of the physical input point to the top of the stack.

Inputs/Outputs	Operands	Data Types
bit (LAD, STL)	I	BOOL
Input (FBD)	I	BOOL

Not



The **NOT** contact changes the state of power flow. When power flow reaches the Not contact, it stops. When power flow does not reach the Not contact, it supplies power flow.

In LAD, the **NOT** instruction is shown as a contact.

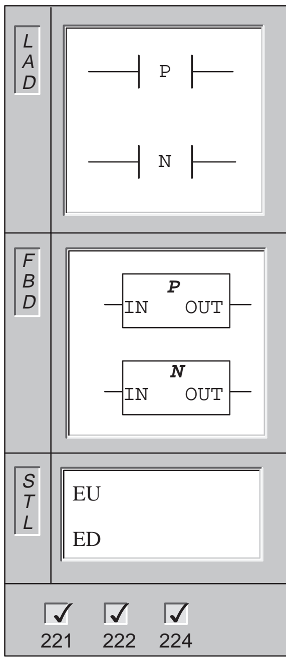
In FBD, the **NOT** instruction uses the graphical negation symbol with Boolean box inputs.

In STL, the **NOT** instruction changes the value on the top of the stack from 0 to 1, or from 1 to 0.

Operands: none

Data Types: None

Positive, Negative Transition



The **Positive Transition** contact allows power to flow for one scan for each off-to-on transition.

The **Negative Transition** contact allows power to flow for one scan for each on-to-off transition.

In LAD, the Positive and Negative Transition instructions are represented by contacts.

In FBD, the instructions are represented by the P and N boxes.

In STL, the Positive Transition contact is represented by the **Edge Up** instruction. Upon detection of a 0-to-1 transition in the value on the top of the stack, the top of the stack value is set to 1; otherwise, it is set to 0.

In STL, the Negative Transition contact is represented by the **Edge Down** instruction. Upon detection of a 1-to-0 transition in the value on the top of the stack, the top of the stack value is set to 1; otherwise, it is set to 0.

Inputs/Outputs	Operands	Data Types
IN (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL
OUT (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Contact Examples

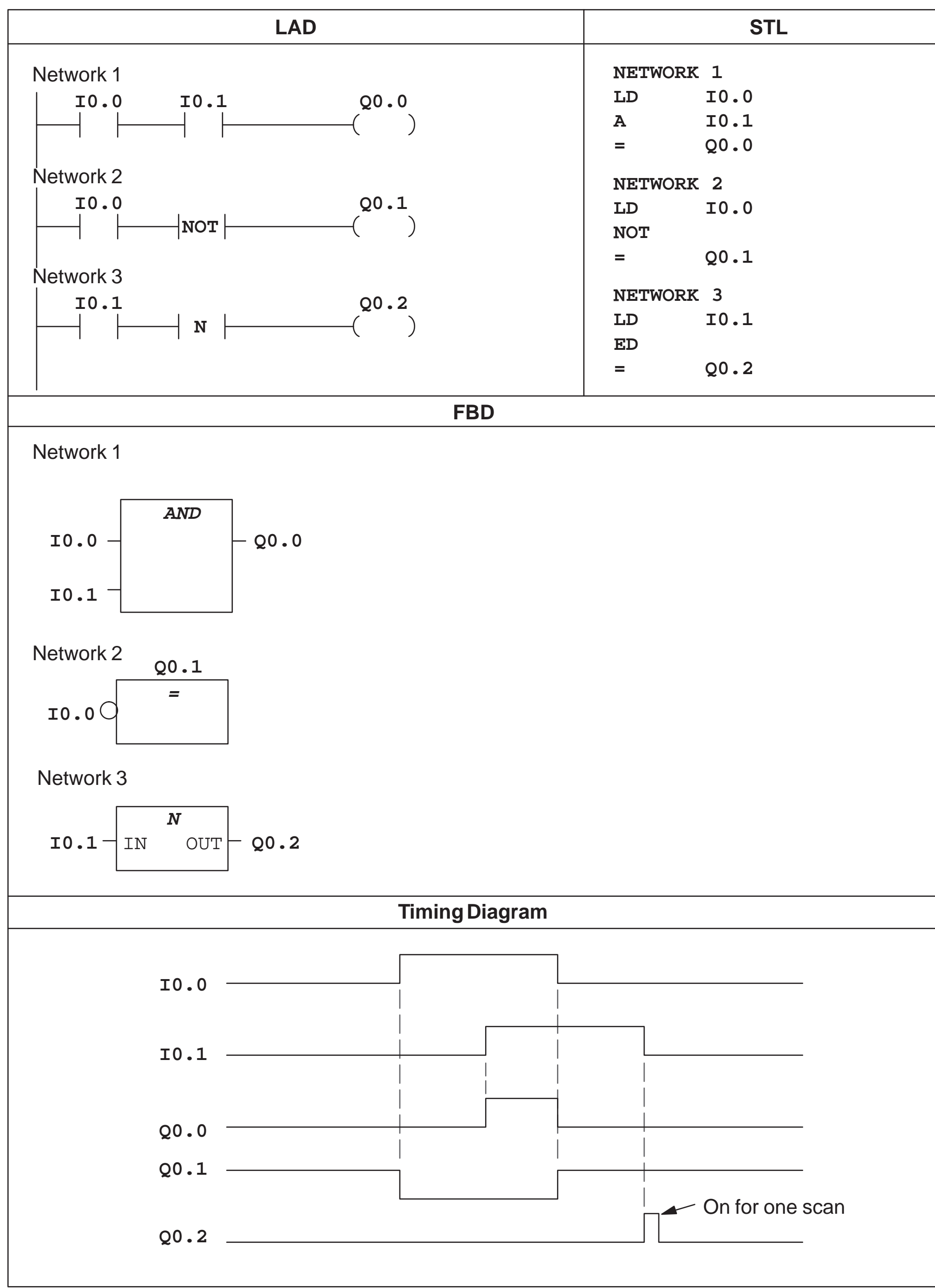
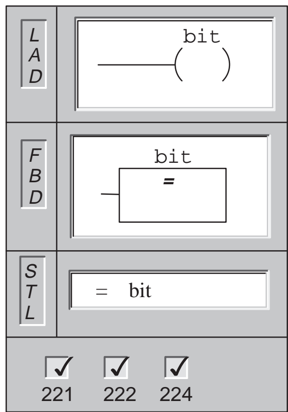


Figure 9-1 Examples of Boolean Contact Instructions for SIMATIC LAD, STL, and FBD

Output



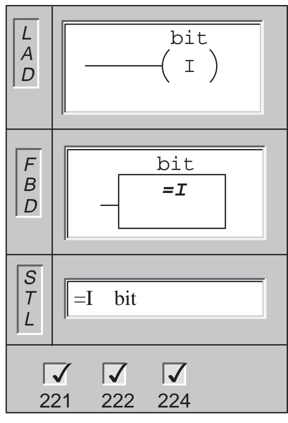
When the **Output** instruction is executed, the output bit in the process image register is turned on.

In LAD and FBD, when the output instruction is executed, the specified bit is set to equal to power flow.

In STL, the output instruction copies the top of the stack to the specified bit.

Inputs/Outputs	Operands	Data Types
bit	I, Q, M, SM, T, C, V, S, L	BOOL
Input (LAD)	Power Flow	BOOL
Input (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Output Immediate



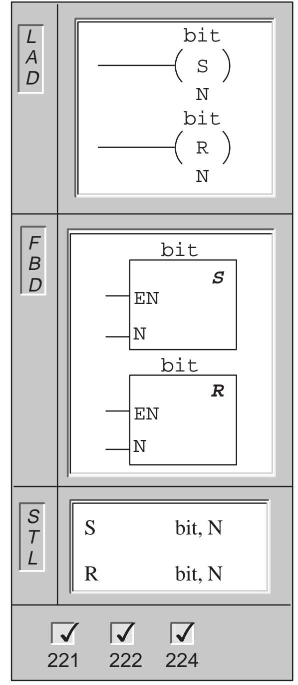
When the **Output Immediate** instruction is executed, the physical output point (bit or OUT) is set equal to power flow.

The "I" indicates an immediate reference; the new value is written to both the physical output and the corresponding process-image register location when the instruction is executed. This differs from the non-immediate references, which write the new value to the process-image register only.

In STL, the output immediate instruction copies the top of the stack to the specified physical output point (bit) immediately.

Inputs/Outputs	Operands	Data Types
bit	Q	BOOL
Input (LAD)	Power Flow	BOOL
Input (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Set, Reset (N Bits)



When the **Set** and **Reset** instructions are executed, the specified number of points (N) starting at the value specified by the bit or OUT parameter are set (turned on) or reset (turned off).

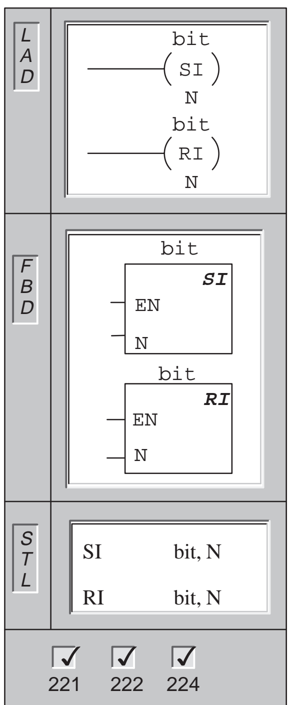
The range of points that can be set or reset is 1 to 255. When using the Reset instruction, if the bit is specified to be either a T- or C-bit, then either the timer or counter bit is reset and the timer/counter current value is cleared.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

Inputs/Outputs	Operands	Data Types
bit	I, Q, M, SM, T, C, V, S, L	BOOL
N	VB, IB, QB, MB, SMB, SB, LB, AC, Constant,*VD, *AC, *LD	BYTE

efesotomasyon.com

Set Immediate, Reset Immediate (N Bits)



When the **Set Immediate** and **Reset Immediate** instructions are executed, the specified number of physical output points (N) starting at the bit or OUT are immediately set (turned on) or immediately reset (turned off).

The range of points that can be set or reset is 1 to 128.

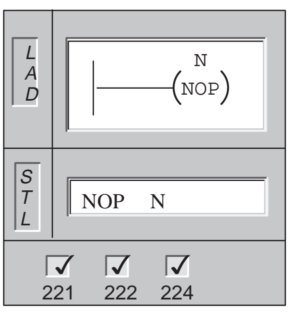
The "I" indicates an immediate reference; the new value is written to both the physical output point and the corresponding process-image register location when the instruction is executed. This differs from the non-immediate references, which write the new value to the process-image register only.

Error conditions that set ENO = 0:

SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

Inputs/Outputs	Operands	Data Types
bit	Q	BOOL
N	VB, IB, QB, MB, SMB, SB, LB, AC, Constant,*VD, *AC, *LD	BYTE

No Operation



The **No Operation** instruction has no effect on the user program execution. The operand N is a number from 0 to 255.

Operands: N: Constant (0 to 255)

Data Types: BYTE

Output Examples

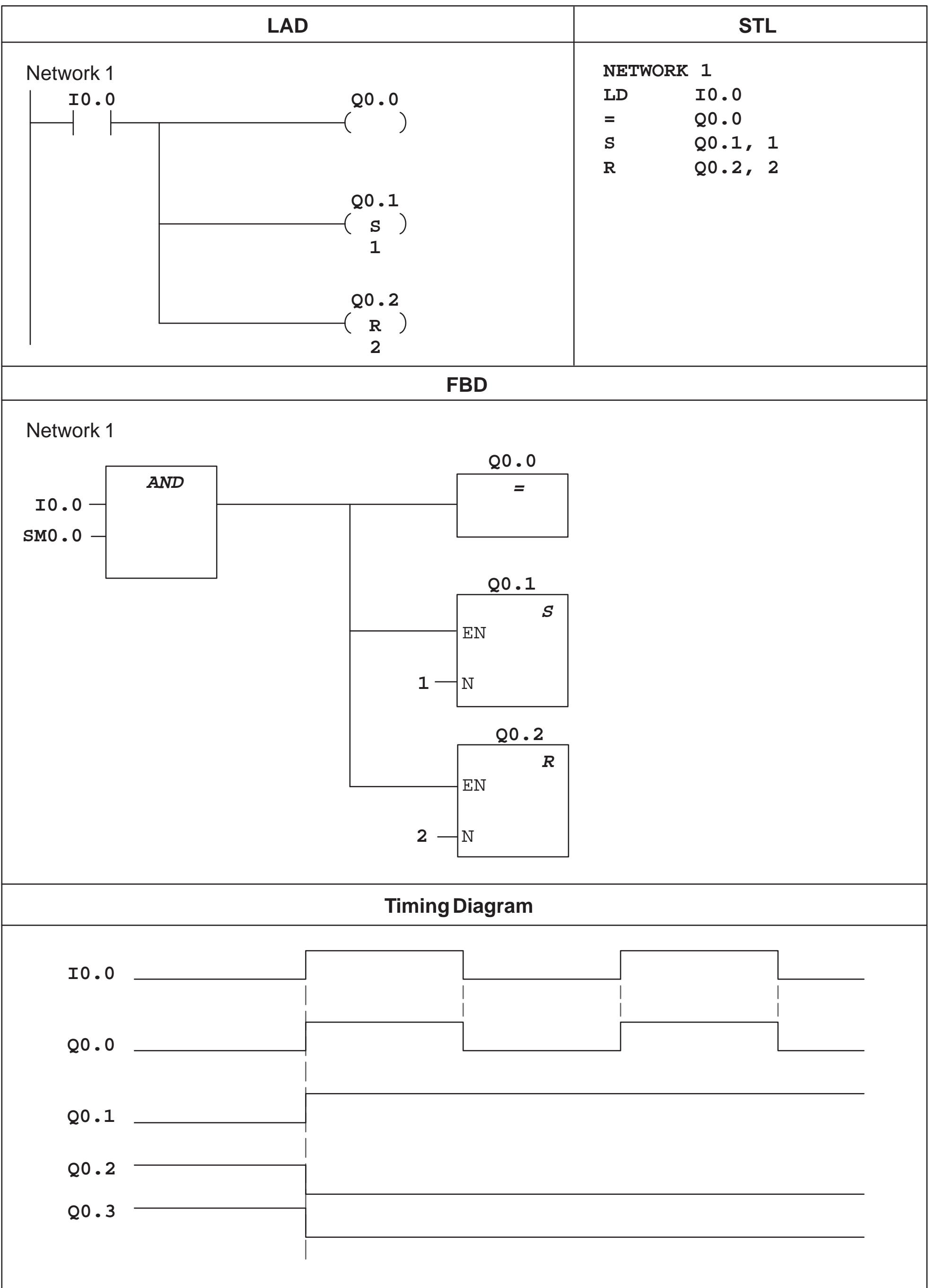
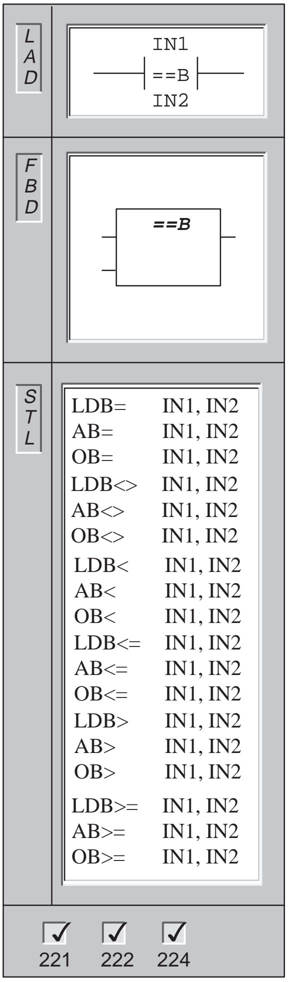


Figure 9-2 Examples of Output Instructions for SIMATIC LAD, STL, and FBD

9.2 SIMATIC Compare Instructions

Compare Byte



The **Compare Byte** instruction is used to compare two values: IN1 to IN2. Comparisons include: IN1 = IN2, IN1 >= IN2, IN1 <= IN2, IN1 > IN2, IN1 < IN2, or IN1 <> IN2.

Byte comparisons are unsigned.

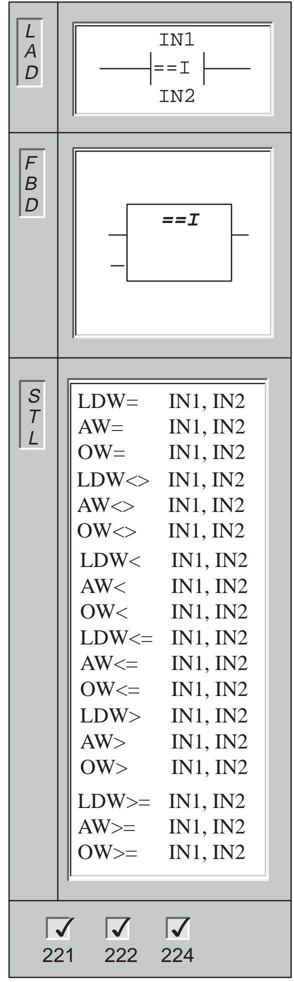
In LAD, the contact is on when the comparison is true.

In FBD, the output is on when the comparison is true.

In STL, the instructions Load, AND, or OR, a 1 with the top of stack when the comparison is true.

Inputs/Outputs	Operands	Data Types
Inputs	IB, QB, MB, SMB, VB, SB, LB, AC, Constant, *VD, *AC,*LD	BYTE
Outputs (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Compare Integer



The **Compare Integer** instruction is used to compare two values: IN1 to IN2. Comparisons include: IN1 = IN2, IN1 >= IN2, IN1 <= IN2, IN1 > IN2, IN1 < IN2, or IN1 <> IN2.

Integer comparisons are signed (16#7FFF > 16#8000).

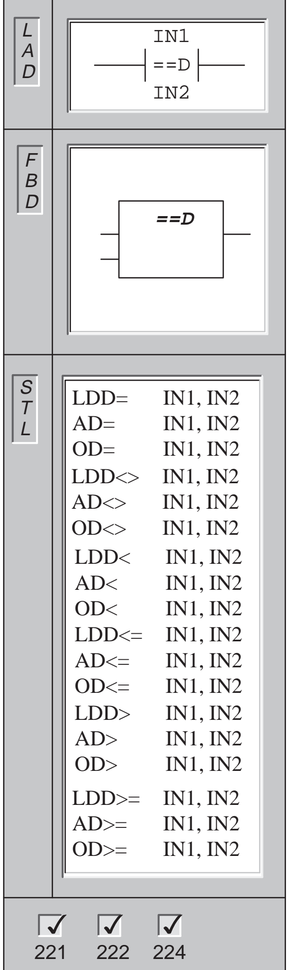
In LAD, the contact is on when the comparison is true.

In FBD, the output is on when the comparison is true.

In STL, the instructions Load, AND, or OR a 1 with the top of stack when the comparison is true.

Inputs/Outputs	Operands	Data Types
Inputs	IW, QW, MW, SW, SMW, T, C, VW, LW, AIW, AC, Constant, *VD, *AC,*LD	INT
Outputs (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Compare Double Word



The **Compare Double Word** instruction is used to compare two values: IN1 to IN2. Comparisons include: IN1 = IN2, IN1 >= IN2, IN1 <= IN2, IN1 > IN2, IN1 < IN2, or IN1 <> IN2.

Double word comparisons are signed (16#7FFFFFFF > 16#80000000).

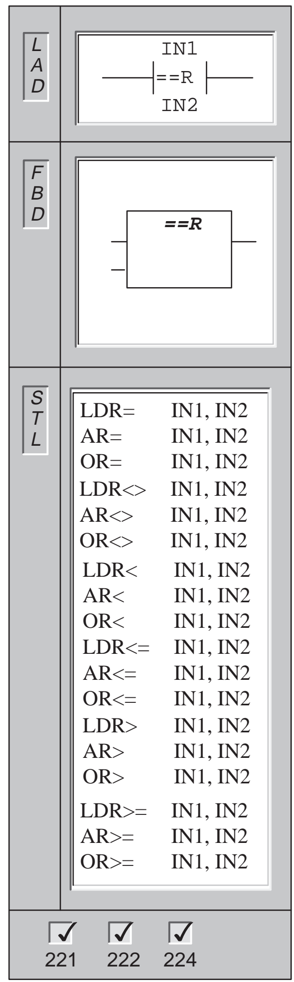
In LAD, the contact is on when the comparison is true.

In FBD, the output is on when the comparison is true.

In STL, the instructions Load, AND, or OR a 1 with the top of stack when the comparison is true.

Inputs/Outputs	Operands	Data Types
Inputs	ID, QD, MD, SD, SMD, VD, LD, HC, AC, Constant, *VD, *AC, *LD	DINT
Outputs (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Compare Real



Compare Real instruction is used to compare two values: IN1 to IN2. Comparisons include: IN1 = IN2, IN1 >= IN2, IN1 <= IN2, IN1 > IN2, IN1 < IN2, or IN1 <> IN2.

Real comparisons are signed.

In LAD, the contact is on when the comparison is true.

In FBD, the output is on when the comparison is true.

In STL, the instructions Load, AND, or OR a 1 with the top of stack when the comparison is true.

Inputs/Outputs	Operands	Data Types
Inputs	ID, QD, MD,SD, SMD, VD, LD, AC, Constant, *VD, *AC, *LD	REAL
Outputs (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL

Comparison Contact Examples

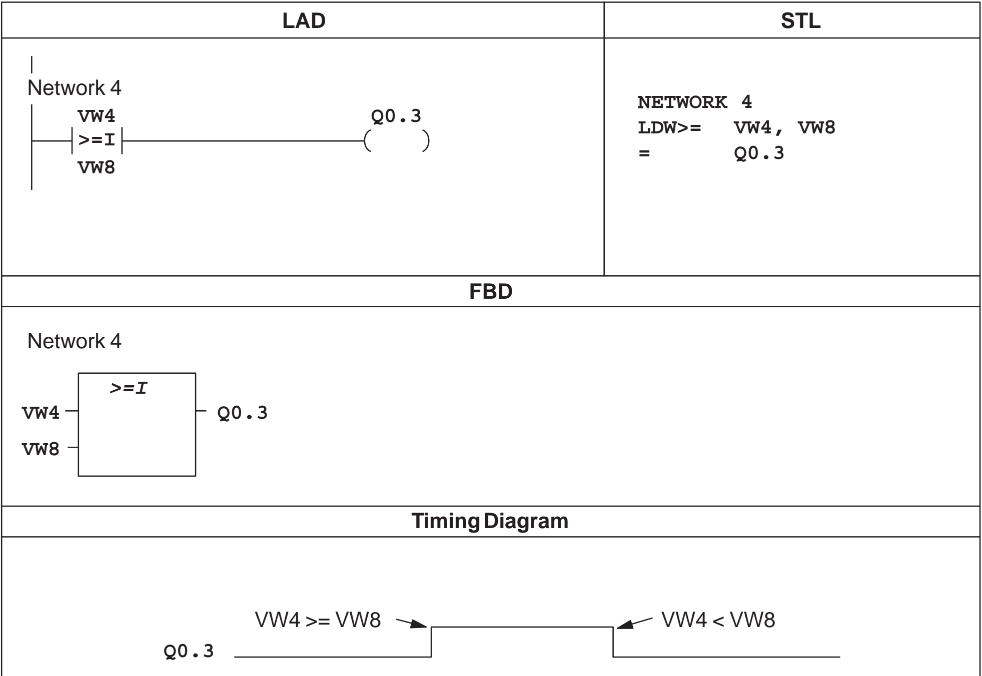
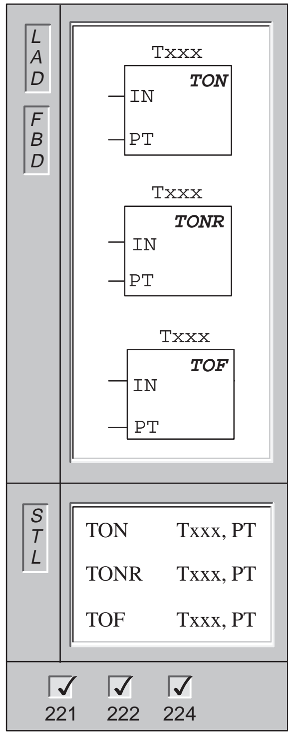


Figure 9-3 Examples of Comparison Contact Instructions for LAD and STL

efesotomasyon.com

9.3 SIMATIC Timer Instructions

On-Delay Timer, Retentive On-Delay Timer, Off-Delay Timer



The **On-Delay Timer** and **Retentive On-Delay Timer** instructions count time when the enabling input is ON. When the current value (Txxx) is greater than or equal to the preset time (PT), the timer bit is ON.

The On-Delay timer current value is cleared when the enabling input is OFF, while the current value of the Retentive On-Delay Timer is maintained when the input is OFF. You can use the Retentive On-Delay Timer to accumulate time for multiple periods of the input ON. A Reset instruction (R) is used to clear the current value of the Retentive On-Delay Timer.

Both the On-Delay Timer and the Retentive On-Delay Timers continue counting after the Preset is reached, and they stop counting at the maximum value of 32767.

The **Off-Delay Timer** is used to delay turning an output OFF for a fixed period of time after the input turns OFF. When the enabling input turns ON, the timer bit turns ON immediately, and the current value is set to 0. When the input turns OFF, the timer counts until the elapsed time reaches the preset time. When the preset is reached, the timer bit turns OFF and the current value stops counting. If the input is OFF for a time shorter than the preset value, the timer bit remains ON. The TOF instruction must see an ON to OFF transition to begin counting.

If the TOF timer is inside an SCR region and the SCR region is inactive, then the current value is set to 0, the timer bit is turned OFF, and the current value does not count.

Inputs/Outputs	Operands	Data Types
IN (LAD)	Power Flow	BOOL
IN (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL
PT	VW, IW, QW, MW, SW, SMW, LW, AIW, T, C, AC, Constant, *VD, *AC, *LD	INT

TON, TONR, and TOF timers are available in three resolutions. The resolution is determined by the timer number as shown in Table 9-1. Each count of the current value is a multiple of the time base. For example, a count of 50 on a 10-ms timer represents 500 ms.

Table 9-1 Timer Numbers and Resolutions

Timer Type	Resolution in milliseconds (ms)	Maximum Value in seconds (s)	Timer Number
TONR	1 ms	32.767 s	T0, T64
	10 ms	327.67 s	T1 to T4, T65 to T68
	100 ms	3276.7 s	T5 to T31, T69 to T95
TON, TOF	1 ms	32.767 s	T32, T96
	10 ms	327.67 s	T33 to T36, T97 to T100
	100 ms	3276.7 s	T37 to T63, T101 to T255

Note

You cannot share the same timer numbers for TOF and TON. For example, you cannot have both a TON T32 and a TOF T32.

Understanding the S7-200 Timer Instructions

You can use timers to implement time-based counting functions. The S7-200 instruction set provides three types of timers as shown below. Table 9-2 shows the actions of the different timers.

- On-Delay Timer (TON) for timing a single interval
- Retentive On-Delay Timer (TONR) for accumulating a number of timed intervals
- Off-Delay Timer (TOF) for extending time past a false condition (in other words, such as cooling a motor after it is turned off)

Table 9-2 Timer Actions

Timer Type	Current >= Preset	Enabling Input ON	Enabling Input OFF	Power Cycle/ First Scan
TON	Timer bit ON, Current continues counting to 32,767	Current value counts time	Timer bit OFF, Current value = 0	Timer bit OFF, Current value = 0
TONR	Timer bit ON, Current continues counting to 32,767	Current value counts time	Timer bit and current value maintain last state	Timer bit OFF, Current value may be maintained[1]
TOF	Timer bit OFF, Current = Preset, stops counting	Timer bit ON, Current value = 0	Timer counts after ON to OFF transition	Timer bit OFF, Current value = 0

[1] The retentive timer current value can be selected for retention through a power cycle. See Section 5.3 for information about memory retention for the S7-200 CPU.

Note

The Reset (R) instruction can be used to reset any timer. The TONR timer can only be reset by the Reset instruction. The Reset instruction performs the following operations:

Timer Bit = OFF
Timer Current = 0

After a reset, TOF timers require the enabling input to make the transition from ON to OFF in order to restart.

The actions of the timers at different resolutions are explained below.

1-Millisecond Resolution

The 1-ms timers count the number of 1-ms timer intervals that have elapsed since the active 1-ms timer was enabled. The execution of the timer instruction starts the timing; however, the 1-ms timers are updated (timer bit and timer current) every millisecond asynchronous to the scan cycle. In other words, the timer bit and timer current are updated multiple times throughout any scan that is greater than 1 ms.

The timer instruction is used to turn the timer on, reset the timer, or, in the case of the TONR timer, to turn the timer off.

Since the timer can be started anywhere within a millisecond, the preset must be set to one time interval greater than the minimum desired timer interval. For example, to guarantee a timed interval of at least 56 ms using a 1-ms timer, the preset time value should be set to 57.

10-Millisecond Resolution

The 10-ms timers count the number of 10-ms timer intervals that have elapsed since the active 10-ms timer was enabled. The execution of the timer instruction starts the timing, however the 10-ms timers are updated at the beginning of each scan cycle (in other words, the timer current and timer bit remain constant throughout the scan), by adding the accumulated number of 10-ms intervals (since the beginning of the previous scan) to the current value for the active timer.

Since the timer can be started anywhere within a 10-ms interval, the preset must be set to one time interval greater than the minimum desired timer interval. For example, to guarantee a timed interval of at least 140 ms using a 10-ms timer, the preset time value should be set to 15.

100-Millisecond Resolution

The 100-ms timers count the number of 100-ms timer intervals that have elapsed since the active 100-ms timer was last updated. These timers are updated by adding the accumulated number of 100-ms intervals (since the previous scan cycle) to the timer's current value when the timer instruction is executed.

The current value of a 100-ms timer is updated only if the timer instruction is executed. Consequently, if a 100-ms timer is enabled but the timer instruction is not executed each scan cycle, the current value for that timer is not updated and it loses time. Likewise, if the same 100-ms timer instruction is executed multiple times in a single scan cycle, the number of 100-ms intervals are added to the timer's current value multiple times, and it gains time. 100-ms timers should only be used where the timer instruction is executed exactly once per scan cycle.

Since the timer can be started anywhere within a 100-ms interval, the preset must be set to one time interval greater than the minimum desired timer interval. For example, to guarantee a timed interval of at least 2100 ms using a 100-ms timer, the preset time value should be set to 22.

Updating the Timer Current Value

The effect of the various ways in which current time values are updated depends upon how the timers are used. For example, consider the timer operation shown in Figure 9-4.

- In the case where the 1-ms timer is used, Q0.0 is turned on for one scan whenever the timer's current value is updated after the normally closed contact T32 is executed and before the normally open contact T32 is executed.
- In the case where the 10-ms timer is used, Q0.0 is never turned on, because the timer bit T33 is turned on from the top of the scan to the point where the timer box is executed. Once the timer box has been executed, the timer's current value and its T-bit is set to zero. When the normally open contact T33 is executed, T33 is off and Q0.0 is turned off.
- In the case where the 100-ms timer is used, Q0.0 is always turned on for one scan whenever the timer's current value reaches the preset value.

By using the normally closed contact Q0.0 instead of the timer bit as the enabling input to the timer box, the output Q0.0 is guaranteed to be turned on for one scan each time the timer reaches the preset value.

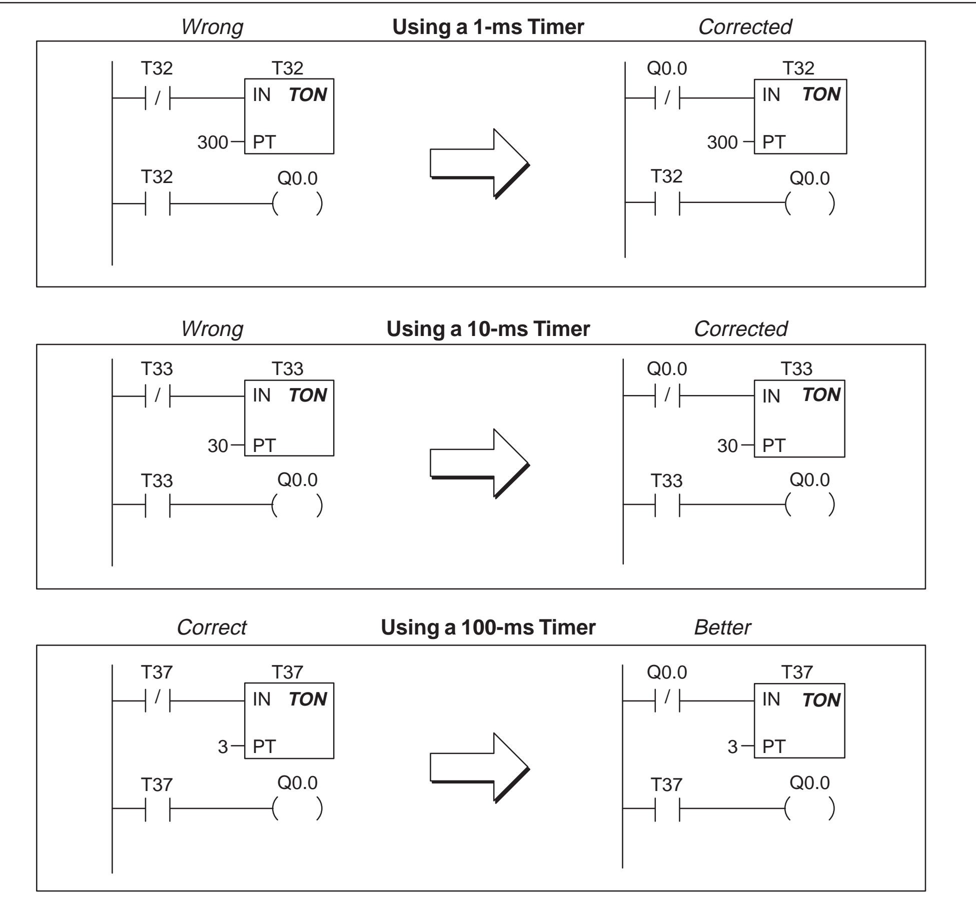


Figure 9-4 Example of Automatically Retriggered One Shot Timer

On-Delay Timer Example

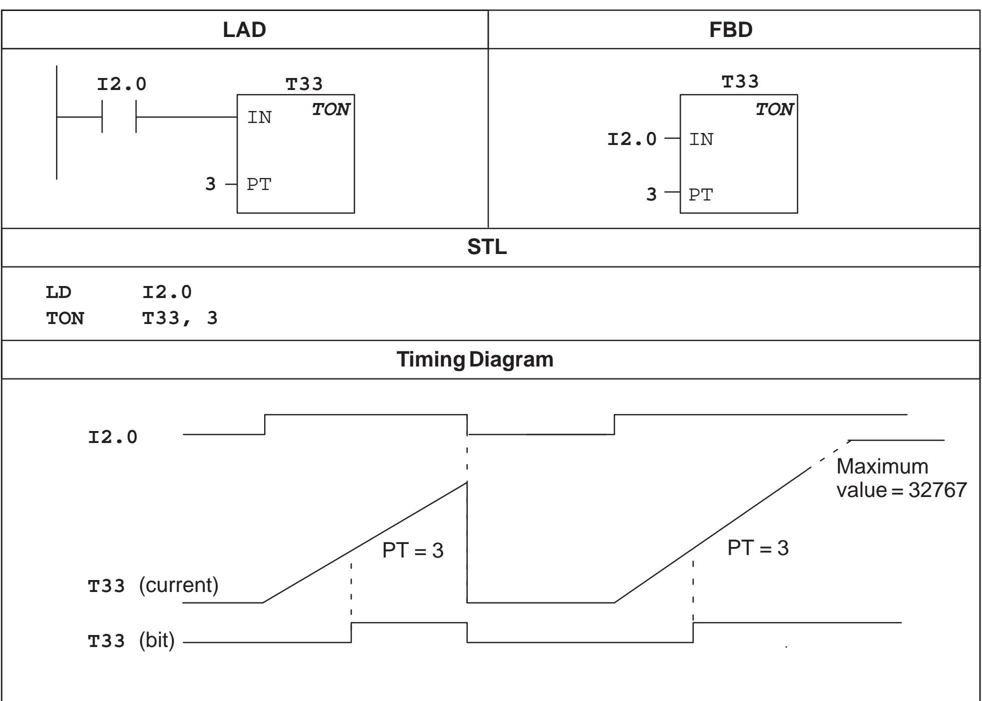


Figure 9-5 Example of On-Delay Timer Instruction for LAD, FBD, and STL

Retentive On-Delay Timer Example

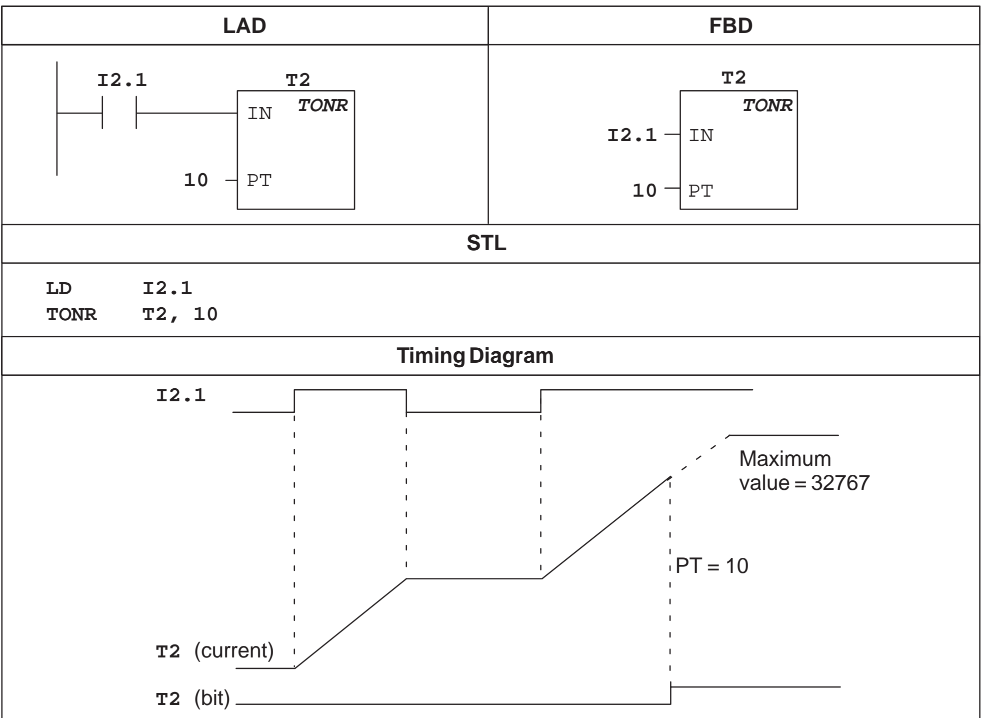


Figure 9-6 Example of Retentive On-Delay Timer Instruction for LAD, FBD, and STL

Off-Delay Timer Example

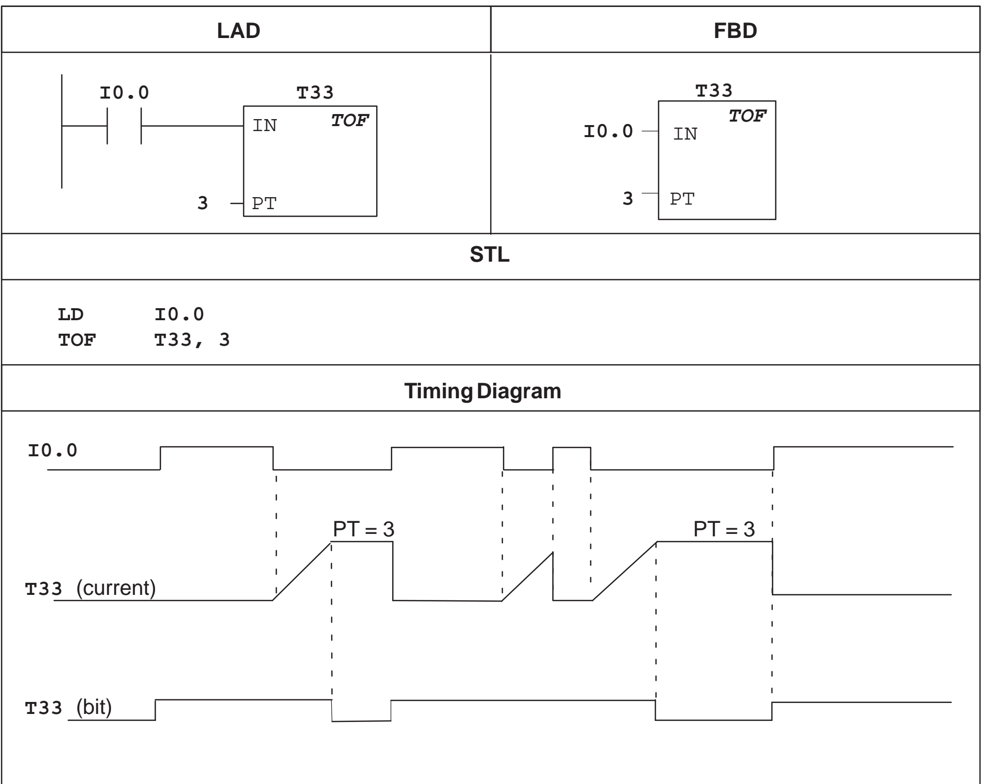
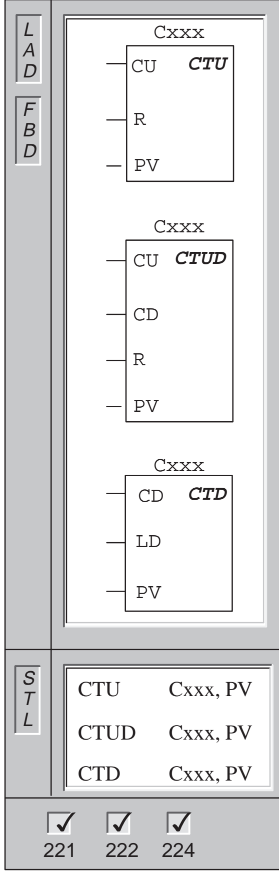


Figure 9-7 Example of Off-Delay Timer Instruction for LAD, FBD, and STL

9.4 SIMATIC Counter Instructions

Count Up, Count Up/Down, Count Down



The **Count Up** instruction counts up to the maximum value on the rising edges of the Count Up (CU) input. When the current value (Cxxx) is greater than or equal to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter is reset when the Reset (R) input turns on.

The **Count Up/Down** instruction counts up on rising edges of the Count Up (CU) input. It counts down on the rising edges of the Count Down (CD) input. When the current value (Cxxx) is greater than or equal to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter is reset when the Reset (R) input turns on.

The **Count Down Counter** counts down from the preset value on the rising edges of the Count Down (CD) input . When the current value is equal to zero, the counter bit (Cxxx) turns on. The counter resets the counter bit (Cxxx) and loads the current value with the preset value (PV) when the load input (LD) turns on. The Down Counter stops counting when it reaches zero.

Counter ranges: Cxxx=C0 through C255

In STL, the CTU Reset input is the top of the stack value, while the Count Up input is the value loaded in the second stack location.

In STL, the CTUD Reset input is the top of the stack value, the Count Down input is the value loaded in the second stack location, and the Count Up input is the value loaded in the third stack location.

In STL, the CTD Load input is the top of stack, and the Count Down input is the value loaded in the second stack location.

Inputs/Outputs	Operands	Data Types
CU, CD (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL
R, LD (FBD)	I, Q, M, SM, T, C, V, S, L, Power Flow	BOOL
PV	VW, IW, QW, MW, SMW, LW, AIW, AC, T, C, Constant, *VD, *AC, *LD, SW	INT

Understanding the S7-200 Counter Instructions

The Up Counter (CTU) counts up from the current value of that counter each time the count-up input makes the transition from off to on. The counter is reset when the reset input turns on, or when the Reset instruction is executed. The counter stops upon reaching the maximum value (32,767).

The Up/Down Counter (CTUD) counts up each time the count-up input makes the transition from off to on, and counts down each time the count-down input makes the transition from off to on. The counter is reset when the reset input turns on, or when the Reset instruction is executed. Upon reaching maximum value (32,767), the next rising edge at the count-up input causes the current count to wrap around to the minimum value (-32,768). Likewise on reaching the minimum value (-32,768), the next rising edge at the count-down input causes the current count to wrap around to the maximum value (32,767).

The Up and Up/Down counters have a current value that maintains the current count. They also have a preset value (PV) that is compared to the current value whenever the counter instruction is executed. When the current value is greater than or equal to the preset value, the counter bit (C-bit) turns on. Otherwise, the C-bit turns off.

The Down counter counts down from the current value of that counter each time the count down input makes the transition from off to on. The counter resets the counter bit and loads the current value with the preset value when the load input turns on. The counter stops upon reaching zero, and the counter bit (C-bit) turns on.

When you reset a counter using the Reset instruction, the counter bit is reset and the counter current value is set to zero. Use the counter number to reference both the current value and the C-bit of that counter.

Note

Since there is one current value for each counter, do not assign the same number to more than one counter. (Up Counters, Up/Down Counters, and Down counters with the same number access the same current value.)

Counter Examples

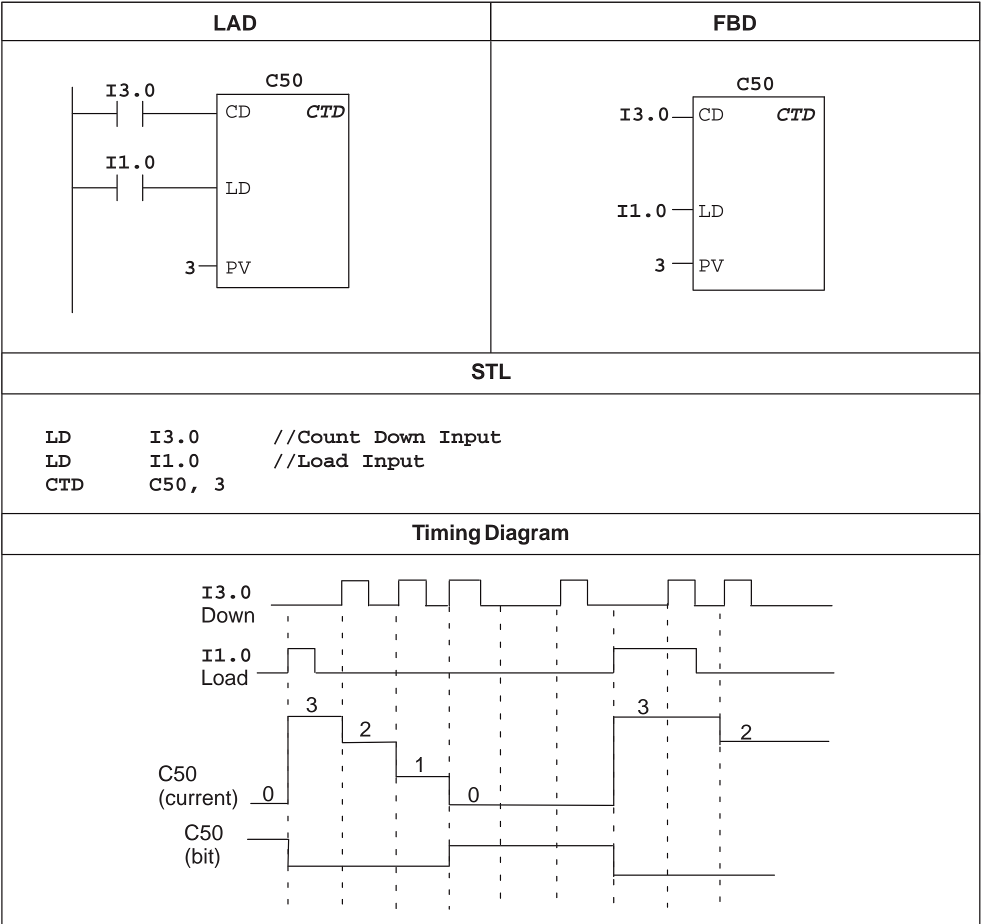


Figure 9-8 Example of CTD Counter Instruction for LAD, FBD, and STL

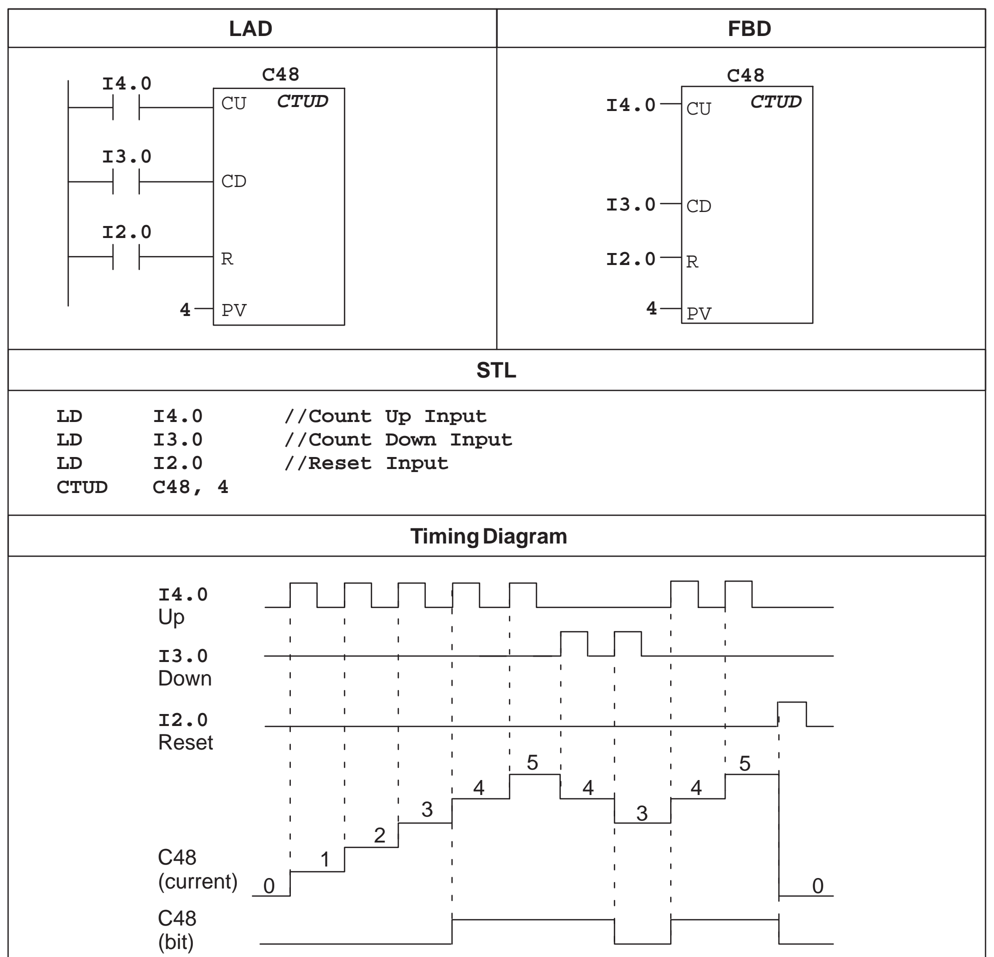
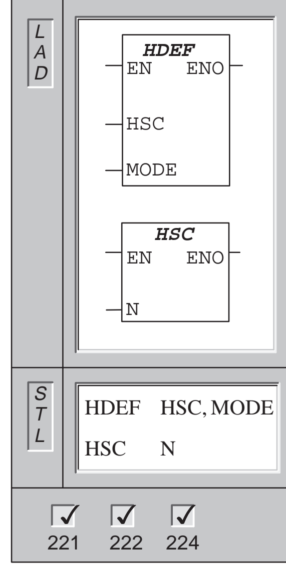


Figure 9-9 Example of CTUD Counter Instruction for LAD, FBD, and STL

9.5 SIMATIC High-Speed Counter Instructions

High-Speed Counter Definition, High-Speed Counter



The **High-Speed Counter Definition** instruction assigns a MODE to the referenced high-speed counter (HSC). See Table 9-5.

The **High-Speed Counter** instruction, when executed, configures and controls the operational mode of the high-speed counter, based on the state of the HSC special memory bits. The parameter N specifies the high-speed counter number.

CPU 221 and CPU 222 do not support HSC1 and HSC2.

Only one HDEF box may be used per counter.

HDEF: Error conditions that set ENO = 0:

SM4.3 (run-time), 0003 (input point conflict), 0004 (illegal instruction in interrupt), 000A (HSC redefinition)

HSC: Error conditions that set ENO = 0:

SM4.3 (run-time), 0001 (HSC before HDEF), 0005 (simultaneous HSC/PLS)

Inputs/Outputs	Operands	Data Types
HSC	Constant	BYTE
MODE	Constant	BYTE
N	Constant	WORD

Understanding the High-Speed Counter Instructions

High-speed counters count high-speed events that cannot be controlled at CPU scan rates, and can be configured for up to twelve different modes of operation. The counter modes are listed in Table 9-5. The maximum counting frequency of a high-speed counter is dependent upon your CPU type. See Appendix A for more information about your CPU.

Each counter has dedicated inputs for clocks, direction control, reset, and start, where these functions are supported. For the two-phase counters, both clocks may run at their maximum rates. In quadrature modes, an option is provided to select one times (1x) or four times (4x) the maximum counting rates. All counters run at maximum rates without interfering with one another.

Using the High-Speed Counter

Typically, a high-speed counter is used as the drive for a drum timer, where a shaft rotating at a constant speed is fitted with an incremental shaft encoder. The shaft encoder provides a specified number of counts per revolution and a reset pulse that occurs once per revolution. The clock(s) and the reset pulse from the shaft encoder provide the inputs to the high-speed counter. The high-speed counter is loaded with the first of several presets, and the desired outputs are activated for the time period where the current count is less than the current preset. The counter is set up to provide an interrupt when the current count is equal to preset and also when reset occurs.

As each current-count-value-equals-preset-value interrupt event occurs, a new preset is loaded and the next state for the outputs is set. When the reset interrupt event occurs, the first preset and the first output states are set, and the cycle is repeated.

Since the interrupts occur at a much lower rate than the counting rates of the high-speed counters, precise control of high-speed operations can be implemented with relatively minor impact to the overall scan cycle of the programmable logic controller. The method of interrupt attachment allows each load of a new preset to be performed in a separate interrupt routine for easy state control, making the program very straightforward and easy to follow. Of course, all interrupt events can be processed in a single interrupt routine. For more information about the interrupt instructions, see Section 9.16.

Understanding the Detailed Timing for the High-Speed Counters

The following timing diagrams (Figure 9-10 through Figure 9-16) show how each counter functions according to mode. The operation of the reset and start inputs is shown in a separate timing diagram and applies to all modes that use reset and start inputs. In the diagrams for the reset and start inputs, both reset and start are shown with the active state programmed to a high level.

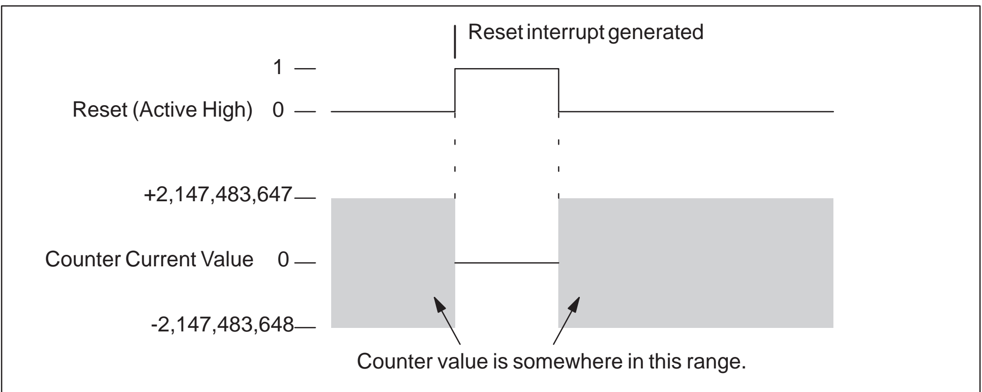


Figure 9-10 Operation Example with Reset and without Start

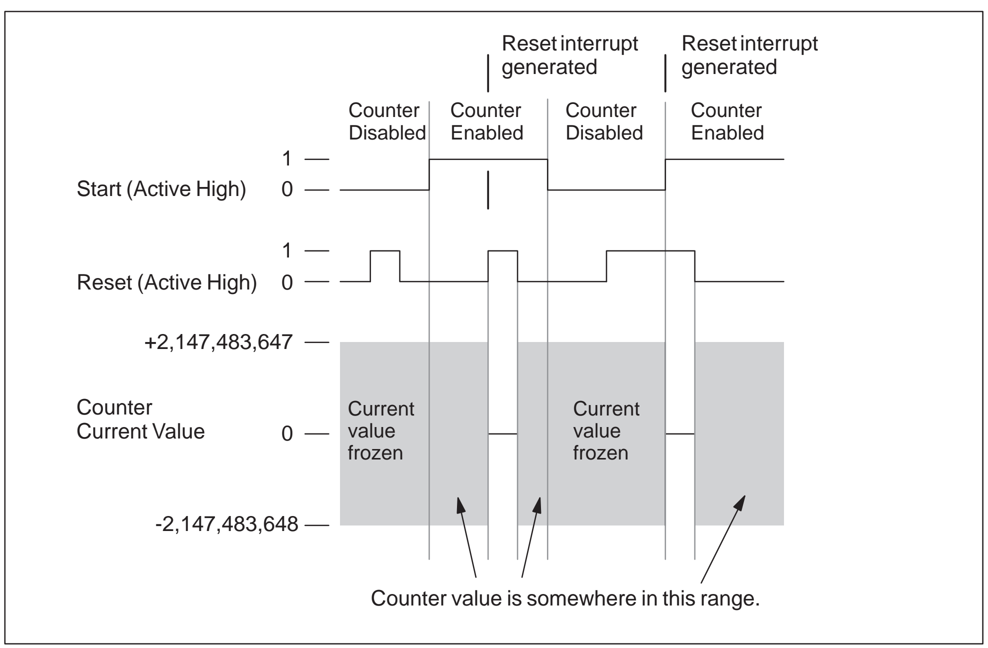


Figure 9-11 Operation Example with Reset and Start

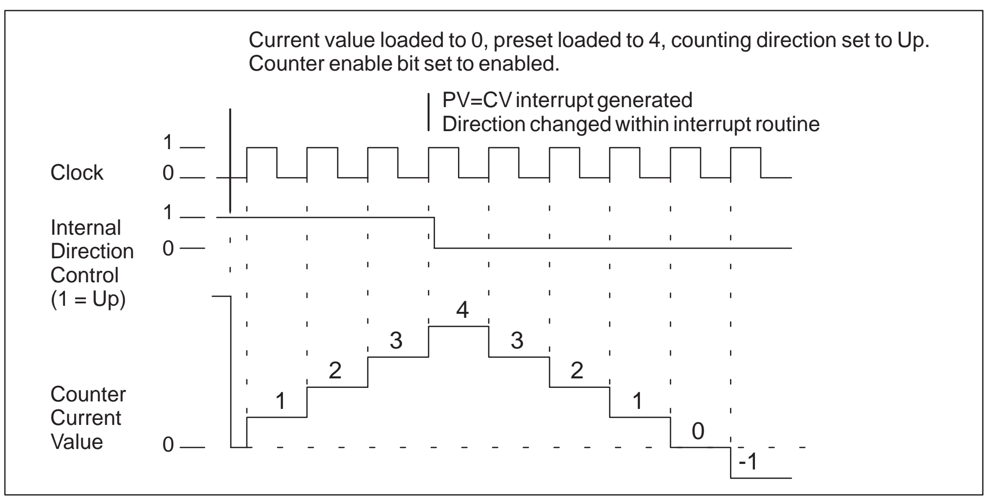


Figure 9-12 Operation Example of Modes 0, 1, or 2

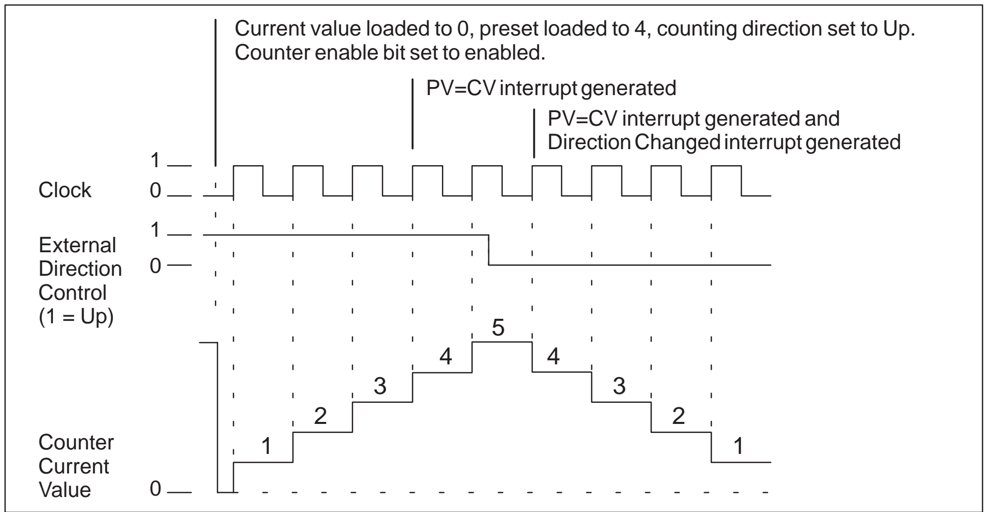


Figure 9-13 Operation Example of Modes 3, 4, or 5

When you use counting modes 6, 7, or 8 and a rising edge on both the up clock and down clock inputs occurs within 0.3 microseconds of each other, the high-speed counter may see these events as happening simultaneously. If this happens, the current value is unchanged and no change in counting direction is indicated. As long as the separation between rising edges of the up and down clock inputs is greater than this time period, the high-speed counter captures each event separately. In either case, no error is generated and the counter maintains the correct count value. See Figure 9-14, Figure 9-15, and Figure 9-16.

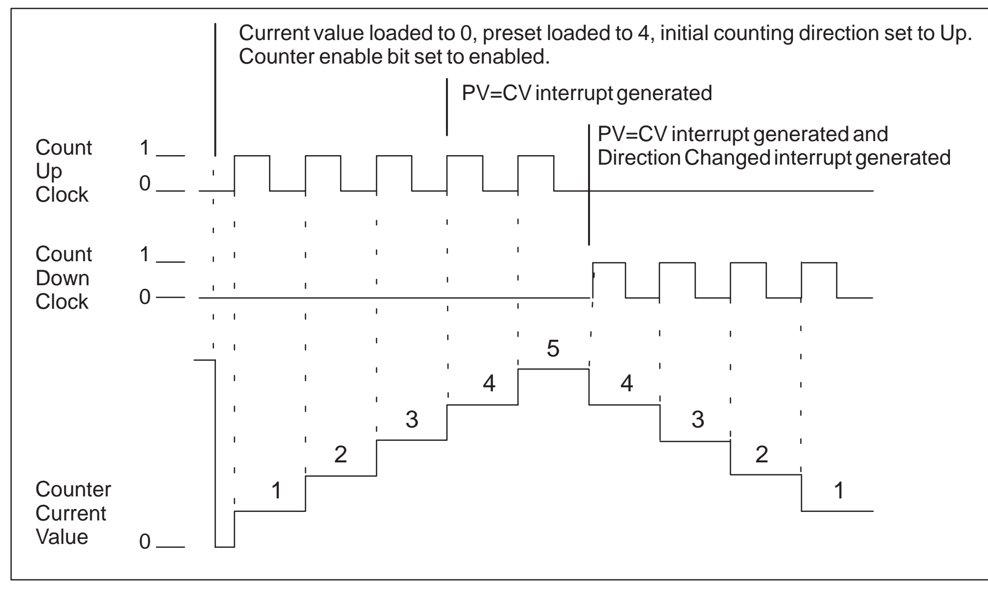


Figure 9-14 Operation Example of Modes 6, 7, or 8

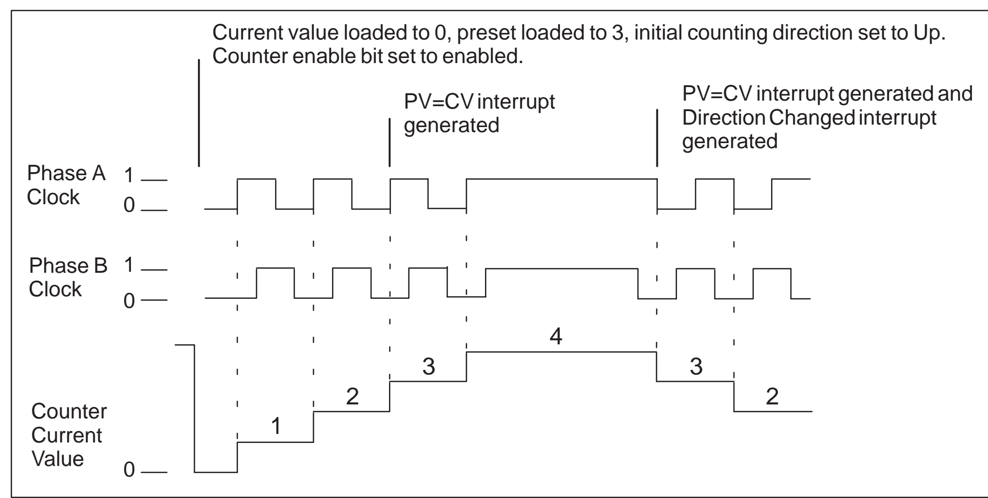


Figure 9-15 Operation Example of Modes 9, 10, or 11 (Quadrature 1x Mode)

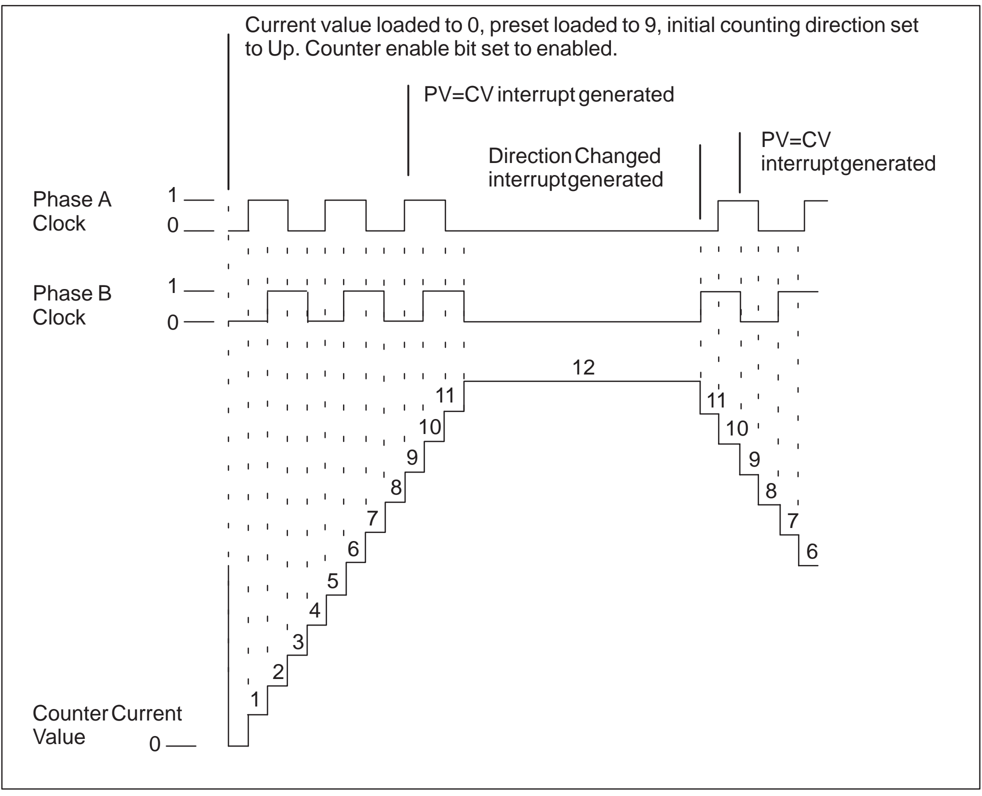


Figure 9-16 Operation Example of Modes 9, 10, or 11 (Quadrature 4x Mode)

Connecting the Input Wiring for the High-Speed Counters

Table 9-3 shows the inputs used for the clock, direction control, reset, and start functions associated with the high-speed counters. These input functions and the HSC modes of operation are described in Table 9-5 through Table 9-10.

Table 9-3 Dedicated Inputs for High-Speed Counters

High-Speed Counter	Inputs Used
HSC0	I0.0, I0.1, 0.2
HSC1	I0.6, I0.7, I1.0, I1.1
HSC2	I1.2, I1.3, I1.4, I1.5
HSC3	I0.1
HSC4	I0.3, I0.4, I0.5
HSC5	I0.4

There is some overlap in the input point assignments for some high-speed counters and edge interrupts, as shown in the shaded area of Table 9-4. The same input cannot be used for two different functions, however, any input not being used by the present mode of its high-speed counter can be used for another purpose. For example, if HSC0 is being used in mode 2 which uses I0.0 and I0.2, I0.1 can be used for edge interrupts or for HSC3.

If a mode of HSC0 is used that does not use input I0.1, then this input is available for use as either HSC3 or edge interrupts. Similarly, if I0.2 is not used in the selected HSC0 mode, this input is available for edge interrupts; and if I0.4 is not used in the selected HSC4 mode, this input is available for HSC5. Note that all modes of HSC0 always use I0.0 and all modes of HSC4 always use I0.3, so these points are never available for other uses when these counters are in use.

Table 9-4 Input Point Assignments for High-Speed Counters and Edge Interrupts

	Input Point (I)													
Element	**0.0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
HSC0	x	x	x											
HSC1							x	x	x	x				
HSC2											x	x	x	x
HSC3		x												
HSC4				x	x	x								
HSC5					x									
Edge Interrupts	x	x	x	x										

Table 9-5 HSC0 Modes of Operation

	HSC0				
Mode	**Description**	**I0.0**	**I0.1**	**I0.2**	
0	Single phase up/down counter with internal direction control SM37.3 = 0, count down SM37.3 = 1, count up	Clock			
1				Reset	
3	Single phase up/down counter with external direction control I0.1 = 0, count down I0.1 = 1, count up	Clock	Dir.		
4				Reset	
6	Two-phase counter with count up and count down clock inputs	Clock (Up)	Clock (Dn)		
7				Reset	
9	A/B phase quadrature counter, phase A leads B by 90 degrees for clockwise rotation, phase B leads A by 90 degrees for counterclockwise rotation	Clock Phase A	Clock Phase B		
10				Reset	

Table 9-6 HSC1 Modes of Operation

<table>
<tr><th colspan="6">HSC1</th></tr>
<tr><th>Mode</th><th>Description</th><th>I0.6</th><th>I0.7</th><th>I1.0</th><th>I1.1</th></tr>
<tr><td>0</td><td rowspan="3">Single phase up/down counter with internal direction control
SM47.3 = 0, count down
SM47.3 = 1, count up</td><td rowspan="3">Clock</td><td rowspan="3"></td><td></td><td rowspan="2"></td></tr>
<tr><td>1</td><td rowspan="2">Reset</td></tr>
<tr><td>2</td><td>Start</td></tr>
<tr><td>3</td><td rowspan="3">Single phase up/down counter with external direction control
I0.7 = 0, count down
I0.7 = 1, count up</td><td rowspan="3">Clock</td><td rowspan="3">Dir.</td><td></td><td rowspan="2"></td></tr>
<tr><td>4</td><td rowspan="2">Reset</td></tr>
<tr><td>5</td><td>Start</td></tr>
<tr><td>6</td><td rowspan="3">Two-phase counter with count up and count down clock inputs</td><td rowspan="3">Clock (Up)</td><td rowspan="3">Clock (Dn)</td><td></td><td rowspan="2"></td></tr>
<tr><td>7</td><td rowspan="2">Reset</td></tr>
<tr><td>8</td><td>Start</td></tr>
<tr><td>9</td><td rowspan="3">A/B phase quadrature counter,
phase A leads B by 90 degrees for clockwise rotation,
phase B leads A by 90 degrees for counterclockwise rotation</td><td rowspan="3">Clock Phase A</td><td rowspan="3">Clock Phase B</td><td></td><td rowspan="2"></td></tr>
<tr><td>10</td><td rowspan="2">Reset</td></tr>
<tr><td>11</td><td>Start</td></tr>
</table>

Table 9-7 HSC2 Modes of Operation

<table>
<tr><th colspan="6">HSC2</th></tr>
<tr><th>Mode</th><th>Description</th><th>I1.2</th><th>I1.3</th><th>I1.4</th><th>I1.5</th></tr>
<tr><td>0</td><td rowspan="3">Single phase up/down counter with internal direction control
SM57.3 = 0, count down
SM57.3 = 1, count up</td><td rowspan="3">Clock</td><td rowspan="3"></td><td></td><td rowspan="2"></td></tr>
<tr><td>1</td><td rowspan="2">Reset</td></tr>
<tr><td>2</td><td>Start</td></tr>
<tr><td>3</td><td rowspan="3">Single phase up/down counter with external direction control
I1.3 = 0, count down
I1.3 = 1, count up</td><td rowspan="3">Clock</td><td rowspan="3">Dir.</td><td></td><td rowspan="2"></td></tr>
<tr><td>4</td><td rowspan="2">Reset</td></tr>
<tr><td>5</td><td>Start</td></tr>
<tr><td>6</td><td rowspan="3">Two phase counter with count up and count down clock inputs</td><td rowspan="3">Clock (Up)</td><td rowspan="3">Clock (Dn)</td><td></td><td rowspan="2"></td></tr>
<tr><td>7</td><td rowspan="2">Reset</td></tr>
<tr><td>8</td><td>Start</td></tr>
<tr><td>9</td><td rowspan="3">A/B phase quadrature counter,
phase A leads B by 90 degrees for clockwise rotation,
phase B leads A by 90 degrees for counterclockwise rotation</td><td rowspan="3">Clock Phase A</td><td rowspan="3">Clock Phase B</td><td></td><td rowspan="2"></td></tr>
<tr><td>10</td><td rowspan="2">Reset</td></tr>
<tr><td>11</td><td>Start</td></tr>
</table>

efesotomasyon.com

Table 9-8 HSC3 Modes of Operation

HSC3					
Mode	**Description**	**I0.1**			
0	Single phase up/down counter with internal direction control SM137.3 = 0, count down SM137.3 = 1, count up	Clock			

Table 9-9 HSC4 Modes of Operation

HSC4					
Mode	**Description**	**I0.3**	**I0.4**	**I0.5**	
0	Single phase up/down counter with internal direction control SM147.3 = 0, count down SM147.3 = 1, count up	Clock			
1				Reset	
3	Single phase up/down counter with external direction control I0.4 = 0, count down I0.4 = 1, count up	Clock	Dir.		
4				Reset	
6	Two phase counter with count up and count down clock inputs	Clock (Up)	Clock (Dn)		
7				Reset	
9	A/B phase quadrature counter, phase A leads B by 90 degrees for clockwise rotation, phase B leads A by 90 degrees for counterclockwise rotation	Clock Phase A	Clock Phase B		
10				Reset	

Table 9-10 HSC5 Modes of Operation

HSC5					
Mode	**Description**	**I0.4**			
0	Single phase up/down counter with internal direction control SM157.3 = 0, count down SM157.3 = 1, count up	Clock			

Addressing the High-Speed Counters (HC)

To access the count value for the high-speed counter, you specify the address of the high-speed counter, using the memory type (HC) and the counter number (such as HC0). The current value of the high-speed counter is a read-only value and can be addressed only as a double word (32 bits), as shown in Figure 9-17.

Format: **HC*[high-speed counter number]*** **HC2**

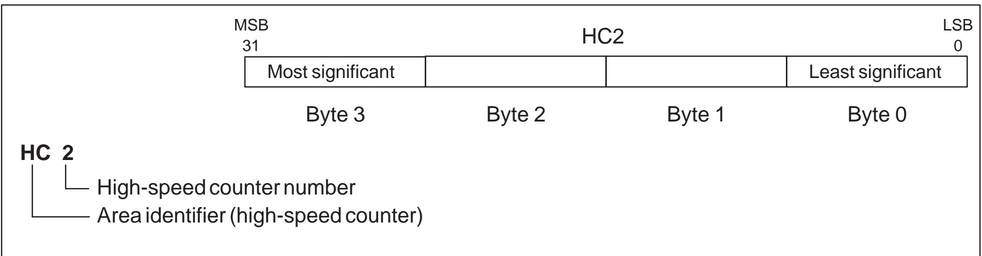


Figure 9-17 Accessing the High-Speed Counter Current Values

Understanding the Different High-Speed Counters

All counters function the same way for the same counter mode of operation. There are four basic types of counter modes as shown in Table 9-5. Note that every mode is not supported by every counter. You can use each type: without reset or start inputs, with reset and without start, or with both start and reset inputs.

When you activate the reset input, it clears the current value and holds it cleared until you de-activate reset. When you activate the start input, it allows the counter to count. While start is de-activated, the current value of the counter is held constant and clocking events are ignored. If reset is activated while start is inactive, the reset is ignored and the current value is not changed. If the start input becomes active while the reset input is active, and the current value is cleared.

You must select the counter mode before a high-speed counter can be used. You can do this with the HDEF instruction (High-Speed Counter Definition). HDEF provides the association between a high-speed counter (HSCx) and a counter mode. You can only use one HDEF instruction for each high-speed counter. Define a high-speed counter by using the first scan memory bit, SM0.1 (this bit is turned on for the first scan and is then turned off), to call a subroutine that contains the HDEF instruction.

Selecting the Active State and 1x/4x Mode

Four counters have three control bits that are used to configure the active state of the reset and start inputs and to select 1x or 4x counting modes (quadrature counters only). These bits are located in the control byte for the respective counter and are only used when the HDEF instruction is executed. These bits are defined in Table 9-11.

You must set these control bits to the desired state before the HDEF instruction is executed. Otherwise, the counter takes on the default configuration for the counter mode selected. The default setting of the reset input and the start input are active high, and the quadrature counting rate is 4x (or four times the input clock frequency). Once the HDEF instruction has been executed, you cannot change the counter setup unless you first place the CPU in the STOP mode.

Table 9-11 Active Level for Reset, Start, and 1x/4x Control Bits

HSC0	HSC1	HSC2	HSC4	Description (used only when HDEF is executed)
SM37.0	SM47.0	SM57.0	SM147.0	Active level control bit for Reset: 0 = Reset is active high; 1 = Reset is active low
--	SM47.1	SM57.1	--	Active level control bit for Start: 0 = Start is active high; 1 = Start is active low
SM37.2	SM47.2	SM57.2	SM147.2	Counting rate selection for Quadrature counters: 0 = 4X counting rate; 1 = 1X counting rate

Control Byte

Once you have defined the counter and the counter mode, you can program the dynamic parameters of the counter. Each high-speed counter has a control byte that allows the counter to be enabled or disabled; the direction to be controlled (modes 0, 1, and 2 only), or the initial counting direction for all other modes; the current value to be loaded; and the preset value to be loaded. Examination of the control byte and associated current and preset values is invoked by the execution of the HSC instruction. Table 9-12 describes each of these control bits.

Table 9-12 Control Bits for HSC0, HSC1, and HSC2

HSC0	HSC1	HSC2	HSC3	HSC4	HSC5	Description
SM37.0	SM47.0	SM57.0	SM137.0	SM147.0	SM157.0	Not used after HDEF has been executed (Never used by counters that do not have an external reset input)
SM37.1	SM47.1	SM57.1	SM137.1	SM147.1	SM157.1	Not used after HDEF has been executed (Never used by counters that do not have a start input)
SM37.2	SM47.2	SM57.2	SM137.2	SM147.2	SM157.2	Not used after HDEF has been executed (Never used by counters that do not support quadrature counting)
SM37.3	SM47.3	SM57.3	SM137.3	SM147.3	SM157.3	Counting direction control bit: 0 = count down; 1 = count up
SM37.4	SM47.4	SM57.4	SM137.4	SM147.4	SM157.4	Write the counting direction to the HSC: 0 = no update; 1 = update direction
SM37.5	SM47.5	SM57.5	SM137.5	SM147.5	SM157.5	Write the new preset value to the HSC: 0 = no update; 1 = update preset
SM37.6	SM47.6	SM57.6	SM137.6	SM147.6	SM157.6	Write the new current value to the HSC: 0 = no update; 1 = update current value
SM37.7	SM47.7	SM57.7	SM137.7	SM147.7	SM157.7	Enable the HSC: 0 = disable the HSC; 1 = enable the HSC

Setting Current Values and Preset Values

Each high-speed counter has a 32-bit current value and a 32-bit preset value. Both the current and the preset values are signed integer values. To load a new current or preset value into the high-speed counter, you must set up the control byte and the special memory bytes that hold the current and/or preset values. You must then execute the HSC instruction to cause the new values to be transferred to the high-speed counter. Table 9-13 describes the special memory bytes used to hold the new current and preset values.

In addition to the control bytes and the new preset and current holding bytes, the current value of each high-speed counter can be read using the data type HC (High-Speed Counter Current) followed by the number (0, 1, 2, 3, 4, or 5) of the counter. Thus, the current value is directly accessible for read operations, but can only be written with the HSC instruction described above.

Table 9-13 Current and Preset Values of HSC0, HSC1, HSC2, HSC3, HSC4, and HSC5

Value to be Loaded	HSC0	HSC1	HSC2	HSC3	HSC4	HSC5
New current	SMD38	SMD48	SMD58	SMD138	SMD148	SMD158
New preset	SMD42	SMD52	SMD62	SMD142	SMD152	SMD162

Status Byte

A status byte is provided for each high-speed counter that provides status memory bits that indicate the current counting direction, and whether the current value is greater or equal to the preset value. Table 9-14 defines these status bits for each high-speed counter.

Table 9-14 Status Bits for HSC0, HSC1, HSC2, HSC3, HSC4, and HSC5

HSC0	HSC1	HSC2	HSC3	HSC4	HSC5	Description
SM36.0	SM46.0	SM56.0	SM136.0	SM146.0	SM156.0	Not used
SM36.1	SM46.1	SM56.1	SM136.1	SM146.1	SM156.1	Not used
SM36.2	SM46.2	SM56.2	SM136.2	SM146.2	SM156.2	Not used
SM36.3	SM46.3	SM56.3	SM136.3	SM146.3	SM156.3	Not used
SM36.4	SM46.4	SM56.4	SM136.4	SM146.4	SM156.4	Not used
SM36.5	SM46.5	SM56.5	SM136.5	SM146.5	SM156.5	Current counting direction status bit: 0 = counting down; 1 = counting up
SM36.6	SM46.6	SM56.6	SM136.6	SM146.6	SM156.6	Current value equals preset value status bit: 0 = not equal; 1 = equal
SM36.7	SM46.7	SM56.7	SM136.7	SM146.7	SM156.7	Current value greater than preset value status bit: 0 = less than or equal; 1 = greater than

Note

Status bits are valid only while the high-speed counter interrupt routine is being executed. The purpose of monitoring the state of the high-speed counter is to enable interrupts for the events that are of consequence to the operation being performed.

HSC Interrupts

All counter modes support an interrupt on current value equal to the preset value. Counter modes that use an external reset input support an interrupt on external reset activated. All counter modes except modes 0, 1, and 2 support an interrupt on a counting direction change. Each of these interrupt conditions may be enabled or disabled separately. For a complete discussion on the use of interrupts, see Section 9.16.

Note

When you are using the external reset interrupt, do not attempt to load a new current value or disable, then re-enable the high-speed counter from within the interrupt routine attached to that event. A fatal error condition can result.

To help you understand the operation of high-speed counters, the following descriptions of the initialization and operation sequences are provided. HSC1 is used as the model counter throughout these sequence descriptions. The initialization descriptions make the assumption that the S7-200 has just been placed in the RUN mode, and for that reason, the first scan memory bit is true. If this is not the case, remember that the HDEF instruction can be executed only one time for each high-speed counter after entering RUN mode. Executing HDEF for a high-speed counter a second time generates a run-time error and does not change the counter setup from the way it was set up on the first execution of HDEF for that counter.

Initialization Modes 0, 1, or 2

The following steps describe how to initialize HSC1 for Single Phase Up/Down Counter with Internal Direction (Modes 0, 1, or 2):

1. Use the first scan memory bit to call a subroutine in which the initialization operation is performed. Since you use a subroutine call, subsequent scans do not make the call to the subroutine, which reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB47 according to the desired control operation. For example:

   SMB47 = 16#F8 produces the following results:
   Enables the counter
   Writes a new current value
   Writes a new preset value
   Sets the direction to count up
   Sets the start and reset inputs to be active high
3. Execute the HDEF instruction with the HSC input set to 1 and the MODE input set to 0 for no external reset or start, to 1 for external reset and no start, or to 2 for both external reset and start.
4. Load SMD48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SMD52 (double word size value) with the desired preset value.
6. In order to capture the current value equal to preset event, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Section 9.16 for complete details on interrupt processing.
7. In order to capture an external reset event, program an interrupt by attaching the external reset interrupt event (event 15) to an interrupt routine.
8. Execute the global interrupt enable instruction (ENI) to enable interrupts.
9. Execute the HSC instruction to cause the S7-200 to program HSC1.
10. Exit the subroutine.

Initialization Modes 3, 4, or 5

The following steps describe how to initialize HSC1 for Single Phase Up/Down Counter with External Direction (Modes 3, 4, or 5):

1. Use the first scan memory bit to call a subroutine in which the initialization operation is performed. Since you use a subroutine call, subsequent scans do not make the call to the subroutine, which reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB47 according to the desired control operation. For example:

   SMB47 = 16#F8 produces the following results:
   Enables the counter
   Writes a new current value
   Writes a new preset value
   Sets the initial direction of the HSC to count up
   Sets the start and reset inputs to be active high
3. Execute the HDEF instruction with the HSC input set to 1 and the MODE input set to 3 for no external reset or start, 4 for external reset and no start, or 5 for both external reset and start.
4. Load SMD48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SMD52 (double word size value) with the desired preset value.
6. In order to capture the current value equal to preset event, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Section 9.16 for complete details on interrupt processing.
7. In order to capture direction changes, program an interrupt by attaching the direction changed interrupt event (event 14) to an interrupt routine.
8. In order to capture an external reset event, program an interrupt by attaching the external reset interrupt event (event 15) to an interrupt routine.
9. Execute the global interrupt enable instruction (ENI) to enable interrupts.
10. Execute the HSC instruction to cause the S7-200 to program HSC1.
11. Exit the subroutine.

Initialization Modes 6, 7, or 8

The following steps describe how to initialize HSC1 for Two Phase Up/Down Counter with Up/Down Clocks (Modes 6, 7, or 8):

1. Use the first scan memory bit to call a subroutine in which the initialization operations are performed. Since you use a subroutine call, subsequent scans do not make the call to the subroutine, which reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB47 according to the desired control operation. For example:

   SMB47 = 16#F8 produces the following results:
   Enables the counter
   Writes a new current value
   Writes a new preset value
   Sets the initial direction of the HSC to count up
   Sets the start and reset inputs to be active high
3. Execute the HDEF instruction with the HSC input set to 1 and the MODE set to 6 for no external reset or start, 7 for external reset and no start, or 8 for both external reset and start.
4. Load SMD48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SMD52 (double word size value) with the desired preset value.
6. In order to capture the current value equal to preset event, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Section 9.16 for complete details on interrupt processing.
7. In order to capture direction changes, program an interrupt by attaching the direction changed interrupt event (event 14) to an interrupt routine.
8. In order to capture an external reset event, program an interrupt by attaching the external reset interrupt event (event 15) to an interrupt routine.
9. Execute the global interrupt enable instruction (ENI) to enable interrupts.
10. Execute the HSC instruction to cause the S7-200 to program HSC1.
11. Exit the subroutine.

Initialization Modes 9, 10, or 11

The following steps describe how to initialize HSC1 for A/B Phase Quadrature Counter (Modes 9, 10, or 11):

1. Use the first scan memory bit to call a subroutine in which the initialization operations are performed. Since you use a subroutine call, subsequent scans do not make the call to the subroutine, which reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB47 according to the desired control operation.

   For example (1x counting mode):
   SMB47 = 16#FC produces the following results:
   Enables the counter
   Writes a new current value
   Writes a new preset value
   Sets the initial direction of the HSC to count up
   Sets the start and reset inputs to be active high

   For example (4x counting mode):
   SMB47 = 16#F8 produces the following results:
   Enables the counter
   Writes a new current value
   Writes a new preset value
   Sets the initial direction of the HSC to count up
   Sets the start and reset inputs to be active high
3. Execute the HDEF instruction with the HSC input set to 1 and the MODE input set to 9 for no external reset or start, 10 for external reset and no start, or 11 for both external reset and start.
4. Load SMD48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SMD52 (double word size value) with the desired preset value.
6. In order to capture the current value equal to preset event, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Section 9.16 for complete details on interrupt processing.
7. In order to capture direction changes, program an interrupt by attaching the direction changed interrupt event (event 14) to an interrupt routine.
8. In order to capture an external reset event, program an interrupt by attaching the external reset interrupt event (event 15) to an interrupt routine.
9. Execute the global interrupt enable instruction (ENI) to enable interrupts.
10. Execute the HSC instruction to cause the S7-200 to program HSC1.
11. Exit the subroutine.

Change Direction in Modes 0, 1, or 2

The following steps describe how to configure HSC1 for Change Direction for Single Phase Counter with Internal Direction (Modes 0, 1, or 2):

1. Load SMB47 to write the desired direction:

   SMB47 = 16#90 Enables the counter
   Sets the direction of the HSC to count down

   SMB47 = 16#98 Enables the counter
   Sets the direction of the HSC to count up

2. Execute the HSC instruction to cause the S7-200 to program HSC1.

Load a New Current Value (Any Mode)

The following steps describe how to change the counter current value of HSC1 (any mode):

Changing the current value forces the counter to be disabled while the change is made. While the counter is disabled, it does not count or generate interrupts.

1. Load SMB47 to write the desired current value:

   SMB47 = 16#C0 Enables the counter
   Writes the new current value

2. Load SMD48 (double word size) with the desired current value (load with 0 to clear it).
3. Execute the HSC instruction to cause the S7-200 to program HSC1.

Load a New Preset Value (Any Mode)

The following steps describe how to change the preset value of HSC1 (any mode):

1. Load SMB47 to write the desired preset value:

   SMB47 = 16#A0 Enables the counter
   Writes the new preset value

2. Load SMD52 (double word size value) with the desired preset value.
3. Execute the HSC instruction to cause the S7-200 to program HSC1.

Disable a High-Speed Counter (Any Mode)

The following steps describe how to disable the HSC1 high-speed counter (any mode):

1. Load SMB47 to disable the counter:

   SMB47 = 16#00 Disables the counter

2. Execute the HSC instruction to disable the counter.

Although the above sequences show how to change direction, current value, and preset value individually, you may change all or any combination of them in the same sequence by setting the value of SMB47 appropriately and then executing the HSC instruction.

High-Speed Counter Example

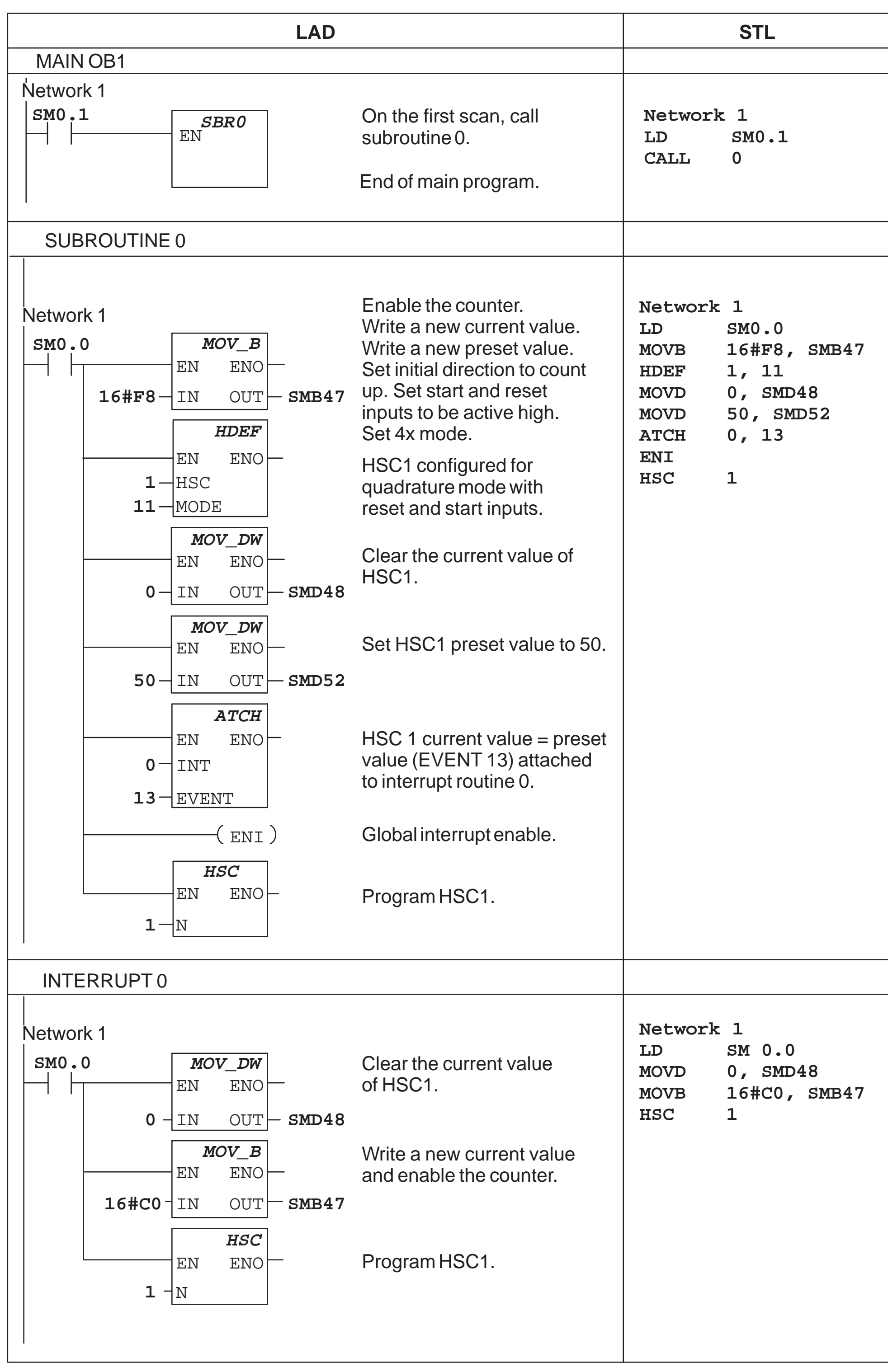


Figure 9-18 Example of Initialization of HSC1 (LAD and STL)

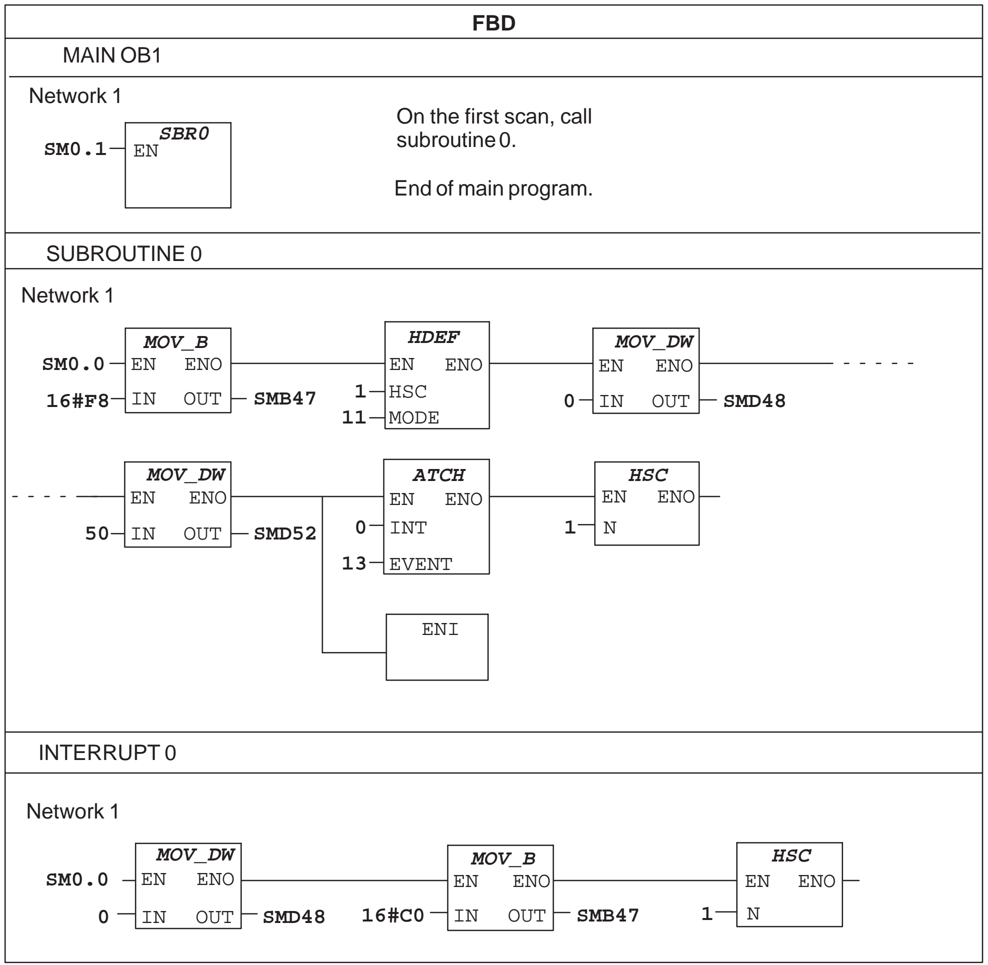
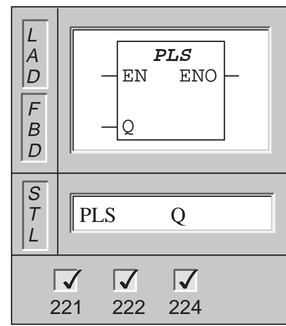


Figure 9-19 Example of Initialization of HSC1 (FBD)

9.6 SIMATIC Pulse Output Instructions

Pulse Output



The **Pulse Output** instruction examines the special memory bits for the pulse output (Q0.0 or Q0.1). The pulse operation defined by the special memory bits is then invoked.

Operands:	Q	Constant (0 or 1)
Data Types:	WORD	
Pulse Output Ranges		Q0.0 through Q0.1

Understanding the S7-200 High-Speed Output Instructions

The CPUs each have two PTO/PWM generators to output high-speed pulse train and pulse width modulated waveforms. One generator is assigned to digital output point Q0.0 and the other generator is assigned to digital output point Q0.1.

The PTO/PWM generators and the process-image register share the use of Q0.0 and Q0.1. When a PTO or PWM function is active on Q0.0 or Q0.1, the PTO/PWM generator has control of the output, and normal use of the output point is inhibited. The output waveform is not affected by the state of the process-image register, the forced value of the point, or execution of immediate output instructions. When the PTO/PWM generator is inactive, control of the output reverts to the process-image register. The process-image register determines the initial and final state of the output waveform, causing the waveform to start and end at a high or low level.

Note

It is recommended that the process-image register for Q0.0 and Q0.1 be set to a value of zero before enabling PTO or PWM operation.

The pulse train (PTO) function provides a square wave (50% duty cycle) output with user control of the cycle time and the number of pulses. The pulse width modulation (PWM) function provides a continuous, variable duty cycle output with user control of the cycle time and the pulse width.

Each PTO/PWM generator has a control byte (8 bits), a cycle time value and pulse width value (unsigned 16-bit values), and a pulse count value (unsigned 32-bit value). These values are all stored in designated locations of the special memory (SM) area. Once these special memory bit locations have been set up to select the desired operation, the operation is invoked by executing the Pulse Output instruction (PLS). This instruction causes the S7-200 to read the SM locations and program the PTO/PWM generator accordingly.

You can change the characteristics of a PTO or PWM waveform by modifying the desired locations in the SM area (including the control byte), and then executing the PLS instruction.

You can disable the generation of a PTO or PWM waveform at any time by writing zero to the PTO/PWM enable bit of the control byte (SM67.7 or SM77.7), and then executing the PLS instruction.

Note

Default values for all control bits, cycle time, pulse width, and pulse count values are zero.

Note

The PTO/PWM outputs must have a minimum load of at least 10% of rated load to provide crisp transitions from off to on, and from on to off.

PWM Operation

The PWM function provides for variable duty cycle output. The cycle time and the pulse width can be specified in a time base of either microseconds or milliseconds. The cycle time has a range either from 50 microseconds to 65,535 microseconds, or from 2 milliseconds to 65,535 milliseconds. The pulse width time has a range either from 0 microseconds to 65,535 microseconds, or from 0 milliseconds to 65,535 milliseconds. When the pulse width is specified to have a value greater or equal to the cycle time value, the duty cycle of the waveform is 100% and the output is turned on continuously. When the pulse width is specified as 0, the duty cycle of the waveform is 0% and the output is turned off. If a cycle time of less than two time units is specified, the cycle time defaults to two time units.

There are two different ways to change the characteristics of a PWM waveform: with a synchronous update and with an asynchronous update.

- Synchronous Update: If no time base changes are required, then a synchronous update can be performed. With a synchronous update, the change in the waveform characteristics occurs on a cycle boundary, providing for a smooth transition.
- Asynchronous Update: Typically with PWM operation, the pulse width is varied while the cycle time remains constant. Therefore, time base changes are not required. However, if a change in the time base of the PTO/PWM generator is required, then an asynchronous update is used. An asynchronous update causes the PTO/PWM generator to be disabled momentarily, asynchronous to the PWM waveform. This can cause undesirable jitter in the controlled device. For that reason, synchronous PWM updates are recommended. Choose a time base that will work for all of your anticipated cycle time values.

The PWM Update Method bit (SM67.4 or SM77.4) in the control byte is used to specify the update type. Execute the PLS instruction to invoke the changes. Be aware that if the time base is changed, an asynchronous update will occur regardless of the state of the PWM Update Method bit.

PTO Operation

The PTO function provides for the generation of a square wave (50% duty cycle) pulse train with a specified number of pulses. The cycle time can be specified in either microsecond or millisecond increments. The cycle time has a range either from 50 microseconds to 65,535 microseconds, or from 2 milliseconds to 65,535 milliseconds. If the specified cycle time is an odd number, some duty cycle distortion will result. The pulse count has a range from 1 to 4,294,967,295 pulses.

If a cycle time of less than two time units is specified, the cycle time defaults to two time units. If a pulse count of zero is specified, the pulse count defaults to one pulse.

The PTO Idle bit in the status byte (SM66.7 or SM76.7) is provided to indicate the completion of the programmed pulse train. In addition, an interrupt routine can be invoked upon the completion of a pulse train (see Section 9.16 for information about the interrupt and communication instructions). If you are using the multiple segment operation, the interrupt routine will be invoked upon completion of the profile table. See Multiple Segment Pipelining below.

The PTO function allows the chaining or pipelining of pulse trains. When the active pulse train is complete, the output of a new pulse train begins immediately. This allows continuity between subsequent output pulse trains.

This pipelining can be done in one of two ways: in single segment pipelining or in multiple segment pipelining.

Single Segment Pipelining In single segment pipelining, you are responsible for updating the SM locations for the next pulse train. Once the initial PTO segment has been started, you must modify immediately the SM locations as required for the second waveform, and execute the PLS instruction again. The attributes of the second pulse train will be held in a pipeline until the first pulse train is completed. Only one entry at a time can be stored in the pipeline. Once the first pulse train is completed, the output of the second waveform will begin and the pipeline is made available for a new pulse train specification. You can then repeat this process to set up the characteristics of the next pulse train.

Smooth transitions between pulse trains will occur except in the following situations:

- If a change in time base is performed
- If the active pulse train is completed before a new pulse train setup is captured by the execution of the PLS instruction.

If you attempt to load the pipeline while it is full, the PTO overflow bit in the status register (SM66.6 or SM76.6) is set. This bit is initialized to zero on entry to RUN mode. If you want to detect subsequent overflows, you must clear this bit manually after an overflow is detected.

Multiple Segment Pipelining In multiple segment pipelining, the characteristics of each pulse train segment are read automatically by the CPU from a profile table located in V memory. The only SM locations used in this mode are the control byte and the status byte. To select multiple segment operation, the starting V memory offset of the profile table must be loaded (SMW168 or SMW178). The time base can be specified to be either microseconds or milliseconds, but the selection applies to all cycle time values in the profile table, and cannot be changed while the profile is running. Multiple segment operation can then be started by executing the PLS instruction.

Each segment entry is 8 bytes in length, and is composed of a 16-bit cycle time value, a 16-bit cycle time delta value, and a 32-bit pulse count value.

The format of the profile table is shown in Table 9-15. An additional feature available with multiple segment PTO operation is the ability to increase or decrease the cycle time automatically by a specified amount for each pulse. Programming a positive value in the cycle time delta field increases cycle time. Programming a negative value in the cycle time delta field decreases cycle time. A value of zero results in an unchanging cycle time.

If you specify a cycle time delta value that results in an illegal cycle time after a number of pulses, a mathematical overflow condition occurs. The PTO function is terminated, and the output reverts to image register control. In addition, the delta calculation error bit in the status byte (SM66.4 or SM76.4) is set to a one.

If you manually abort a PTO profile in progress, the user abort bit in the status byte (SM66.5 or SM76.5) will be set to one as a result.

While the PTO profile is operating, the number of the currently active segment is available in SMB166 (or SMB176).

Table 9-15 Profile Table Format for Multiple Segment PTO Operation

Byte Offset From Profile Table Start	Profile Segment Number	Description of Table Entries
0		Number of segments (1 to 255); a value of 0 generates a non-fatal error. No PTO output is generated.
1	#1	Initial cycle time (2 to 65535 units of the time base)
3		Cycle time delta per pulse (signed value) (-32768 to 32767 units of the time base)
5		Pulse count (1 to 4294967295)
9	#2	Initial cycle time (2 to 65535 units of the time base)
11		Cycle time delta per pulse (signed value) (-32768 to 32767 units of the time base)
13		Pulse count (1 to 4294967295)
: :	: :	: :

Calculating Profile Table Values

The multiple segment pipelining capability of the PTO/PWM generators can be useful in many applications, particularly in stepper motor control.

The example shown in Figure 9-20 illustrates how to determine the profile table values required to generate an output waveform that accelerates a stepper motor, operates the motor at a constant speed, and then decelerates the motor.

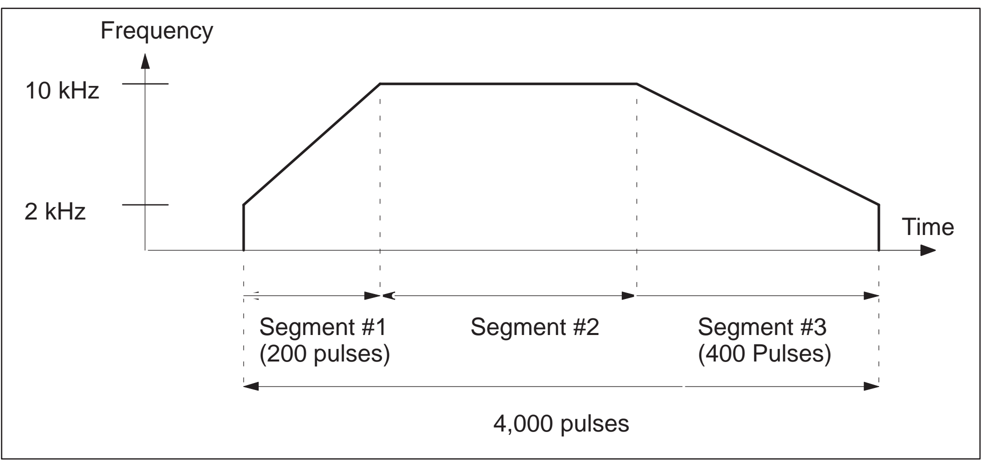


Figure 9-20 Example Frequency vs. Time Diagram for Simple Stepper Motor Application

For this example, it is assumed that 4,000 pulses are required to achieve the desired number of motor revolutions. The starting and final pulse frequency is 2 kHz and the maximum pulse frequency is 10 kHz. Since profile table values are expressed in terms of period (cycle time) instead of frequency, convert the given frequency values into cycle time values. Therefore, the starting and final cycle time is 500 μs and the cycle time corresponding to the maximum frequency is 100 μs.

During the acceleration portion of the output profile, it is desired that the maximum pulse frequency be reached in approximately 200 pulses. It is also assumed that the deceleration portion of the profile should be completed in around 400 pulses.

In this example, a simple formula can be used to determine the delta cycle time value that the PTO/PWM generator uses to adjust the cycle time of each pulse:

delta cycle time = | ending cycle time - initial cycle time | / quantity of pulses

Using this formula, the delta cycle time for the acceleration portion (or segment #1) is calculated to be -2. Likewise, the delta cycle time for the deceleration portion (or segment #3) is calculated to be 1. Since segment #2 is the constant speed portion of the output waveform, the delta cycle time value for that segment is zero.

Assuming that the profile table is located in V memory starting at V500, the table values used to generate the desired waveform are shown in Table 9-16.

Table 9-16 Profile Table Values

V Memory Address	Value
VB500	3 (total number of segments)
VW501	500 (initial cycle time - segment #1)
VW503	-2 (initial cycle time - segment #1)
VW505	200 (number of pulses - segment #1)
VW509	100 (initial cycle time - segment #2)
VW511	0 (delta cycle time - segment #2)
VW513	3400 (number of pulses - segment #2)
VW517	100 initial cycle time - segment #3)
VW519	1 (delta cycle time - segment #3)
VD521	400 (number of pulses - segment #3)

The values of this table can be placed in V memory by using instructions in your program. An alternate method is to define the values of the profile in the data block. An example containing the program instructions to use the multiple segment PTO operation is shown in Figure 9-23.

The cycle time of the last pulse of a segment is not directly specified in the profile, but instead must be calculated (except of course for the case in which the delta cycle time is zero). Knowing the cycle time of a segment's last pulse is useful to determine if the transitions between waveform segments are acceptable. The formula for calculating the cycle time of a segment's last pulse is:

cycle time of last pulse=initial cycle time + (delta cycle time * (number of pulses - 1))

While the simplified example above is useful as an introduction, real applications may require more complicated waveform profiles. Remember that:

- The delta cycle time can be specified only as an integer number of microseconds or milliseconds
- The cycle time modification is performed on each pulse

The effect of these two items is that calculation of the delta cycle time value for a given segment may require an iterative approach. Some flexibility in the value of the ending cycle time or the number of pulses for a given segment may be required.

The duration of a given profile segment can be useful in the process of determining correct profile table values. The length of time to complete a given profile segment can be calculated using the following formula:

duration = no. of pulses *(initial cycle time + ((delta cycle time / 2) * (no. of pulses-1)))

PTO/PWM Control Registers

Table 9-17 describes the registers used to control the PTO/PWM operation. You can use Table 9-18 as a quick reference to determine the value to place in the PTO/PWM control register to invoke the desired operation. Use SMB67 for PTO/PWM 0, and SMB77 for PTO/PWM 1. If you are going to load the new pulse count (SMD72 or SMD82), pulse width (SMW70 or SMW80), or cycle time (SMW68 or SMW78), you should load these values as well as the control register before you execute the PLS instruction. If you are using the multiple segment pulse train operation, you also need to load the starting offset (SMW168 or SMW178) of the profile table and the profile table values before you execute the PLS instruction.

Table 9-17 PTO /PWM Control Registers

Q0.0	Q0.1	Status Byte
SM66.4	SM76.4	PTO profile aborted due to delta calculation error 0 = no error; 1 = aborted
SM66.5	SM76.5	PTO profile aborted due to user command 0 = no abort; 1 = aborted
SM66.6	SM76.6	PTO pipeline overflow/underflow 0 = no overflow; 1 = overflow/underflow
SM66.7	SM76.7	PTO idle 0 = in progress; 1 = PTO idle
Q0.0	**Q0.1**	**Control Byte**
SM67.0	SM77.0	PTO/PWM update cycle time value 0 = no update; 1 = update cycle time
SM67.1	SM77.1	PWM update pulse width time value 0 = no update; 1 = update pulse width
SM67.2	SM77.2	PTO update pulse count value 0 = no update; 1 = update pulse count
SM67.3	SM77.3	PTO/PWM time base select 0 = 1 μs/tick; 1 = 1ms/tick
SM67.4	SM77.4	PWM update method: 0 = asynchronous update, 1 = synchronous update
SM67.5	SM77.5	PTO operation: 0 = single segment operation 1 = multiple segment operation
SM67.6	SM77.6	PTO/PWM mode select 0 = selects PTO; 1 = selects PWM
SM67.7	SM77.7	PTO/PWM enable 0 = disables PTO/PWM; 1 = enables PTO/PWM
Q0.0	**Q0.1**	**Other PTO/PWM Registers**
SMW68	SMW78	PTO/PWM cycle time value (range: 2 to 65535)
SMW70	SMW80	PWM pulse width value (range: 0 to 65535)
SMD72	SMD82	PTO pulse count value (range: 1 to 4294967295)
SMB166	SMB176	Number of segment in progress (used only in multiple segment PTO operation)
SMW168	SMW178	Starting location of profile table, expressed as a byte offset from V0 (used only in multiple segment PTO operation)

Table 9-18 PTO/PWM Control Byte Reference

Control Register (Hex Value)	Result of executing the PLS instruction							
	Enable	Select Mode	PTO Segment Operation	PWM Update Method	Time Base	Pulse Count	Pulse Width	Cycle Time
16#81	Yes	PTO	Single		1 μs/cycle			Load
16#84	Yes	PTO	Single		1 μs/cycle	Load		
16#85	Yes	PTO	Single		1 μs/cycle	Load		Load
16#89	Yes	PTO	Single		1 ms/cycle			Load
16#8C	Yes	PTO	Single		1 ms/cycle	Load		
16#8D	Yes	PTO	Single		1 ms/cycle	Load		Load
16#A0	Yes	PTO	Multiple		1 μs/cycle			
16#A8	Yes	PTO	Multiple		1 ms/cycle			
16#D1	Yes	PWM		Synchronous	1 μs/cycle			Load
16#D2	Yes	PWM		Synchronous	1 μs/cycle		Load	
16#D3	Yes	PWM		Synchronous	1 μs/cycle		Load	Load
16#D9	Yes	PWM		Synchronous	1 ms/cycle			Load
16#DA	Yes	PWM		Synchronous	1 ms/cycle		Load	
16#DB	Yes	PWM		Synchronous	1 ms/cycle		Load	Load

PTO/PWM Initialization and Operation Sequences

Descriptions of the initialization and operation sequences follow. They can help you better understand the operation of PTO and PWM functions. The pulse output Q0.0 is used throughout these sequence descriptions. The initialization descriptions assume that the S7-200 has just been placed in RUN mode, and for that reason the first scan memory bit is true. If this is not the case, or if the PTO/PWM function must be re-initialized, you can call the initialization routine using a condition other than the first scan memory bit.

PWM Initialization

To initialize the PWM for Q0.0, follow these steps:

1. Use the first scan memory bit (SM0.1) to initialize the output to 0, and call the subroutine that you need in order to perform the initialization operations. When you use the subroutine call, subsequent scans do not make the call to the subroutine. This reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB67 with a value of 16#D3 for PWM using microsecond increments (or 16#DB for PWM using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PWM operation, select either microsecond or millisecond increments, and set the update pulse width and cycle time values.
3. Load SMW68 (word size value) with the desired cycle time.
4. Load SMW70 (word size value) with the desired pulse width.
5. Execute the PLS instruction so that the S7-200 programs the PTO/PWM generator.
6. Load SMB67 with a value of 16#D2 for microsecond increments (or 16#DA for millisecond increments). This preloads a new control byte value for subsequent pulse width changes.
7. Exit the subroutine.

Changing the Pulse Width for PWM Outputs

To change the pulse width for PWM outputs in a subroutine, follow these steps. (It is assumed that SMB67 was preloaded with a value of 16#D2 or 16#DA.)

1. Call a subroutine to load SMW70 (word size value) with the desired pulse width.
2. Execute the PLS instruction to cause the S7-200 to program the PTO/PWM generator.
3. Exit the subroutine.

PTO Initialization - Single Segment Operation

To initialize the PTO, follow these steps:

1. Use the first scan memory bit (SM0.1) to initialize the output to 0, and call the subroutine that you need to perform the initialization operations. This reduces scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB67 with a value of 16#85 for PTO using microsecond increments (or 16#8D for PTO using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PTO operation, select either microsecond or millisecond increments, and set the update pulse count and cycle time values.
3. Load SMW68 (word size value) with the desired cycle time.
4. Load SMD72 (double word size value) with the desired pulse count.
5. This is an optional step. If you want to perform a related function as soon as the pulse train output is complete, you can program an interrupt by attaching the pulse train complete event (Interrupt Category 19) to an interrupt subroutine, using the ATCH instruction, and executing the global interrupt enable instruction ENI. Refer to Section 9.16 for complete details on interrupt processing.
6. Execute the PLS instruction to cause the S7-200 to program the PTO/PWM generator.
7. Exit the subroutine.

Changing the PTO Cycle Time - Single Segment Operation

To change the PTO Cycle Time in an interrupt routine or subroutine when using single segment PTO operation, follow these steps:

1. Load SMB67 with a value of 16#81 for PTO using microsecond increments (or 16#89 for PTO using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PTO operation, select either microsecond or millisecond increments, and set the update cycle time value.
2. Load SMW68 (word size value) with the desired cycle time.
3. Execute the PLS instruction to cause the S7-200 to program the PTO/PWM generator. The CPU must complete any PTO that is in process before output of the PTO waveform with the updated cycle time begins.
4. Exit the interrupt routine or the subroutine.

Changing the PTO Pulse Count - Single Segment Operation

To change the PTO Pulse Count in an interrupt routine or a subroutine when using single segment PTO operation, follow these steps:

1. Load SMB67 with a value of 16#84 for PTO using microsecond increments (or 16#8C for PTO using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PTO operation, select either microsecond or millisecond increments, and set the update pulse count value.
2. Load SMD72 (double word size value) with the desired pulse count.
3. Execute the PLS instruction to cause the S7-200 to program the PTO/PWM generator. The CPU must complete any PTO that is in process before output of the waveform with the updated pulse count begins.
4. Exit the interrupt routine or the subroutine.

Changing the PTO Cycle Time and the Pulse Count - Single Segment Operation

To change the PTO Cycle Time and Pulse Count in an interrupt routine or a subroutine when using single segment PTO operation, follow these steps:

1. Load SMB67 with a value of 16#85 for PTO using microsecond increments (or 16#8D for PTO using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PTO operation, select either microsecond or millisecond increments, and set the update cycle time and pulse count values.
2. Load SMW68 (word size value) with the desired cycle time.
3. Load SMD72 (double word size value) with the desired pulse count.
4. Execute the PLS instruction so that the S7-200 programs the PTO/PWM generator. The CPU must complete any PTO that is in process before output of the waveform with the updated pulse count and cycle time begins.
5. Exit the interrupt routine or the subroutine.

PTO Initialization - Multiple Segment Operation

To initialize the PTO, follow these steps:

1. Use the first scan memory bit (SM0.1) to initialize the output to 0, and call the subroutine that you need to perform the initialization operations. This reduces the scan time execution and provides a more structured program.
2. In the initialization subroutine, load SMB67 with a value of 16#A0 for PTO using microsecond increments (or 16#A8 for PTO using millisecond increments). These values set the control byte to enable the PTO/PWM function, select PTO and multiple segment operation, and select either microsecond or millisecond increments
3. Load SMW168 (word size value) with the starting V memory offset of the profile table.
4. Set up the segment values in the profile table. Ensure that the Number of Segment field (the first byte of the table) is correct.
5. This is an optional step. If you want to perform a related function as soon as the PTO profile is complete, you can program an interrupt by attaching the pulse train complete event (Interrupt Category 19) to an interrupt subroutine. Use the ATCH instruction, and execute the global interrupt enable instruction ENI. Refer to Section 9.16 for complete details on interrupt processing.
6. Execute the PLS instruction. The S7-200 programs the PTO/PWM generator.
7. Exit the subroutine.

Example of Pulse Width Modulation

Figure 9-21 shows an example of the Pulse Width Modulation.

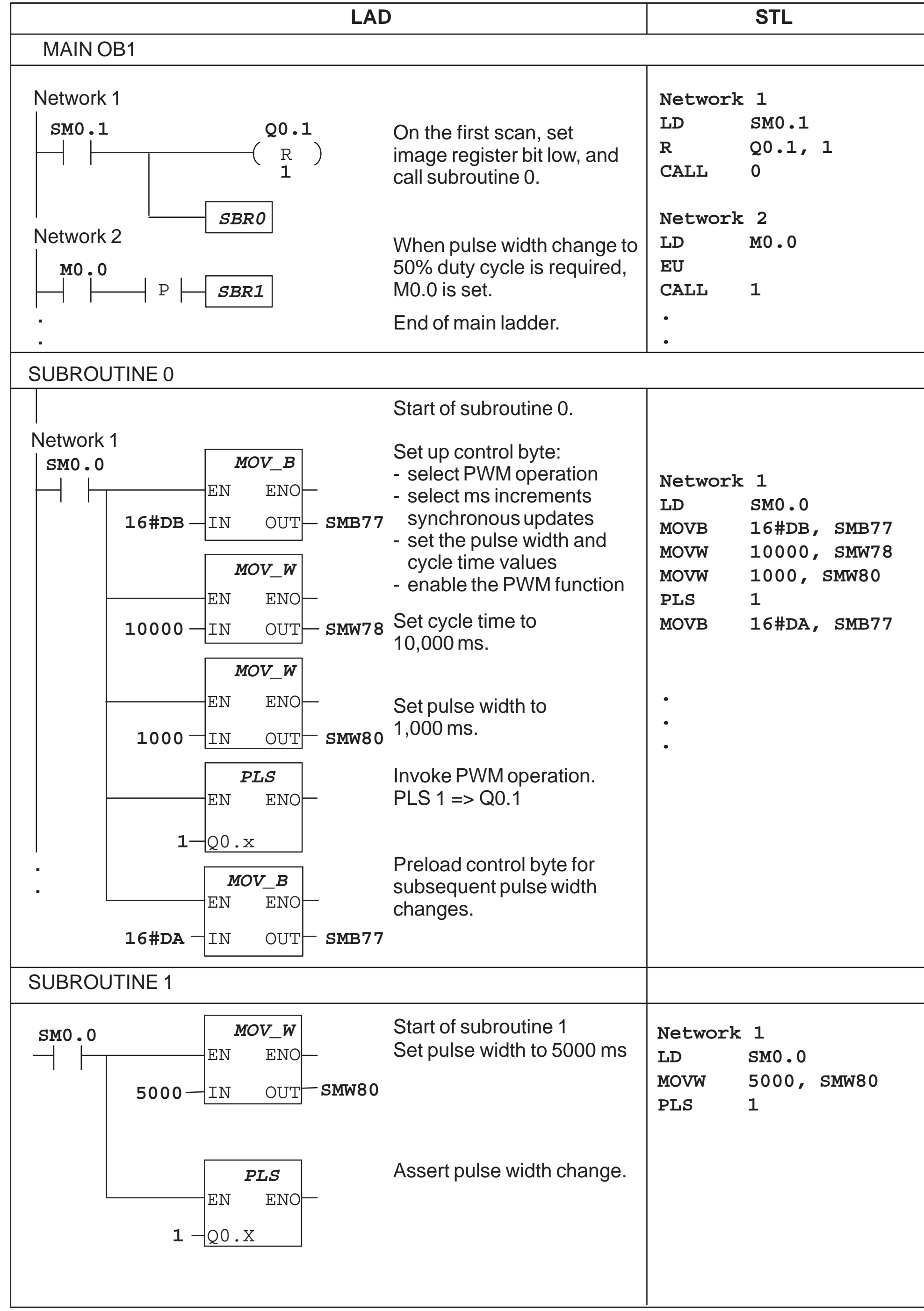


Figure 9-21 Example of High-Speed Output Using Pulse Width Modulation

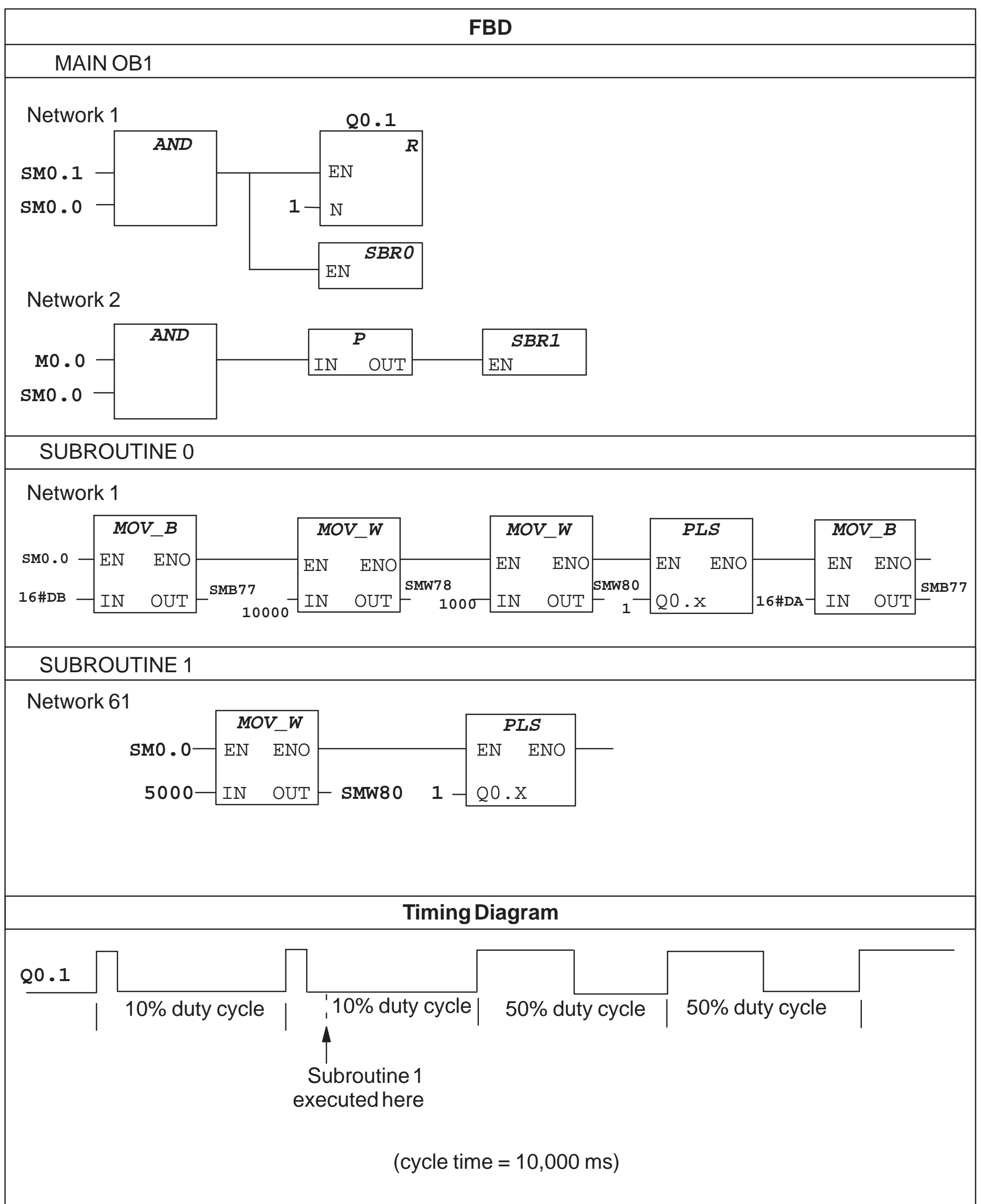


Figure 9-21 Example of High-Speed Output Using Pulse Width Modulation (continued)

Example of Pulse Train Output Using Single Segment Operation

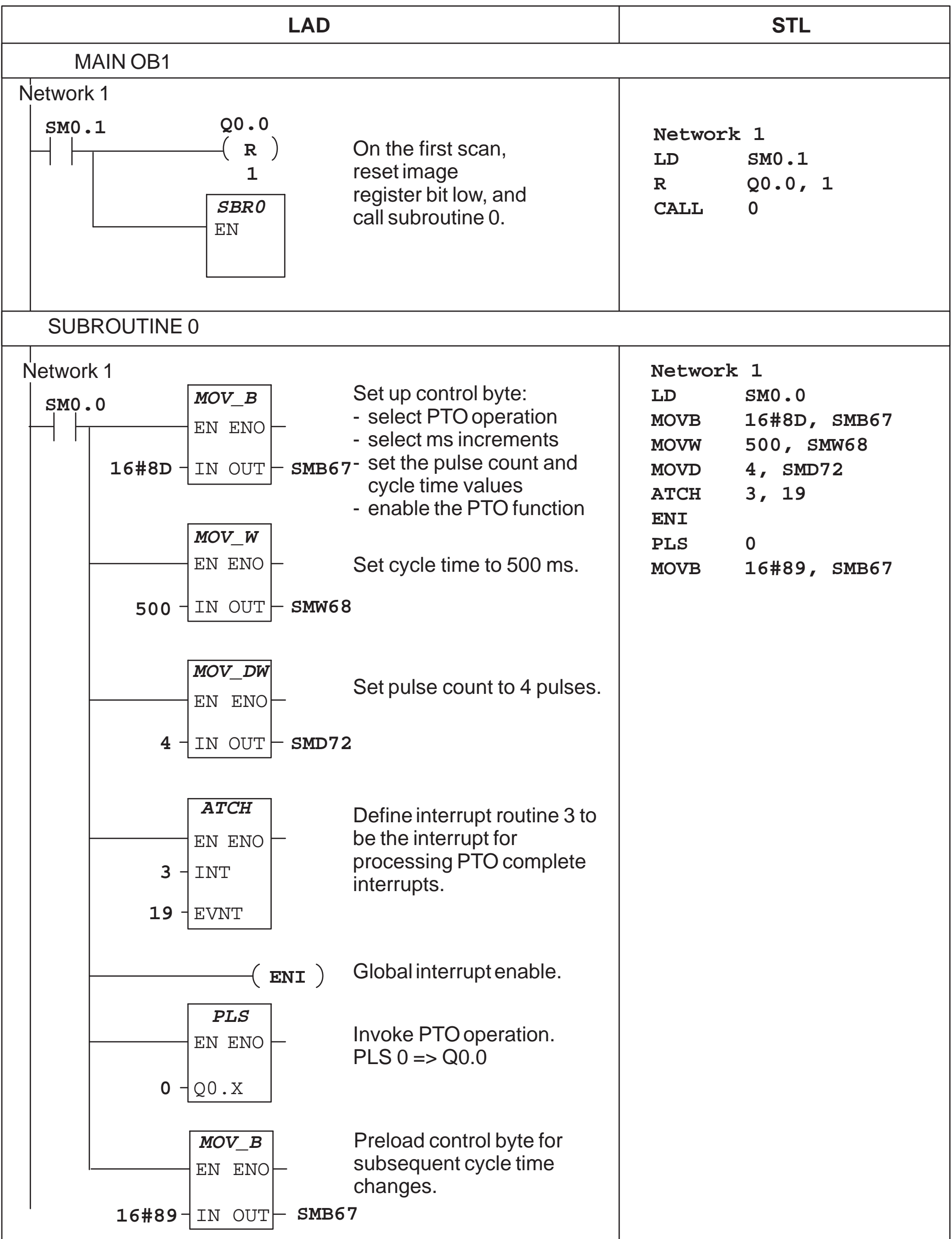


Figure 9-22 Example of a Pulse Train Output Using Single Segment Operation in SM Memory

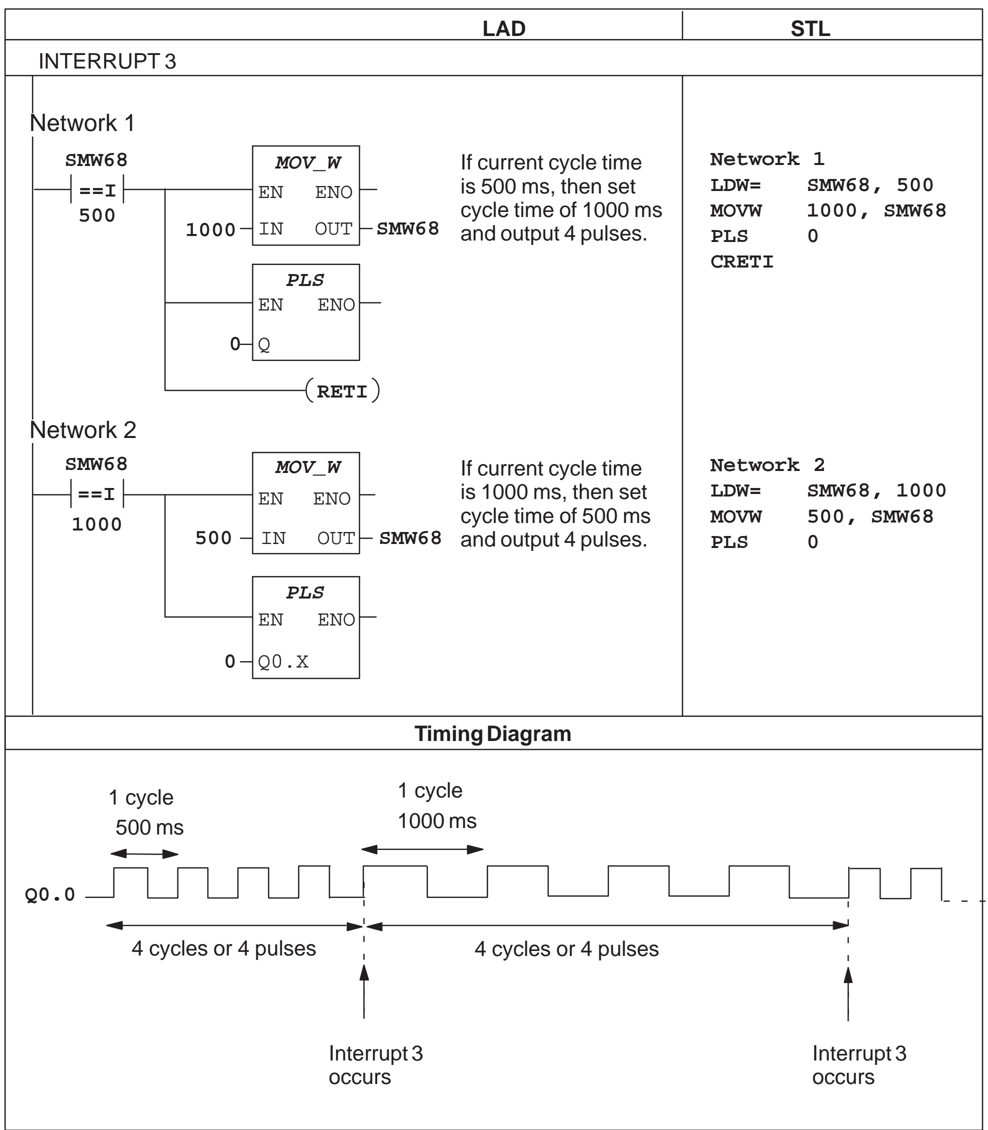


Figure 9-22 Example of a Pulse Train Output Using Single Segment Operation (continued)

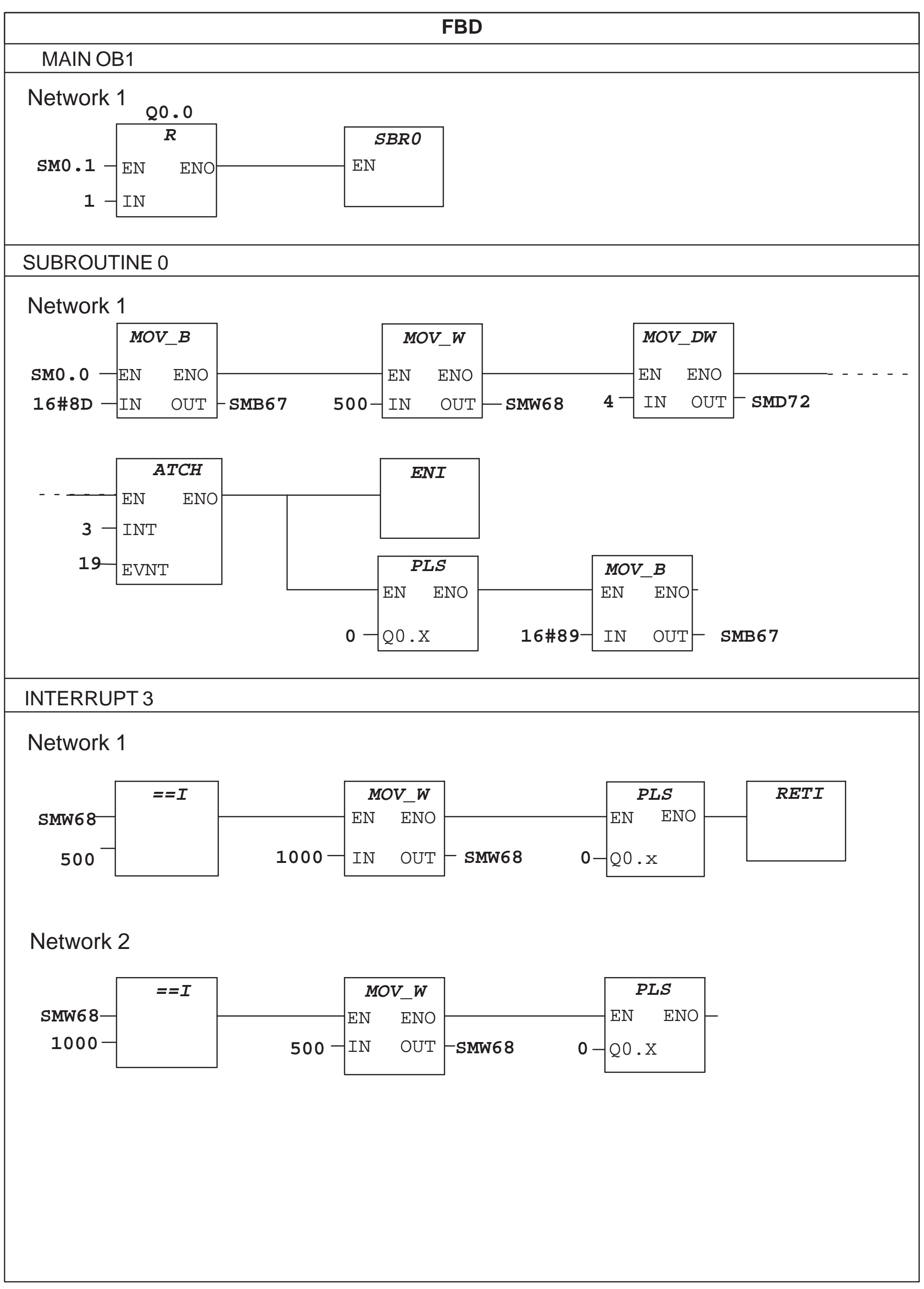


Figure 9-22 Example of a Pulse Train Output Using Single Segment Operation (continued)

Example of Pulse Train Output Using Multiple Segment Operation

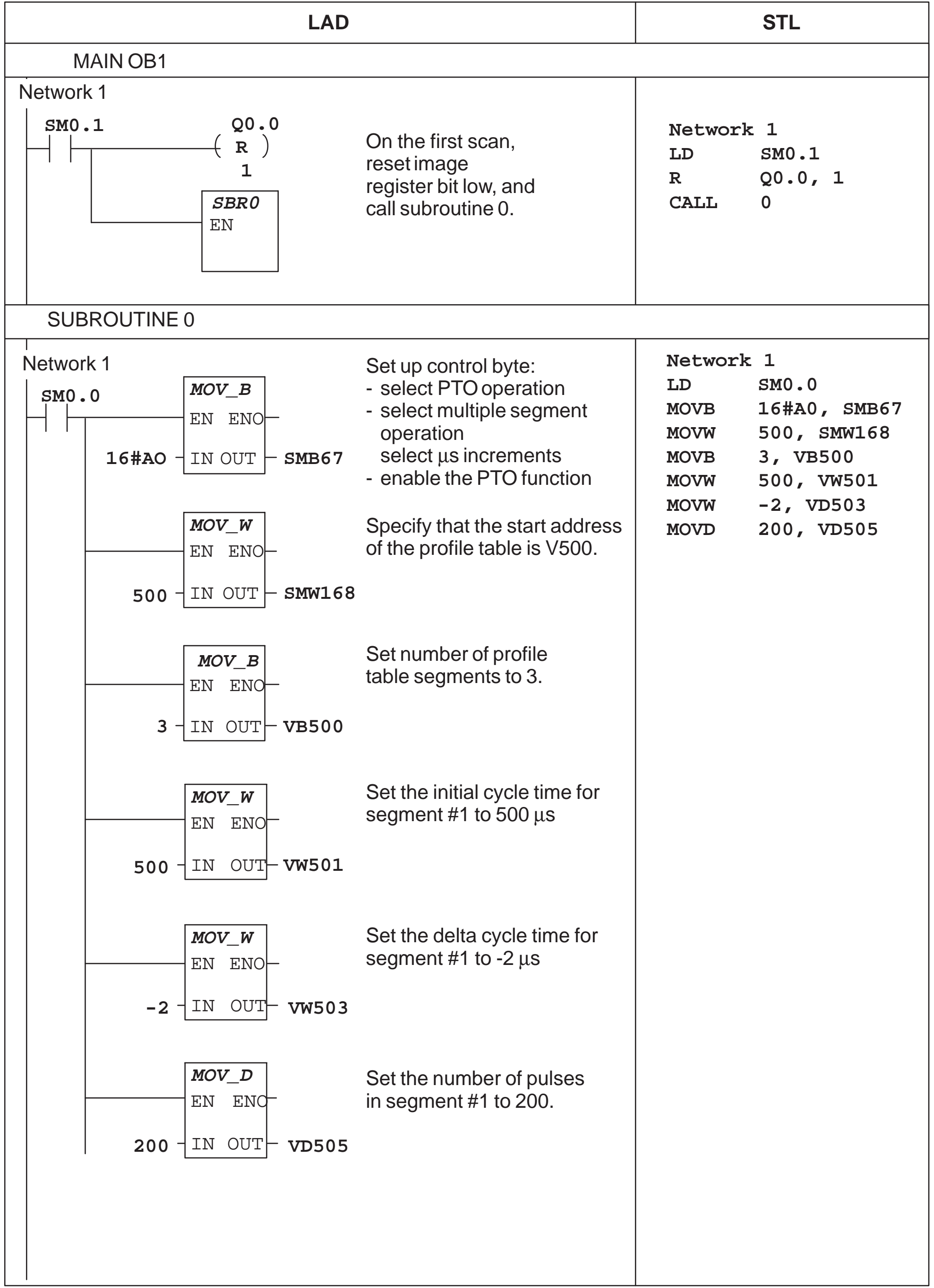


Figure 9-23 Example of a Pulse Train Output Using Multiple Segment Operation

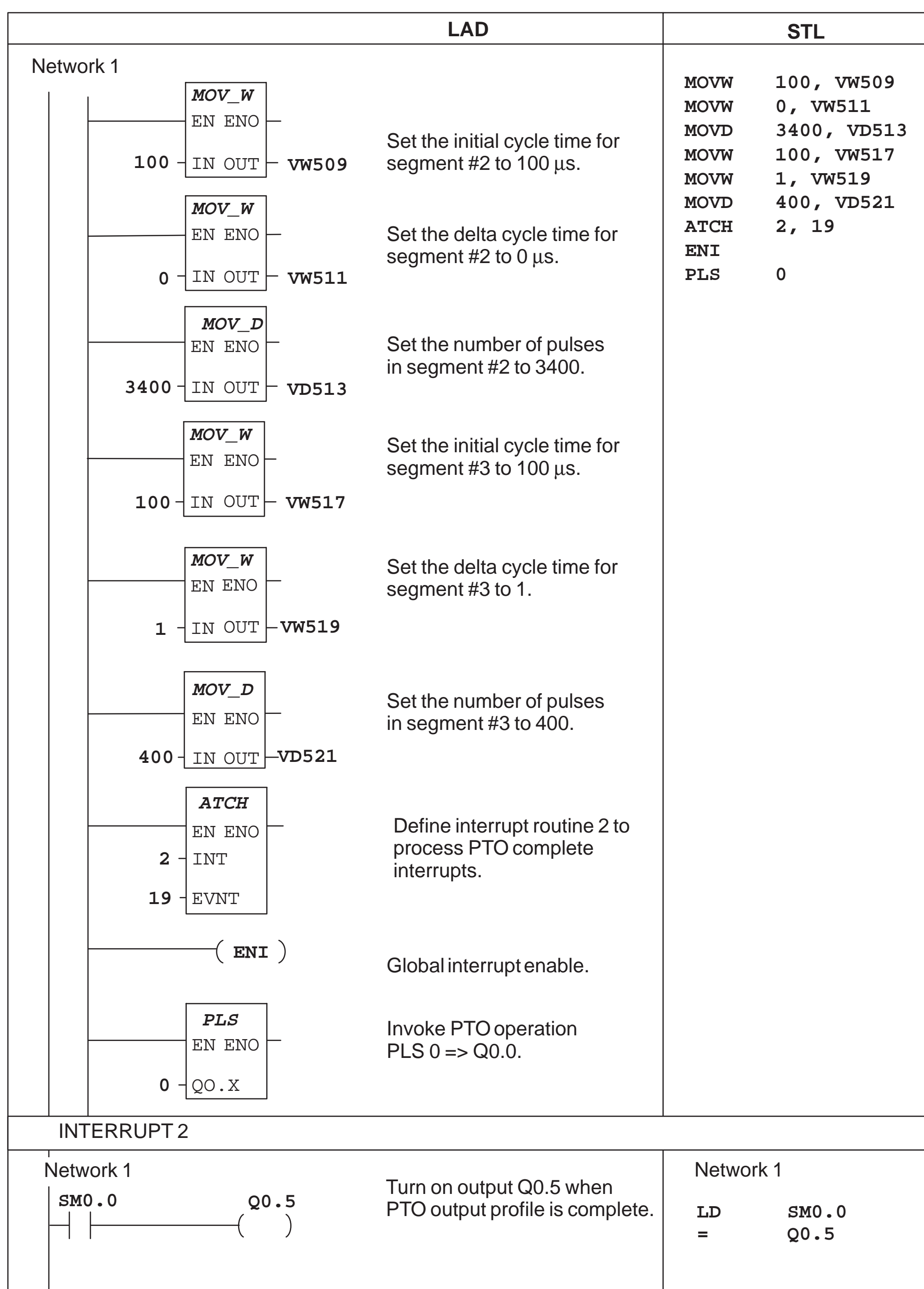
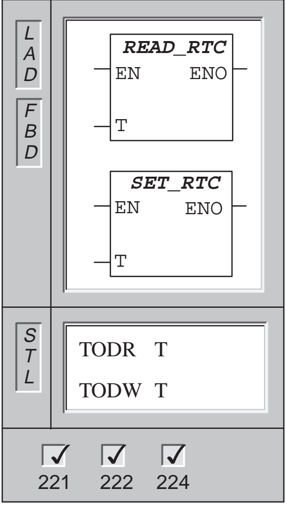


Figure 9-74 Example of Pulse Train Output Using Multiple Segment Operation (continued)

9.7 SIMATIC Clock Instructions

Read Real-Time Clock, Set Real-Time Clock



The **Read Real-Time Clock** instruction reads the current time and date from the clock and loads it in an 8-byte buffer (starting at address T).

The **Set Real-Time Clock** instruction writes the current time and date loaded in an 8-byte buffer (starting at address T) to the clock.

In STL, the TODR and TODW instructions are represented as Time of Day Read (TODR) and Time of Day Write (TODW).

TODR: Error conditions that set ENO = 0:
SM4.3 (run-time), 0006 (indirect address),
000C (clock cartridge not present)

TODW: Error conditions that set ENO = 0:
SM 4.3 (run-time), 0006 (indirect address),
0007 (TOD data error), 000C (clock cartridge not present)

Inputs/Outputs	Operands	Data Types
T	VB, IB, QB, MB, SMB, SB, LB, *VD, *AC, *LD	BYTE

Figure 9-24 shows the format of the time buffer (T).

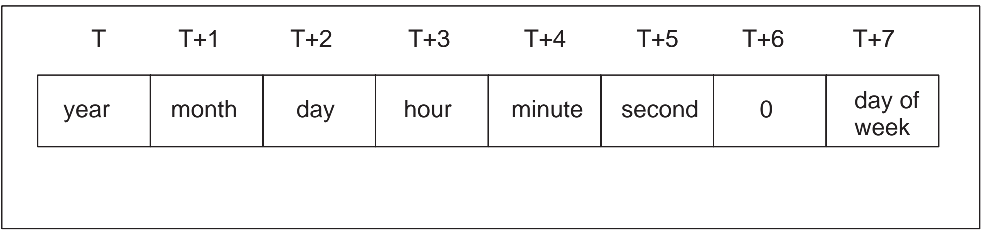


Figure 9-24 Format of the Time Buffer

The time-of-day clock initializes the following date and time after extended power outages or memory has been lost:

Date:	01-Jan-90
Time:	00:00:00
Day of Week	Sunday

The time-of-day clock in the S7-200 uses only the least significant two digits for the year, so for the year 2000, the year will be represented as 00 (it will go from 99 to 00).

You must code all date and time values in BCD format (for example, 16#97 for the year 1997). Use the following data formats:

Year/Month	yymm	yy -	0 to 99	mm -	1 to 12
Day/Hour	ddhh	dd -	1 to 31	hh -	0 to 23
Minute/Second	mmss	mm -	0 to 59	ss -	0 to 59
Day of week	d	d -	0 to 7	1 =	Sunday
				0 =	disables day of week (remains 0)

Note

The S7-200 CPU does not perform a check to verify that the day of week is correct based upon the date. Invalid dates, such as February 30, may be accepted. You should ensure that the date you enter is correct.

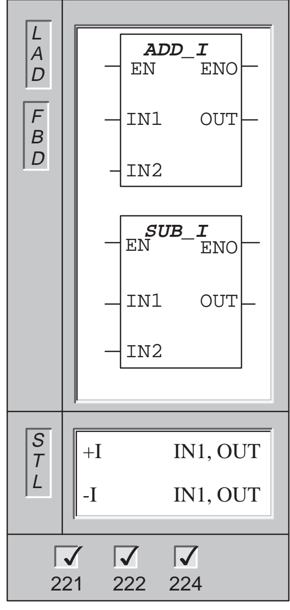
Do not use the TODR/TODW instruction in both the main program and in an interrupt routine. A TODR/TODW instruction in an interrupt routine which attempts to execute while another TODR/TODW instruction is in process will not be executed. SM4.3 is set indicating that two simultaneous accesses to the clock were attempted (non-fatal error 0007).

The S7-200 PLC does not use the year information in any way and will not be affected by the century rollover (year 2000). However, user programs that use arithmetic or compares with the year’s value must take into account the two-digit representation and the change in century.

Leap year is correctly handled through year 2096.

9.8 SIMATIC Integer Math Instructions

Add Integer and Subtract Integer



The **Add Integer** and **Subtract Integer** instructions add or subtract two 16-bit integers and produce a 16-bit result (OUT).

In LAD and FBD: IN1 + IN2 = OUT
IN1 - IN2 = OUT

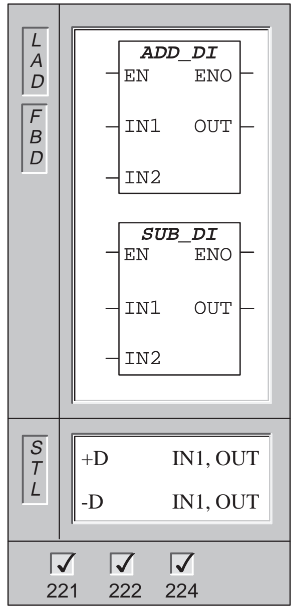
In STL: IN1 + OUT = OUT
OUT - IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

Inputs/Outputs	Operands	Data Types
IN1, IN2	VW, IW, QW, MW, SW, SMW, LW, AIW, T, C, AC, Constant, *VD, *AC, *LD	INT
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	INT

Add Double Integer and Subtract Double Integer



The **Add Double Integer** and **Subtract Double Integer** instructions add or subtract two 32-bit integers, and produce a 32-bit result (OUT).

In LAD and FBD: IN1 + IN2 = OUT
IN1 - IN2 = OUT

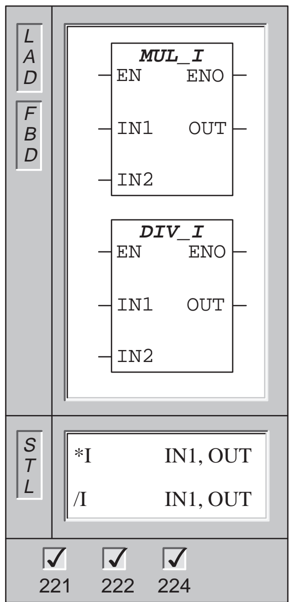
In STL: IN1 + OUT = OUT
OUT - IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

Inputs/Outputs	Operands	Data Types
IN1, IN2	VD, ID, QD, MD, SMD, SD, LD, AC, HC, Constant, *VD, *AC, *LD	DINT
OUT	VD, ID, QD, MD, SM, SD, LD, AC, *VD, *AC, *LD	DINT

Multiply Integer and Divide Integer



The **Multiply Integer** instruction multiplies two 16-bit integers and produces a 16-bit product.

The **Divide Integer** instruction divides two 16-bit integers and produces a 16-bit quotient. No remainder is kept.

The overflow bit is set if the result is greater than a word output.

In LAD and FBD: IN1 ∗ IN2 = OUT
IN1 / IN2 = OUT

In STL: IN1 ∗ OUT = OUT
OUT / IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM1.3 (divide-by-zero), SM4.3 (run-time), 0006 (indirect address)

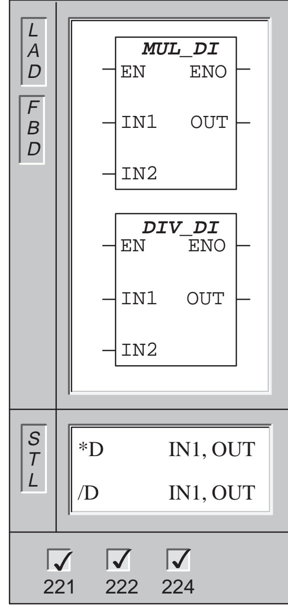
These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative); SM1.3 (divide-by-zero)

If SM1.1 (overflow) is set during a multiply or divide operation, the output is not written and all other math status bits are set to zero.

If SM1.3 (divide by zero) is set during a divide operation, then the other math status bits are left unchanged and the original input operands are not altered. Otherwise, all supported math status bits contain valid status upon completion of the math operation.

Inputs/Outputs	Operands	Data Types
IN1, IN2	VW, IW, QW, MW, SW, SMW, LW, AIW, T, C, AC, Constant, *VD, *AC, *LD	INT
OUT	VW, QW, IW, MW, SW, SMW, LW, T, C, AC, *VD, *LD, *AC	INT

Multiply Double Integer and Divide Double Integer



The **Multiply Double Integer** instruction multiplies two 32-bit integers and produces a 32-bit product.

The **Divide Double Integer** instruction divides two 32-bit integers and produces a 32-bit quotient. No remainder is kept.

In LAD and FBD: IN1 ∗ IN2 = OUT
IN1 / IN2 = OUT

In STL: IN1 ∗ OUT = OUT
OUT / IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM1.3 (divide-by-zero), SM4.3 (run-time), 0006 (indirect address)

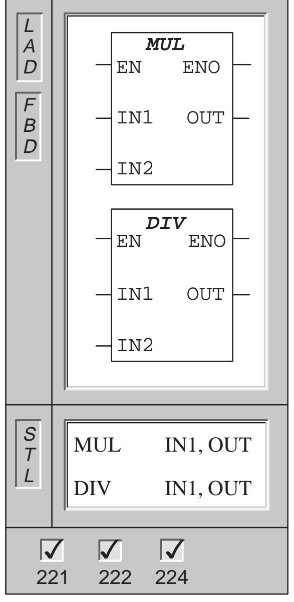
These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative); SM1.3 (divide-by-zero)

If SM1.1 (overflow) is set during a multiply or divide operation, the output is not written and all other math status bits are set to zero.

If SM1.3 (divide by zero) is set during a divide operation, then the other math status bits are left unchanged and the original input operands are not altered. Otherwise, all supported math status bits contain valid status upon completion of the math operation.

Inputs/Outputs	Operands	Data Types
IN1, IN2	VD, ID, QD, MD, SMD, SD, LD, HC, AC, Constant, *VD, *AC, *LD	DINT
OUT	VD, ID, QD, MD, SMD, SD, LD, AC, *VD, *LD, *AC	DINT

Multiply Integer To Double Integer and Divide Integer to Double Integer



The **Multiply Integer to Double Integer** instruction multiplies two 16-bit integers and produces a 32-bit product.

The **Divide Integer to Double Integer** instruction divides two 16-bit integers and produces a 32-bit result consisting of a 16-bit remainder (most-significant) and a 16-bit quotient (least-significant).

In the STL Multiply instruction, the least-significant word (16 bits) of the 32-bit OUT is used as one of the factors.

In the STL Divide instruction, the least-significant word (16 bits) of the 32-bit OUT is used as the dividend.

In LAD and FBD: IN1 ∗ IN2 = OUT
IN1 / IN2 = OUT

In STL: IN1 ∗ OUT = OUT
OUT / IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM1.3 (divide-by-zero), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative); SM1.3 (divide-by-zero)

If SM1.3 (divide by zero) is set during a divide operation, then the other math status bits are left unchanged and the original input operands are not altered. Otherwise, all supported math status bits contain valid status upon completion of the math operation.

Inputs/Outputs	Operands	Data Types
IN1, IN2	VW, IW, QW, MW, SW, SMW, LW, AC, AIW, T, C, Constant, *VD, *AC, *LD	INT
OUT	VD, ID, QD, MD, SMD, SD, LD, AC, *VD, *LD, *AC	DINT

Math Examples

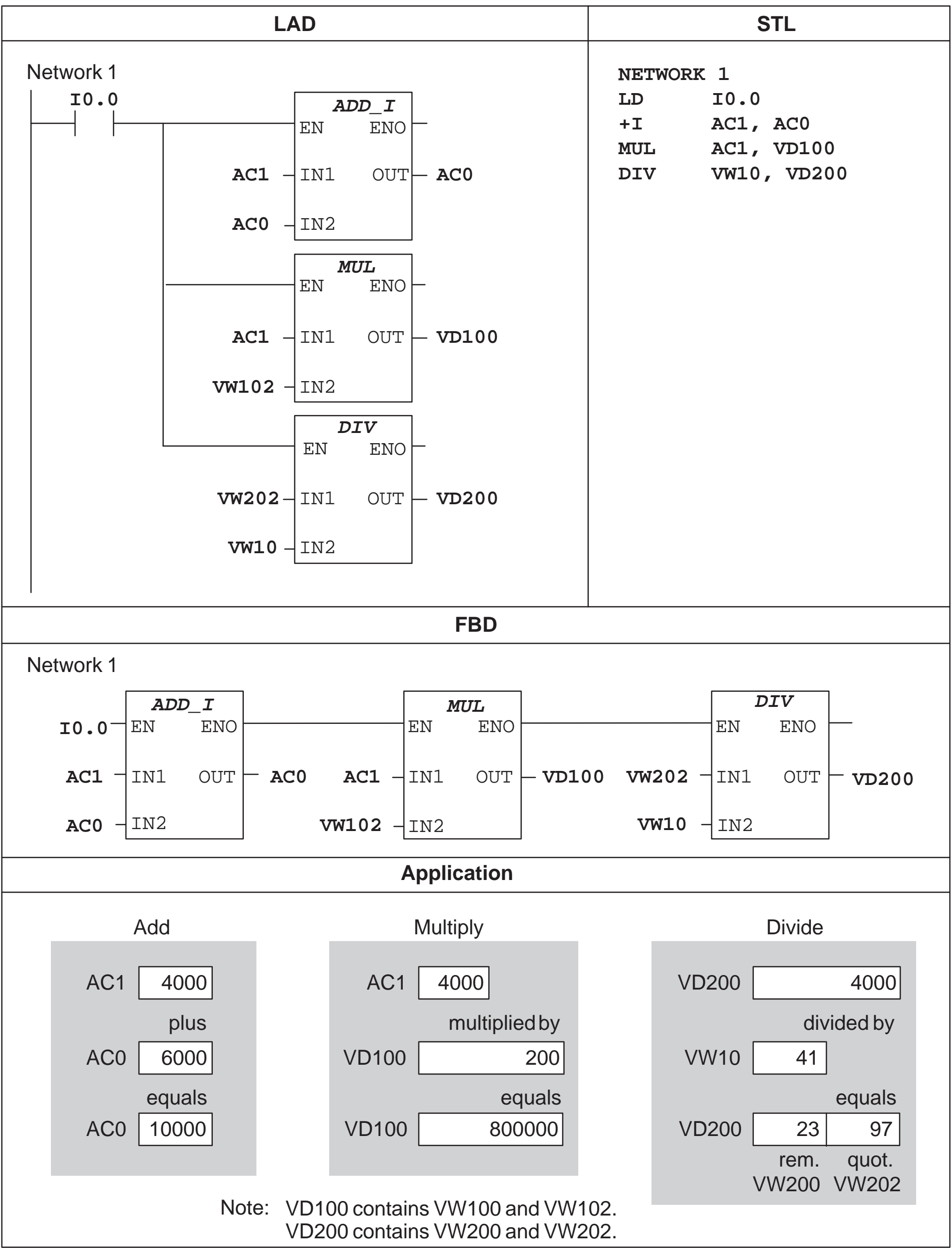
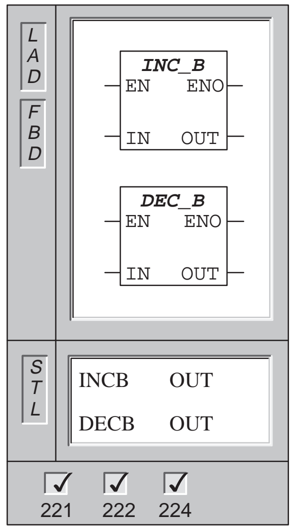


Figure 9-25 Examples of Integer Math Instructions for LAD, STL, and FBD

Increment Byte and Decrement Byte



The **Increment Byte** and **Decrement Byte** instructions add or subtract 1 to or from the input byte (IN) and place the result into the variable specified by OUT.

Increment and decrement byte operations are unsigned.

In LAD and FBD: IN + 1 = OUT
IN - 1 = OUT

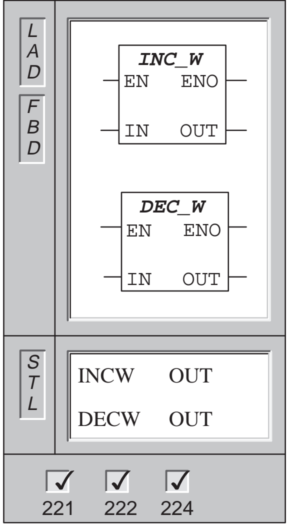
In STL: OUT+ 1 = OUT
OUT - 1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VB, IB, QB, MB, SB, SMB, LB, AC,*VD, *AC, *LD	BYTE

Increment Word and Decrement Word



The **Increment Word** and **Decrement Word** instructions add or subtract 1 to or from the input word (IN) and place the result in OUT.

Increment and decrement word operations are signed (16#7FFF > 16#8000).

In LAD and FBD: IN + 1 = OUT
IN - 1 = OUT

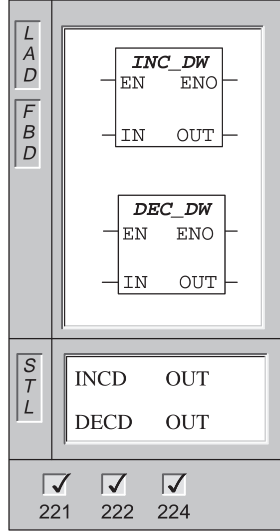
In STL: OUT + 1 = OUT
OUT - 1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, AC, AIW, LW, T, C, Constant, *VD, *AC, *LD	INT
OUT	VW, IW, QW, MW, SW, SMW, LW, AC, T, C, *VD, *AC, *LD	INT

Increment Double Word and Decrement Double Word



The **Increment Double Word** and **Decrement Double Word** instructions add or subtract 1 to or from the input double word (IN) and place the result in OUT.

In LAD and FBD: IN + 1 = OUT
IN - 1 = OUT

Increment and decrement double word operations are signed (16#7FFFFFFF > 16#80000000).

In STL: OUT + 1 = OUT
OUT - 1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SD, SMD, LD, AC, HC, Constant, *VD, *AC, *LD	DINT
OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	DINT

Increment, Decrement Example

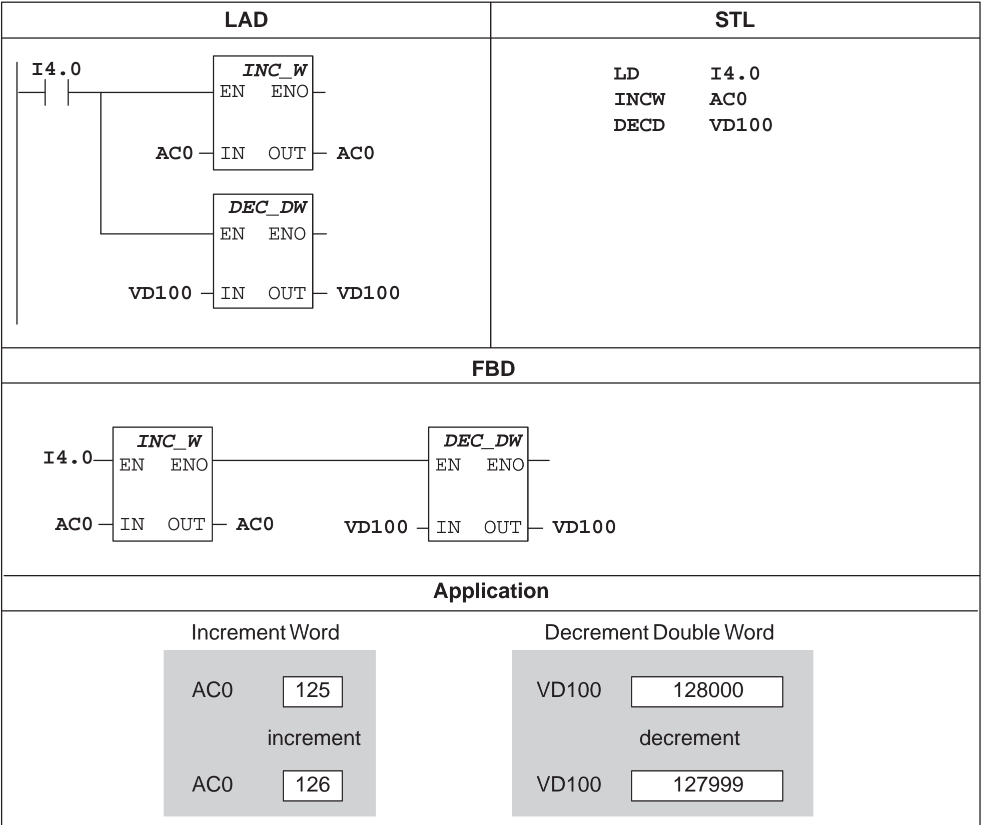
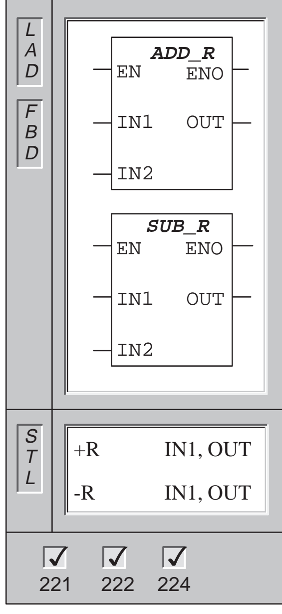


Figure 9-26 Example of Increment/Decrement Instructions for LAD, STL, and FBD

9.9 SIMATIC Real Math Instructions

Add Real, Subtract Real



The **Add Real** and **Subtract Real** instructions add or subtract two 32-bit real numbers and produce a 32-bit real number result (OUT).

In LAD and FBD: IN1 + IN2 = OUT
IN1 - IN2 = OUT

In STL: IN1 + OUT = OUT
OUT - IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

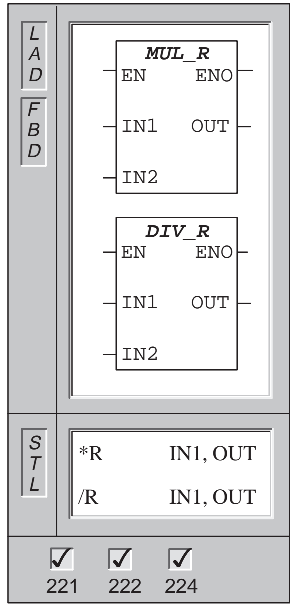
SM1.1 is used to indicate overflow errors and illegal values. If SM1.1 is set, then the status of SM1.0 and SM1.2 is not valid and the original input operands are not altered. If SM1.1 is not set, then the math operation has completed with a valid result and SM1.0 and SM1.2 contain valid status.

Inputs/Outputs	Operands	Data Types
IN1, IN2	VD, ID, QD, MD, SD, SMD, AC, LD, Constant, *VD, *AC, *LD	REAL
OUT	VD, ID, QD, MD, SD, SMD, AC, LD, *VD, *AC, *LD	REAL

Note

Real or floating-point numbers are represented in the format described in the ANSI/IEEE 754-1985 standard (single-precision). Refer to the standard for more information.

Multiply Real, Divide Real



The **Multiply Real** instruction multiplies two 32-bit real numbers, and produces a 32-bit real number result (OUT).

The **Divide Real** instruction divides two 32-bit real numbers, and produces a 32-bit real number quotient.

In LAD and FBD: IN1 ∗ IN2 = OUT
IN1/ IN2 = OUT

In STL: IN1 ∗ OUT = OUT
OUT / IN1 = OUT

Error conditions that set ENO = 0: SM1.1 (overflow), SM1.3 (divide-by-zero), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow or illegal value generated during the operation or illegal input parameter found); SM1.2 (negative); SM1.3 (divide-by-zero)

If SM1.3 is set during a divide operation, then the other math status bits are left unchanged and the original input operands are not altered. SM1.1 is used to indicate overflow errors and illegal values. If SM1.1 is set, then the status of SM1.0 and SM1.2 is not valid and the original input operands are not altered. If SM1.1 and SM1.3 (during a divide operation) are not set, then the math operation has completed with a valid result and SM1.0 and SM1.2 contain valid status.

Inputs/Outputs	Operands	Data Types
IN1, IN2	VD, ID, QD, MD, SMD, SD, LD, AC, Constant, *VD, *AC, *LD	REAL
OUT	VD, ID, QD, MD, SMD, SD, LD, AC, *VD, *AC, *LD	REAL

Note

Real or floating-point numbers are represented in the format described in the ANSI/IEEE 754-1985 standard (single-precision). Refer to the standard for more information.

Math Examples

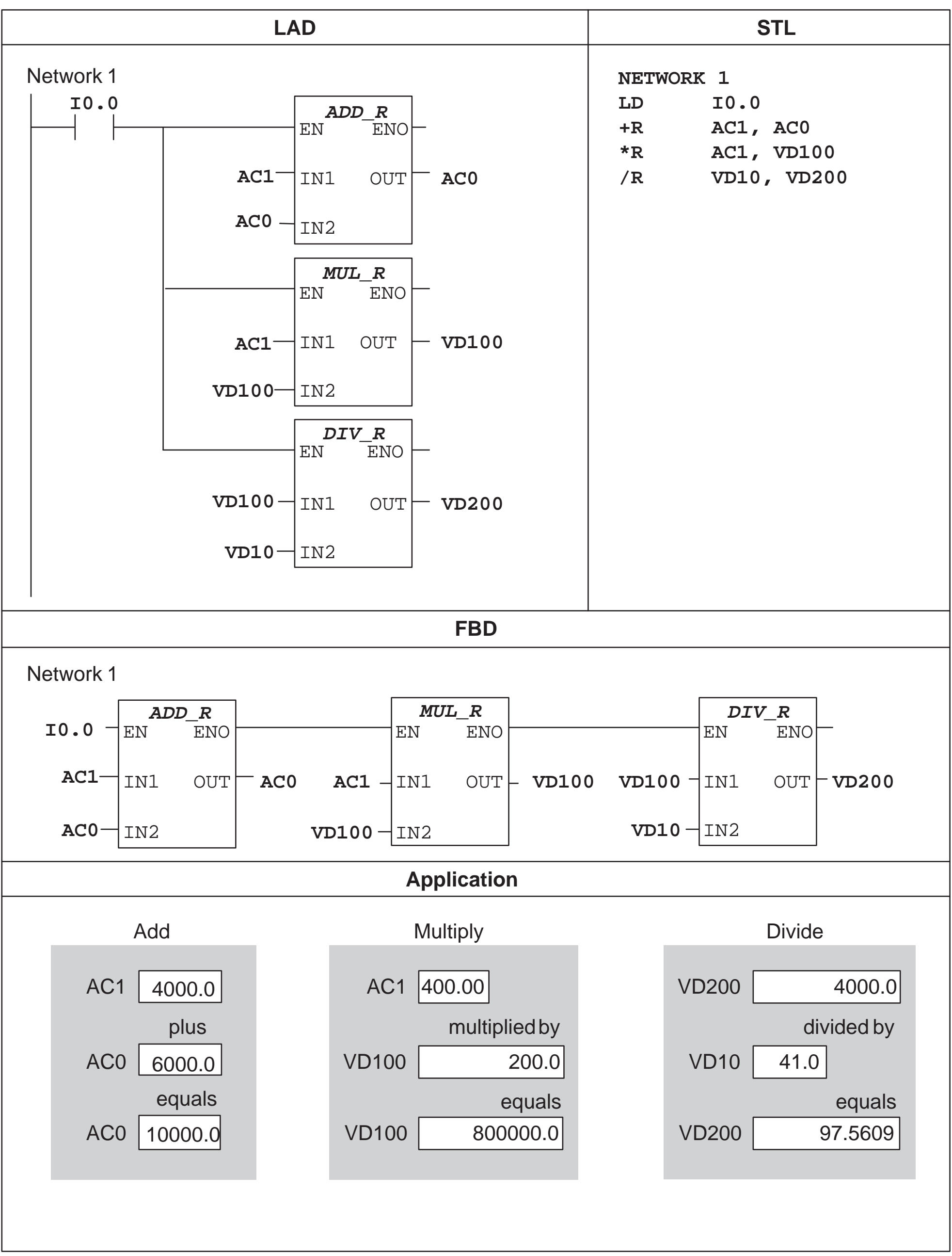
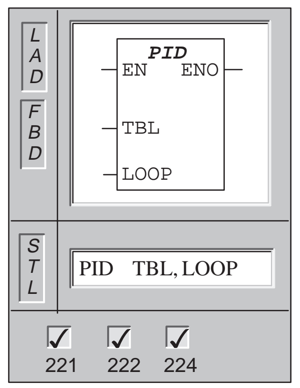


Figure 9-27 Examples of Real Math Instructions for LAD, STL, and FBD

PID Loop



The **PID Loop** instruction executes a PID loop calculation on the referenced LOOP based on the input and configuration information in Table (TBL).

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

This instruction affects the following Special Memory bits: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
TBL	VB	BYTE
LOOP	Constant (0 to 7)	BYTE

The PID loop instruction (Proportional, Integral, Derivative Loop) is provided to perform the PID calculation. The top of the logic stack (TOS) must be ON (power flow) to enable the PID calculation. The instruction has two operands: a TABLE address which is the starting address of the loop table and a LOOP number which is a constant from 0 to 7. Eight PID instructions can be used in a program. If two or more PID instructions are used with the same loop number (even if they have different table addresses), the PID calculations will interfere with one another and the output will be unpredictable.

The loop table stores nine parameters used for controlling and monitoring the loop operation and includes the current and previous value of the process variable, the setpoint, output, gain, sample time, integral time (reset), derivative time (rate), and the integral sum (bias).

To perform the PID calculation at the desired sample rate, the PID instruction must be executed either from within a timed interrupt routine or from within the main program at a rate controlled by a timer. The sample time must be supplied as an input to the PID instruction via the loop table.

Using the PID Wizard in STEP 7-Micro/WIN 32

STEP 7-Micro/WIN 32 offers the PID Wizard to guide you in defining a PID algorithm for a closed-loop control process. Select the menu command **Tools Instruction Wizard**, and then select PID from the Instruction Wizard window.

PID Algorithm

In steady state operation, a PID controller regulates the value of the output so as to drive the error (e) to zero. A measure of the error is given by the difference between the setpoint (SP) (the desired operating point) and the process variable (PV) (the actual operating point). The principle of PID control is based upon the following equation that expresses the output, M(t), as a function of a proportional term, an integral term, and a differential term:

$M(t)$	=	$K_C * e$	+	$K_C \int_0^t e\,dt + M_{initial}$	+	$K_C * de/dt$
output	=	proportional term	+	integral term	+	differential term

where:

$M(t)$	is the loop output as a function of time
K_C	is the loop gain
e	is the loop error (the difference between setpoint and process variable)
$M_{initial}$	is the initial value of the loop output

In order to implement this control function in a digital computer, the continuous function must be quantized into periodic samples of the error value with subsequent calculation of the output. The corresponding equation that is the basis for the digital computer solution is:

M_n	=	$K_C * e_n$	+	$K_I * \sum_1^n + M_{initial}$	+	$K_D * (e_n - e_{n-1})$
output	=	proportional term	+	integral term	+	differential term

where:

M_n	is the calculated value of the loop output at sample time n
K_C	is the loop gain
e_n	is the value of the loop error at sample time n
e_{n-1}	is the previous value of the loop error (at sample time n - 1)
K_I	is the proportional constant of the integral term
$M_{initial}$	is the initial value of the loop output
K_D	is the proportional constant of the differential term

From this equation, the integral term is shown to be a function of all the error terms from the first sample to the current sample. The differential term is a function of the current sample and the previous sample, while the proportional term is only a function of the current sample. In a digital computer it is not practical to store all samples of the error term, nor is it necessary.

Since the digital computer must calculate the output value each time the error is sampled beginning with the first sample, it is only necessary to store the previous value of the error and the previous value of the integral term. As a result of the repetitive nature of the digital computer solution, a simplification in the equation that must be solved at any sample time can be made. The simplified equation is:

M_n	=	$K_C * e_n$	+	$K_I * e_n + MX$	+	$K_D * (e_n - e_{n-1})$
output	=	proportional term	+	integral term	+	differential term

where:

- M_n is the calculated value of the loop output at sample time n
- K_C is the loop gain
- e_n is the value of the loop error at sample time n
- e_{n-1} is the previous value of the loop error (at sample time n - 1)
- K_I is the proportional constant of the integral term
- MX is the previous value of the integral term (at sample time n - 1)
- K_D is the proportional constant of the differential term

The CPU uses a modified form of the above simplified equation when calculating the loop output value. This modified equation is:

M_n	=	MP_n	+	MI_n	+	MD_n
output	=	proportional term	+	integral term	+	differential term

where:

- M_n is the calculated value of the loop output at sample time n
- MP_n is the value of the proportional term of the loop output at sample time n
- MI_n is the value of the integral term of the loop output at sample time n
- MD_n is the value of the differential term of the loop output at sample time n

Proportional Term

The proportional term MP is the product of the gain (K_C), which controls the sensitivity of the output calculation, and the error (e), which is the difference between the setpoint (SP) and the process variable (PV) at a given sample time. The equation for the proportional term as solved by the CPU is:

$$MP_n = K_C * (SP_n - PV_n)$$

where:

MP_n is the value of the proportional term of the loop output at sample time n
K_C is the loop gain
SP_n is the value of the setpoint at sample time n
PV_n is the value of the process variable at sample time n

Integral Term

The integral term MI is proportional to the sum of the error over time. The equation for the integral term as solved by the CPU is:

$$MI_n = K_C * T_S / T_I * (SP_n - PV_n) + MX$$

where:

MI_n is the value of the integral term of the loop output at sample time n
K_C is the loop gain
T_S is the loop sample time
T_I is the integration period of the loop (also called the integral time or reset)
SP_n is the value of the setpoint at sample time n
PV_n is the value of the process variable at sample time n
MX is the value of the integral term at sample time n - 1 (also called the integral sum or the bias)

The integral sum or bias (MX) is the running sum of all previous values of the integral term. After each calculation of MI_n, the bias is updated with the value of MI_n which may be adjusted or clamped (see the section “Variables and Ranges” for details). The initial value of the bias is typically set to the output value ($M_{initial}$) just prior to the first loop output calculation. Several constants are also part of the integral term, the gain (K_C), the sample time (T_S), which is the cycle time at which the PID loop recalculates the output value, and the integral time or reset (T_I), which is a time used to control the influence of the integral term in the output calculation.

Differential Term

The differential term MD is proportional to the change in the error. The equation for the differential term:

$$MD_n = K_C * T_D / T_S * ((SP_n - PV_n) - (SP_{n-1} - PV_{n-1}))$$

To avoid step changes or bumps in the output due to derivative action on setpoint changes, this equation is modified to assume that the setpoint is a constant ($SP_n = SP_{n-1}$). This results in the calculation of the change in the process variable instead of the change in the error as shown:

$$MD_n = K_C * T_D / T_S * (SP_n - PV_n - SP_n + PV_{n-1})$$

or just:

$$MD_n = K_C * T_D / T_S * (PV_{n-1} - PV_n)$$

where:

MD_n	is the value of the differential term of the loop output at sample time n
K_C	is the loop gain
T_S	is the loop sample time
T_D	is the differentiation period of the loop (also called the derivative time or rate)
SP_n	is the value of the setpoint at sample time n
SP_{n-1}	is the value of the setpoint at sample time n - 1
PV_n	is the value of the process variable at sample time n
PV_{n-1}	is the value of the process variable at sample time n - 1

The process variable rather than the error must be saved for use in the next calculation of the differential term. At the time of the first sample, the value of PV_{n-1} is initialized to be equal to PV_n.

Selection of Loop Control

In many control systems it may be necessary to employ only one or two methods of loop control. For example only proportional control or proportional and integral control may be required. The selection of the type of loop control desired is made by setting the value of the constant parameters.

If you do not want integral action (no “I” in the PID calculation), then a value of infinity should be specified for the integral time (reset). Even with no integral action, the value of the integral term may not be zero, due to the initial value of the integral sum MX.

If you do not want derivative action (no “D” in the PID calculation), then a value of 0.0 should be specified for the derivative time (rate).

If you do not want proportional action (no “P” in the PID calculation) and you want I or ID control, then a value of 0.0 should be specified for the gain. Since the loop gain is a factor in the equations for calculating the integral and differential terms, setting a value of 0.0 for the loop gain will result in a value of 1.0 being used for the loop gain in the calculation of the integral and differential terms.

Converting and Normalizing the Loop Inputs

A loop has two input variables, the setpoint and the process variable. The setpoint is generally a fixed value such as the speed setting on the cruise control in your automobile. The process variable is a value that is related to loop output and therefore measures the effect that the loop output has on the controlled system. In the example of the cruise control, the process variable would be a tachometer input that measures the rotational speed of the tires.

Both the setpoint and the process variable are real world values whose magnitude, range, and engineering units may be different. Before these real world values can be operated upon by the PID instruction, the values must be converted to normalized, floating-point representations.

The first step is to convert the real world value from a 16-bit integer value to a floating-point or real number value. The following instruction sequence is provided to show how to convert from an integer value to a real number.

```
XORD    AC0,  AC0            //Clear the accumulator.
MOVW    AIW0,  AC0           //Save the analog value in the accumulator.
LDW>=   AC0,  0              //If the analog value is positive,
JMP     0                    //then convert to a real number.
NOT                          //Else,
ORD     16#FFFF0000,  AC0    //sign extend the value in AC0.
LBL     0
DTR     AC0,  AC0            //Convert the 32-bit integer to a real number.
```

The next step is to convert the real number value representation of the real world value to a normalized value between 0.0 and 1.0. The following equation is used to normalize either the setpoint or process variable value:

$$\mathbf{R_{Norm} = (R_{Raw} / Span) + Offset}$$

where:

R_{Norm}	is the normalized, real number value representation of the real world value
R_{Raw}	is the un-normalized or raw, real number value representation of the real world value
Offset	is 0.0 for unipolar values is 0.5 for bipolar values
Span	is the maximum possible value minus the minimum possible value = 32,000 for unipolar values (typical) = 64,000 for bipolar values (typical)

The following instruction sequence shows how to normalize the bipolar value in AC0 (whose span is 64,000) as a continuation of the previous instruction sequence:

```
/R      64000.0,  AC0        //Normalize the value in the accumulator
+R      0.5,  AC0            //Offset the value to the range from 0.0 to 1.0
MOVR    AC0,  VD100          //Store the normalized value in the loop TABLE
```

Converting the Loop Output to a Scaled Integer Value

The loop output is the control variable, such as the throttle setting in the example of the cruise control on the automobile. The loop output is a normalized, real number value between 0.0 and 1.0. Before the loop output can be used to drive an analog output, the loop output must be converted to a 16-bit, scaled integer value. This process is the reverse of converting the PV and SP to a normalized value. The first step is to convert the loop output to a scaled, real number value using the formula given below:

$$R_{Scal} = (M_n - Offset) * Span$$

where:

R_{Scal} is the scaled, real number value of the loop output
M_n is the normalized, real number value of the loop output

Offset is 0.0 for unipolar values
is 0.5 for bipolar values

Span is the maximum possible value minus the minimum possible value
= 32,000 for unipolar values (typical)
= 64,000 for bipolar values (typical)

The following instruction sequence shows how to scale the loop output:

```
MOVR   VD108,  AC0      //Move the loop output to the accumulator.
-R     0.5,  AC0        //Include this statement only if the value is
                        //bipolar.
*R     64000.0,  AC0    //Scale the value in the accumulator.
```

Next, the scaled, real number value representing the loop output must be converted to a 16-bit integer. The following instruction sequence shows how to do this conversion:

```
ROUND  AC0   AC0        //Convert the real number to a 32-bit integer.
MOVW   AC0,  AQW0       //Write the 16-bit integer value to the analog
                        //output.
```

Forward- or Reverse-Acting Loops

The loop is forward-acting if the gain is positive and reverse-acting if the gain is negative. (For I or ID control, where the gain value is 0.0, specifying positive values for integral and derivative time will result in a forward-acting loop, and specifying negative values will result in a reverse-acting loop.)

Variables and Ranges

The process variable and setpoint are inputs to the PID calculation. Therefore the loop table fields for these variables are read but not altered by the PID instruction.

The output value is generated by the PID calculation, so the output value field in the loop table is updated at the completion of each PID calculation. The output value is clamped between 0.0 and 1.0. The output value field can be used as an input by the user to specify an initial output value when making the transition from manual control to PID instruction (auto) control of the output (see discussion in the Modes section below).

If integral control is being used, then the bias value is updated by the PID calculation and the updated value is used as an input in the next PID calculation. When the calculated output value goes out of range (output would be less than 0.0 or greater than 1.0), the bias is adjusted according to the following formulas:

$$MX = 1.0 - (MP_n + MD_n)$$ when the calculated output, $M_n > 1.0$

or

$$MX = - (MP_n + MD_n)$$ when the calculated output, $M_n < 0.0$

where:

MX is the value of the adjusted bias
MP_n is the value of the proportional term of the loop output at sample time n
MD_n is the value of the differential term of the loop output at sample time n
M_n is the value of the loop output at sample time n

By adjusting the bias as described, an improvement in system responsiveness is achieved once the calculated output comes back into the proper range. The calculated bias is also clamped between 0.0 and 1.0 and then is written to the bias field of the loop table at the completion of each PID calculation. The value stored in the loop table is used in the next PID calculation.

The bias value in the loop table can be modified by the user prior to execution of the PID instruction in order to address bias value problems in certain application situations. Care must be taken when manually adjusting the bias, and any bias value written into the loop table must be a real number between 0.0 and 1.0.

A comparison value of the process variable is maintained in the loop table for use in the derivative action part of the PID calculation. You should not modify this value.

Modes

There is no built-in mode control for S7-200 PID loops. The PID calculation is performed only when power flows to the PID box. Therefore, “automatic” or “auto” mode exists when the PID calculation is performed cyclically. “Manual” mode exists when the PID calculation is not performed.

The PID instruction has a power-flow history bit, similar to a counter instruction. The instruction uses this history bit to detect a 0-to-1 power flow transition, which when detected will cause the instruction to perform a series of actions to provide a bumpless change from manual control to auto control. In order for change to auto mode control to be bumpless, the value of the output as set by the manual control must be supplied as an input to the PID instruction (written to the loop table entry for M_n) before switching to auto control. The PID instruction performs the following actions to values in the loop table to ensure a bumpless change from manual to auto control when a 0-to-1 power flow transition is detected:

- Sets setpoint (SP_n) = process variable (PV_n)
- Sets old process variable (PV_{n-1}) = process variable (PV_n)
- Sets bias (MX) = output value (M_n)

The default state of the PID history bits is “set” and that state is established at CPU startup and on every STOP-to-RUN mode transition of the controller. If power flows to the PID box the first time that it is executed after entering RUN mode, then no power flow transition is detected and the bumpless mode change actions will not be performed.

Alarm Checking and Special Operations

The PID instruction is a simple but powerful instruction that performs the PID calculation. If other processing is required such as alarm checking or special calculations on loop variables, these must be implemented using the basic instructions supported by the CPU.

Error Conditions

When it is time to compile, the CPU will generate a compile error (range error) and the compilation will fail if the loop table start address or PID loop number operands specified in the instruction are out of range.

Certain loop table input values are not range checked by the PID instruction. You must take care to ensure that the process variable and setpoint (as well as the bias and previous process variable if used as inputs) are real numbers between 0.0 and 1.0.

If any error is encountered while performing the mathematical operations of the PID calculation, then SM1.1 (overflow or illegal value) will be set and execution of the PID instruction will be terminated. (Update of the output values in the loop table may be incomplete, so you should disregard these values and correct the input value causing the mathematical error before the next execution of the loop's PID instruction.)

Loop Table

The loop table is 36 bytes long and has the format shown in Table 9-19.

Table 9-19 Format of the Loop Table

Offset	Field	Format	Type	Description
0	Process variable (PV_n)	Double word - real	in	Contains the process variable, which must be scaled between 0.0 and 1.0.
4	Setpoint (SP_n)	Double word - real	in	Contains the setpoint, which must be scaled between 0.0 and 1.0.
8	Output (M_n)	Double word - real	in/out	Contains the calculated output, scaled between 0.0 and 1.0.
12	Gain (K_C)	Double word - real	in	Contains the gain, which is a proportional constant. Can be a positive or negative number.
16	Sample time (T_S)	Double word - real	in	Contains the sample time, in seconds. Must be a positive number.
20	Integral time or reset (T_I)	Double word - real	in	Contains the integral time or reset, in minutes. Must be a positive number.
24	Derivative time or rate (T_D)	Double word - real	in	Contains the derivative time or rate, in minutes. Must be a positive number.
28	Bias (MX)	Double word - real	in/out	Contains the bias or integral sum value between 0.0 and 1.0.
32	Previous process variable (PV_{n-1})	Double word - real	in/out	Contains the previous value of the process variable stored from the last execution of the PID instruction.

PID Program Example

In this example, a water tank is used to maintain a constant water pressure. Water is continuously being taken from the water tank at a varying rate. A variable speed pump is used to add water to the tank at a rate that will maintain adequate water pressure and also keep the tank from being emptied.

The setpoint for this system is a water level setting that is equivalent to the tank being 75% full. The process variable is supplied by a float gauge that provides an equivalent reading of how full the tank is and which can vary from 0% or empty to 100% or completely full. The output is a value of pump speed that allows the pump to run from 0% to 100% of maximum speed.

The setpoint is predetermined and will be entered directly into the loop table. The process variable will be supplied as a unipolar, analog value from the float gauge. The loop output will be written to a unipolar, analog output which is used to control the pump speed. The span of both the analog input and analog output is 32,000.

Only proportional and integral control will be employed in this example. The loop gain and time constants have been determined from engineering calculations and may be adjusted as required to achieve optimum control. The calculated values of the time constants are:

K_C is 0.25

T_S is 0.1 seconds

T_I is 30 minutes

The pump speed will be controlled manually until the water tank is 75% full, then the valve will be opened to allow water to be drained from the tank. At the same time, the pump will be switched from manual to auto control mode. A digital input will be used to switch the control from manual to auto. This input is described below:

I0.0 is manual/auto control; 0 is manual, 1 is auto

While in manual control mode, the pump speed will be written by the operator to VD108 as a real number value from 0.0 to 1.0.

Figure 9-28 shows the control program for this application.

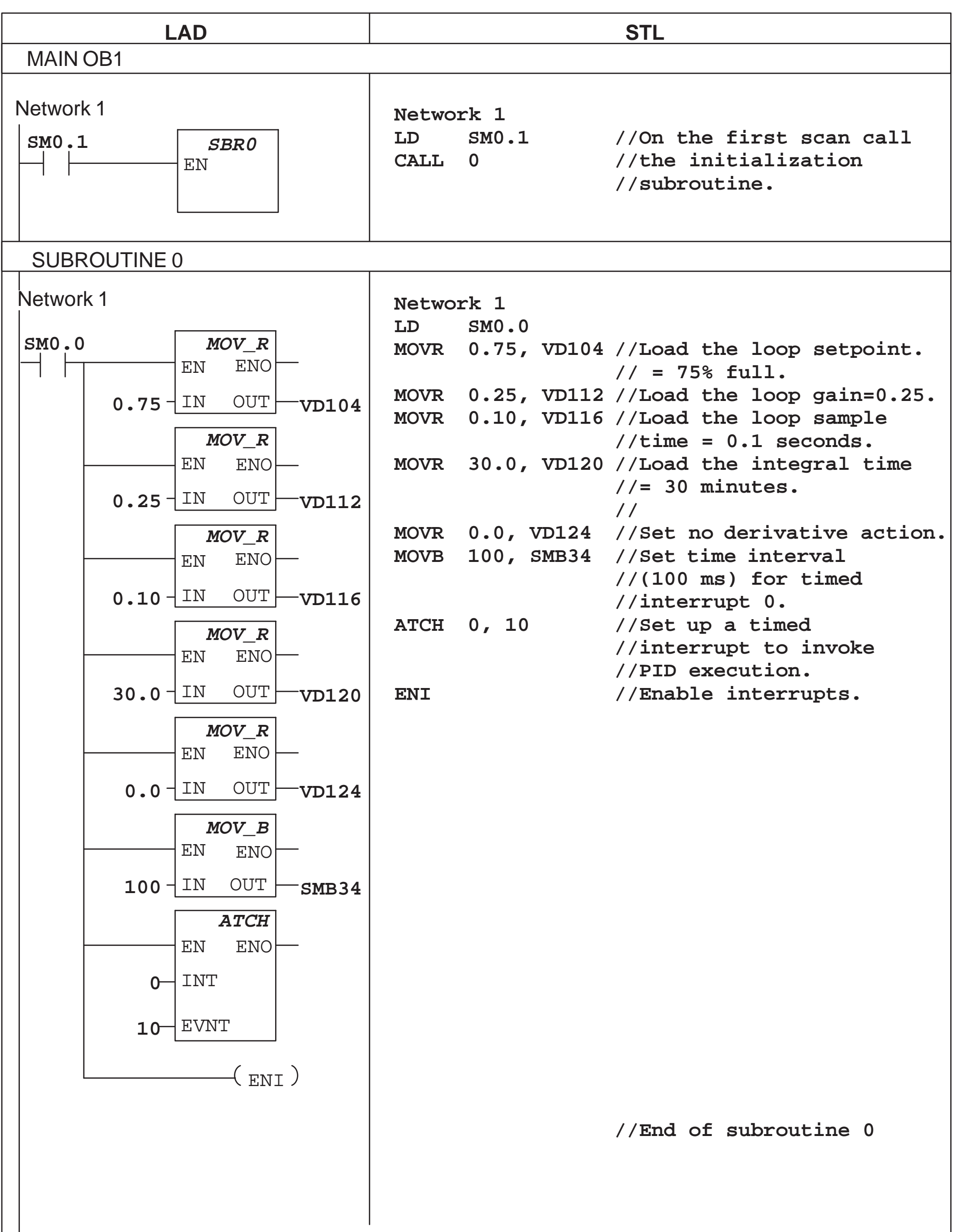


Figure 9-28 Example of PID Loop Control

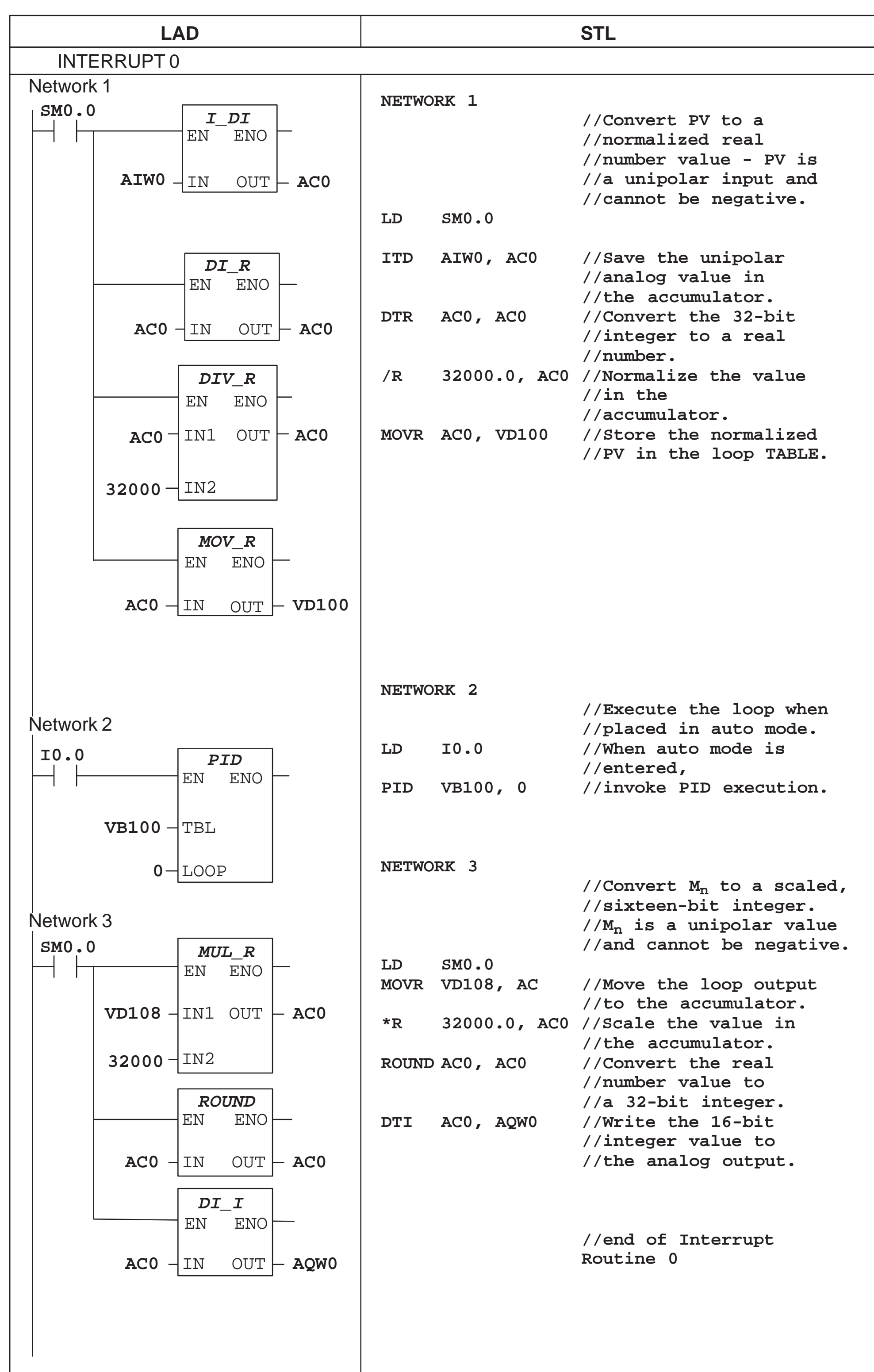


Figure 9-28 Example of PID Loop Control (continued)

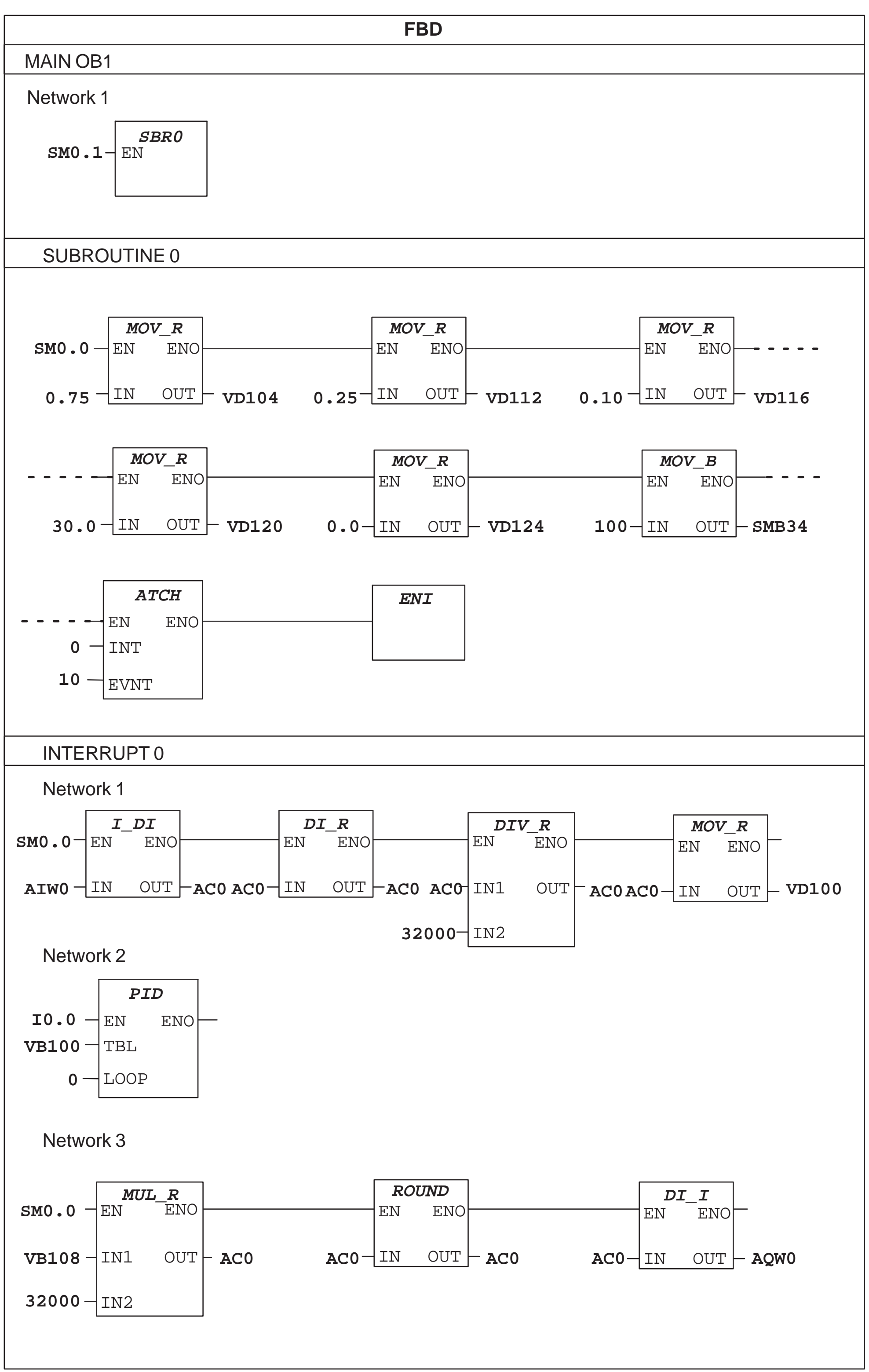
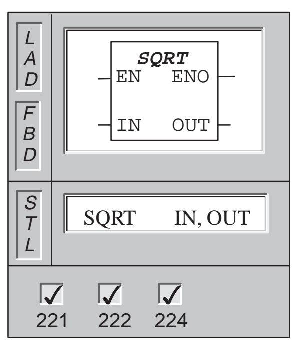


Figure 9-28 Example of PID Loop Control (continued)

Square Root



The **Square Root** instruction takes the square root of a 32-bit real number (IN) and produces a 32-bit real number result (OUT) as shown in the equation:

$\sqrt{IN} = OUT$

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

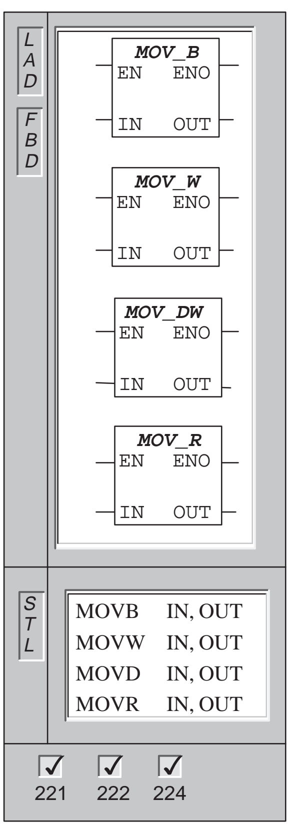
This instruction affects the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative).

SM1.1 is used to indicate overflow errors and illegal values. If SM1.1 is set, then the status of SM1.0 and SM1.2 is not valid and the original input operands are not altered. If SM1.1 is not set, then the math operation has completed with a valid result and SM1.0 and SM1.2 contain valid status.

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, SD, LD, AC, Constant, *VD, *AC, *LD	REAL
OUT	VD, ID, QD, MD, SMD, SD, LD,AC, *VD, *AC, *LD	REAL

9.10 SIMATIC Move Instructions

Move Byte, Move Word, Move Double Word, Move Real



The **Move Byte** instruction moves the input byte (IN) to the output byte (OUT). The input byte is not altered by the move.

The **Move Word** instruction moves the input word (IN) to the output word (OUT). The input word is not altered by the move.

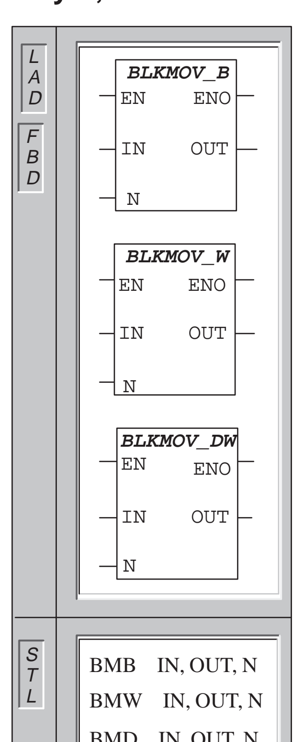
The **Move Double Word** instruction moves the input double word (IN) to the output double word (OUT). The input double word is not altered by the move.

The **Move Real** instruction moves a 32-bit, real input double word (IN) to the output double word (OUT). The input double word is not altered by the move.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Move...	Inputs/Outputs	Operands	Data Types
Byte	IN	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
	OUT	VB, IB, QB, MB, SB, SMB, LB, AC, *VD, *AC, *LD	BYTE
Word	IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, Constant, AC *VD, *AC, *LD	WORD, INT
	OUT	VW, T, C, IW, QW, SW, MW, SMW, LW, AC, AQW, *VD, *AC, *LD	WORD, INT
Double Word	IN	VD, ID, QD, MD, SD, SMD, LD, HC, &VB, &IB, &QB, &MB, &SB, &T, &C, AC, Constant, *VD, *AC, *LD	DWORD, DINT
	OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	DWORD, DINT
Real	IN	VD, ID, QD, MD, SD, SMD, LD, AC, Constant, *VD, *AC, *LD	REAL
	OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	REAL

Block Move Byte, Block Move Word, Block Move Double Word



The **Block Move Byte** instruction moves the number of bytes (N) from the input address IN to the output address OUT. N has a range of 1 to 255.

The **Block Move Word** instruction moves the number of words (N), from the input address IN to the output address OUT. N has a range of 1 to 255.

The **Block Move Double Word** instruction moves the number of double words (N), from the input address IN, to the output address OUT. N has a range of 1 to 255.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

Block Move...	Inputs/Outputs	Operands	Data Types
Byte	IN, OUT	VB, IB, QB, MB,SB, SMB, LB, *VD, *AC, *LD	BYTE
	N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
Word	IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, *VD, *AC, *LD	WORD
	N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
	OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AQW, *VD, *LD, *AC	WORD
Double Word	IN, OUT	VD, ID, QD, MD, SD, SMD, LD, *VD, *AC, *LD	DWORD
	N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE

Block Move Example

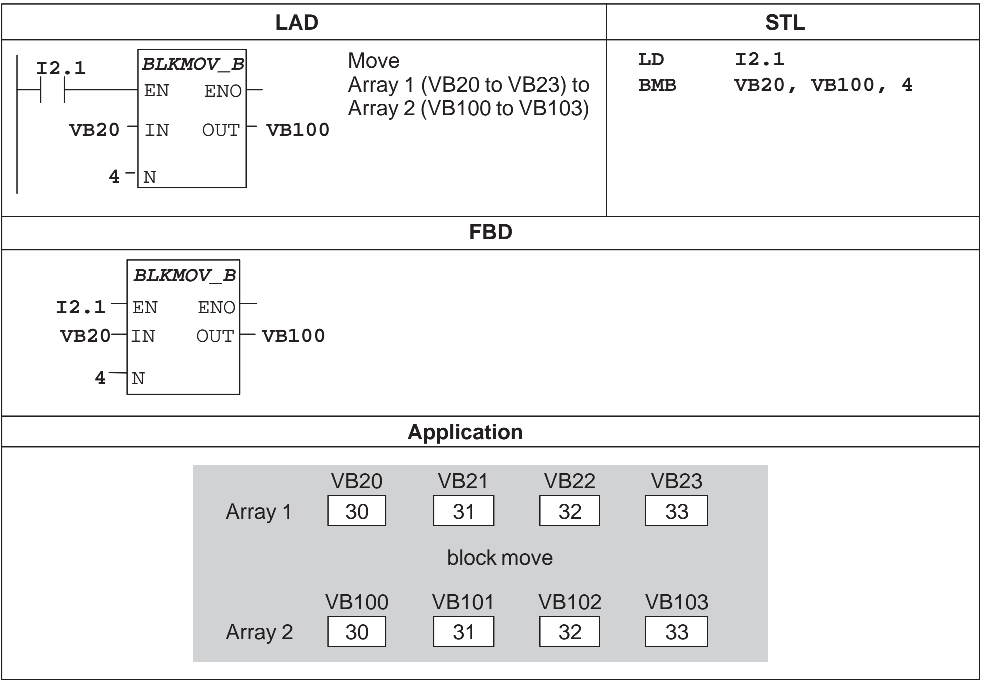
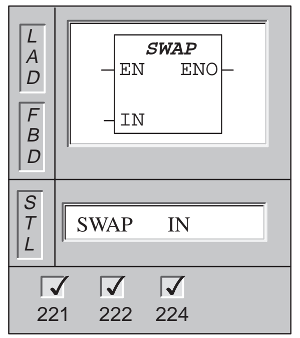


Figure 9-29 Example of Block Move Instructions for LAD, STL, and FBD

Swap Bytes



The **Swap Bytes** instruction exchanges the most significant byte with the least significant byte of the word (IN).

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	WORD

Move and Swap Examples

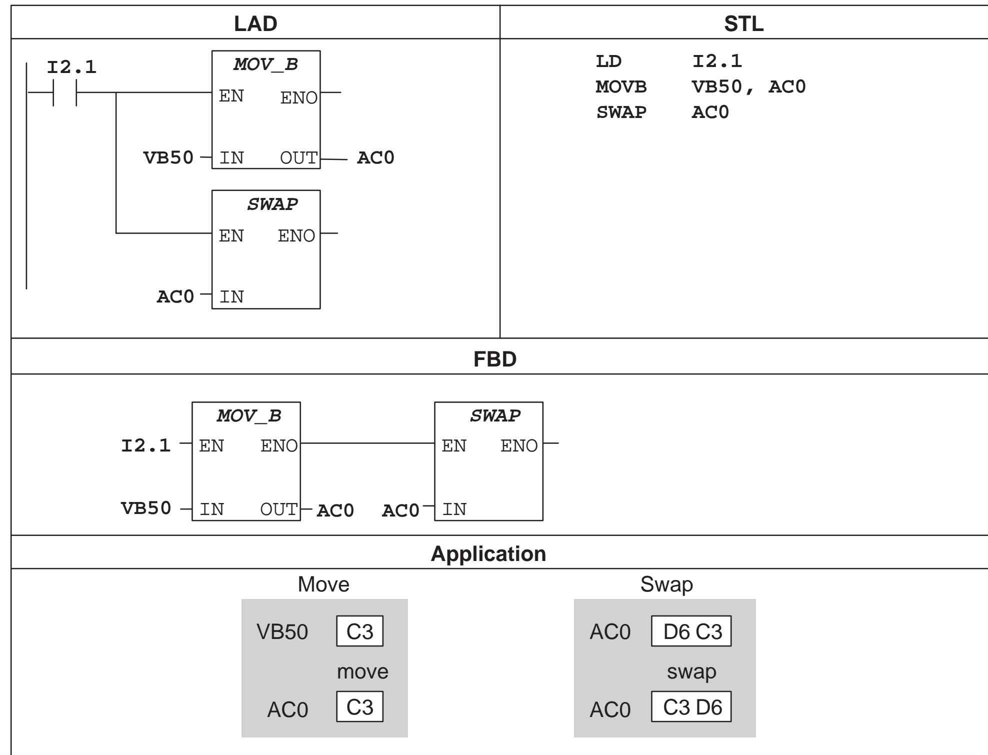
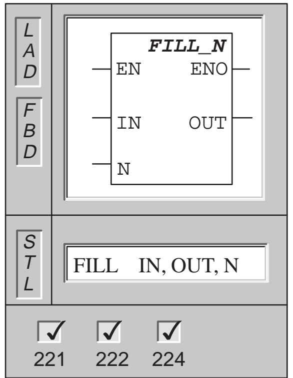


Figure 9-30 Example of Move and Swap Instructions for LAD, STL, and FBD

Memory Fill



The **Memory Fill** instruction fills memory starting at the output word (OUT), with the word input pattern (IN) for the number of words specified by N. N has a range of 1 to 255.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, AIW, T, C, AC, Constant, *VD, *AC, *LD	WORD
N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AQW, *VD, *AC, *LD	WORD

Fill Example

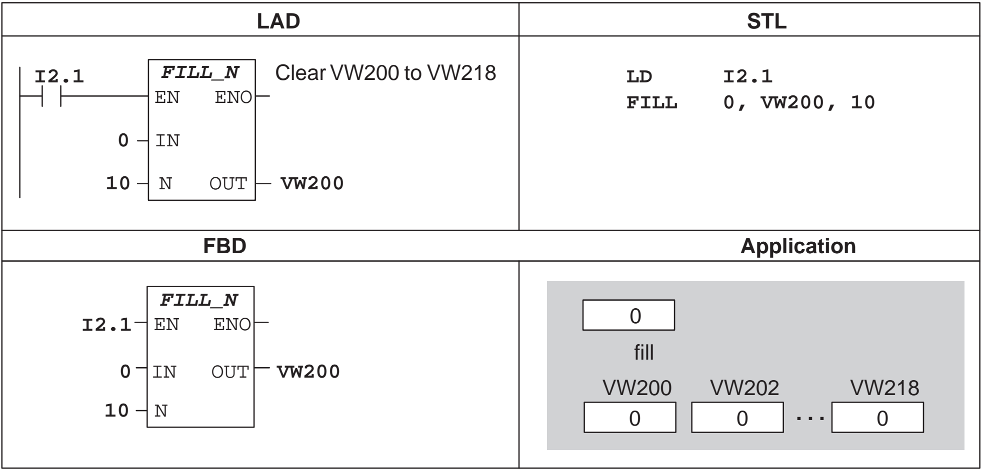
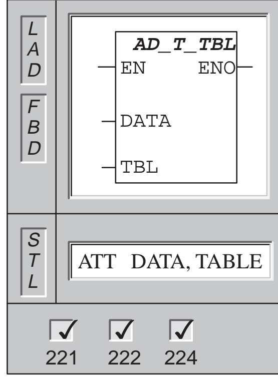


Figure 9-31 Example of Fill Instructions for LAD, STL, and FBD

9.11 SIMATIC Table Instructions

Add to Table



The **Add To Table** instruction adds word values (DATA) to the table (TBL).

The first value of the table is the maximum table length (TL). The second value is the entry count (EC), which specifies the number of entries in the table. (See Figure 9-32.) New data are added to the table after the last entry. Each time new data are added to the table, the entry count is incremented. A table may have up to 100 data entries.

Error conditions that set ENO = 0: SM1.4 (table overflow), SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

This instruction affects the following Special Memory bits: SM1.4 is set to 1 if you try to overfill the table.

Inputs/Outputs	Operands	Data Types
DATA	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, AC, Constant, *VD, *AC, *LD	WORD
TBL	VW, IW, QW, MW, SW, SMW, LW, T, C, *VD, *AC, *LD	WORD

Add to Table Example

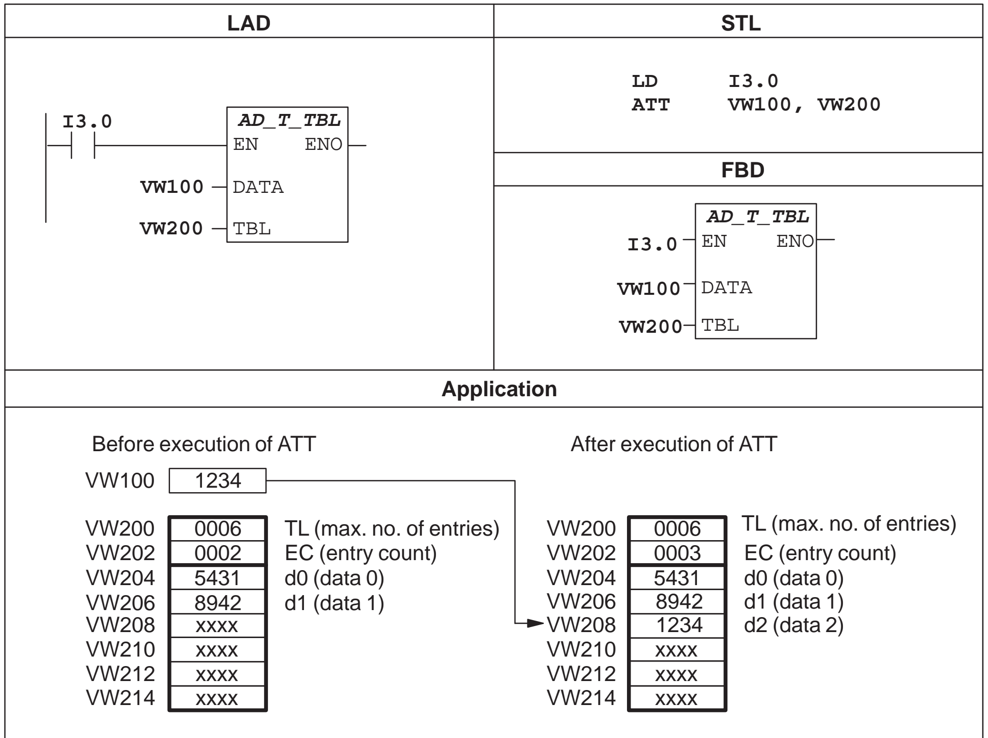
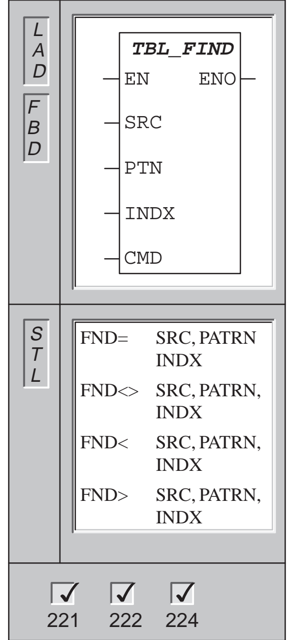


Figure 9-32 Example of Add To Table Instruction

Table Find



The **Table Find** instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PTN) that matches the search criteria defined by CMD. The command parameter (CMD) is given a numeric value of 1 to 4 that corresponds to =, <>, <, and >, respectively.

If a match is found, the INDX points to the matching entry in the table. To find the next matching entry, the INDX must be incremented before invoking the Table Find instruction again. If a match is not found, the INDX has a value equal to the entry count.

A table may have up to 100 data entries. The data entries (area to be searched) are numbered from 0 to a maximum value of 99.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

Inputs/Outputs	Operands	Data Types
SRC	VW, IW, QW, MW, SMW, LW, T, C, *VD, *AC, *LD	WORD
PTN	VW, IW, QW, MW, SW, SMW, AIW, LW, T, C, AC, Constant, *VD, *AC, *LD	INT
INDX	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	WORD
CMD	Constant	BYTE

Note

When you use the Find instructions with tables generated with ATT, LIFO, and FIFO instructions, the entry count and the data entries correspond directly. The maximum-number-of-entries word required for ATT, LIFO, and FIFO is not required by the Find instructions. Consequently, the SRC operand of a Find instruction is one word address (two bytes) higher than the TBL operand of a corresponding ATT, LIFO, or FIFO instruction, as shown in Figure 9-33.

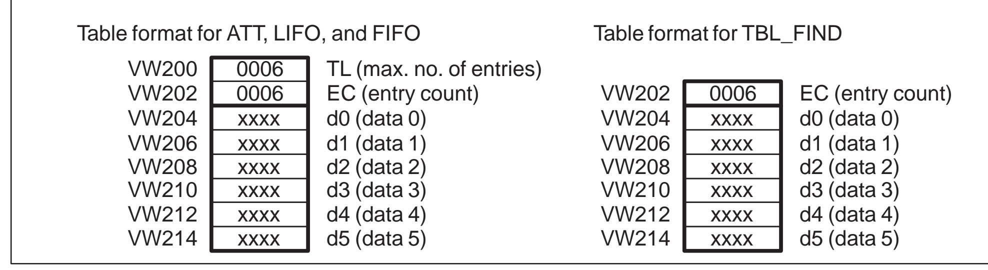


Figure 9-33 Difference in Table Format between Find Instructions and ATT, LIFO, FIFO

Table Find Example

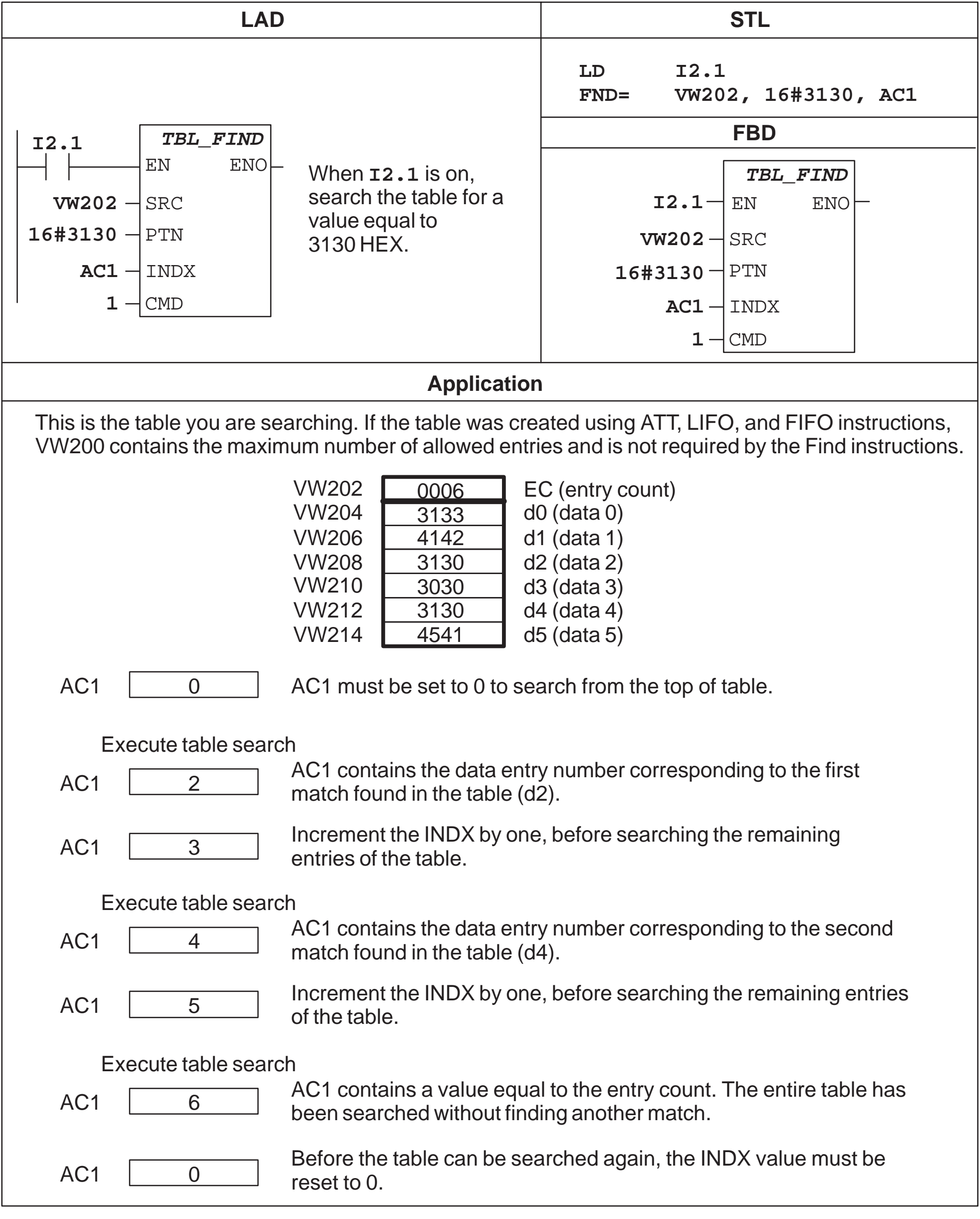
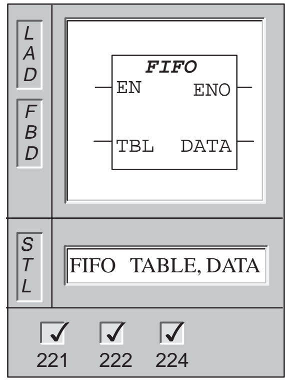


Figure 9-34 Example of Find Instructions for LAD, STL, and FBD

First-In-First-Out



The **First-In-First-Out** instruction removes the first entry in the table (TBL), and outputs the value to a specified location (DATA). All other entries of the table are shifted up one location. The entry count in the table is decremented for each instruction execution.

Error conditions that set ENO = 0: SM1.5 (empty table), SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

This instruction affects the following Special Memory bits: SM1.5 is set to 1 if you try to remove an entry from an empty table.

Inputs/Outputs	Operands	Data Types
TABLE	VW, IW, QW, MW, SW, SMW, LW, T, C,*VD, *AC, *LD	WORD
DATA	VW, IW, QW, MW, SW, SMW, LW, AC, AQW, T, C, *VD, *AC, *LD	WORD

First-In-First-Out Example

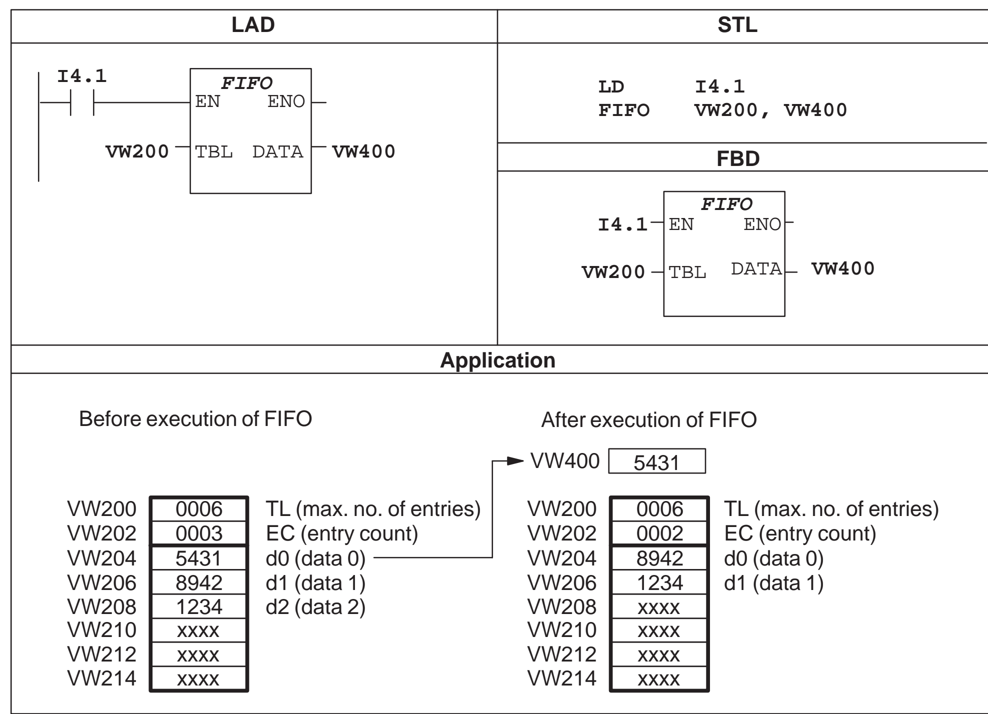
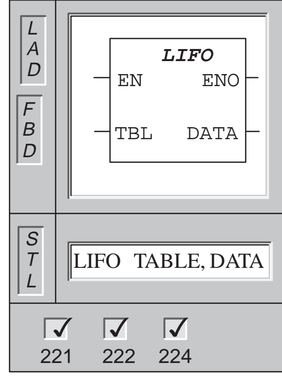


Figure 9-35 Example of First-In-First-Out Instruction

Last-In-First-Out



The **Last-In-First-Out** instruction removes the last entry in the table (TBL), and outputs the value to a location specified by DATA. The entry count in the table is decremented for each instruction execution.

Error conditions that set ENO = 0: SM1.5 (empty table), SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range)

This instruction affects the following Special Memory bits: SM1.5 is set to 1 if you try to remove an entry from an empty table.

Inputs/Outputs	Operands	Data Types
TABLE	VW, IW, QW, MW, SW, SMW, LW, T, C,*VD, *AC, *LD	WORD
DATA	VW, IW, QW, MW, SW, SMW, LW, AQW, T, C, AC, *VD, *AC, *LD	WORD

Last-In-First-Out Example

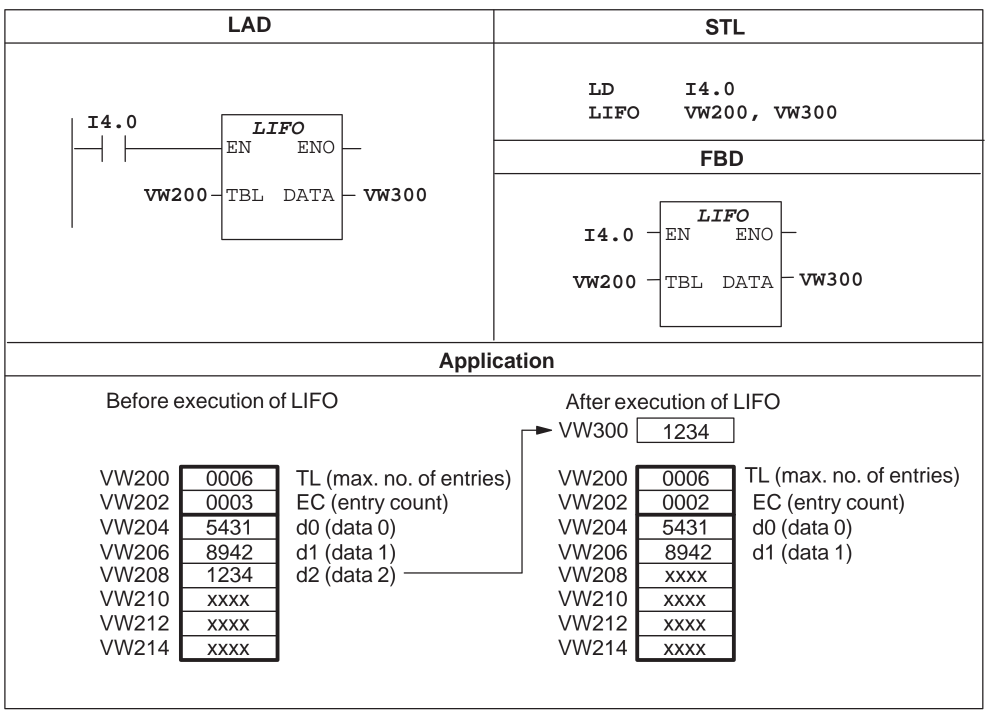
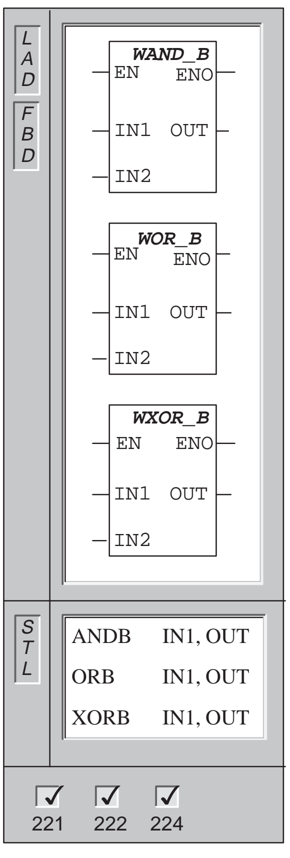


Figure 9-36 Example of Last-In-First-Out Instruction

9.12 SIMATIC Logical Operations Instructions

And Byte, Or Byte, Exclusive Or Byte



The **And Byte** instruction ANDs the corresponding bits of two input bytes and loads the result (OUT) in a byte.

The **Or Byte** instruction ORs the corresponding bits of two input bytes and loads the result (OUT) in a byte.

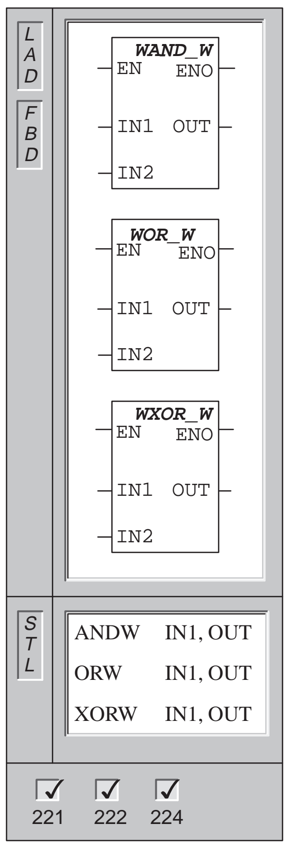
The **Exclusive Or Byte** instruction XORs the corresponding bits of two input bytes and loads the result (OUT) in a byte.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero)

Inputs/Outputs	Operands	Data Types
IN1, IN2	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VB, IB, QB, MB,SB, SMB, LB, AC, *VD, *AC, *LD	BYTE

And Word, Or Word, Exclusive Or Word



The **And Word** instruction ANDs the corresponding bits of two input words and loads the result (OUT) in a word.

The **Or Word** instruction ORs the corresponding bits of two input words and loads the result (OUT) in a word.

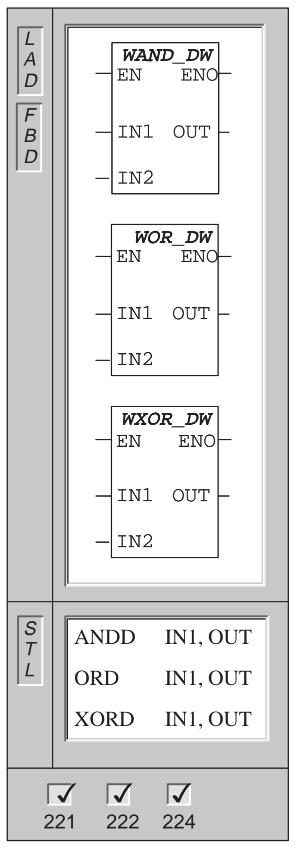
The **Exclusive Or Word** instruction XORs the corresponding bits of two input words and loads the result (OUT) in a word.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero)

Inputs/Outputs	Operands	Data Types
IN1, IN2	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, AC, Constant, *VD, *AC, *LD	WORD
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	WORD

And Double Word, Or Double Word, Exclusive Or Double Word



The **And Double Word** instruction ANDs the corresponding bits of two double word inputs and loads the result (OUT) in a double word.

The **Or Double Word** instruction ORs the corresponding bits of two double word inputs and loads the result (OUT) in a double word.

The **Exclusive Or Double Word** instruction XORs the corresponding bits of two double word inputs and loads the result (OUT) in a double word.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero)

Inputs/Outputs	Operands	Data Types
IN1, IN2	VD, ID, QD, MD, SMD, AC, LD, HC, Constant, *VD, *AC, SD, *LD	DWORD
OUT	VD, ID, QD, MD, SMD, LD, AC, *VD, *AC, SD, *LD	DWORD

And, Or, and Exclusive Or Instructions Example

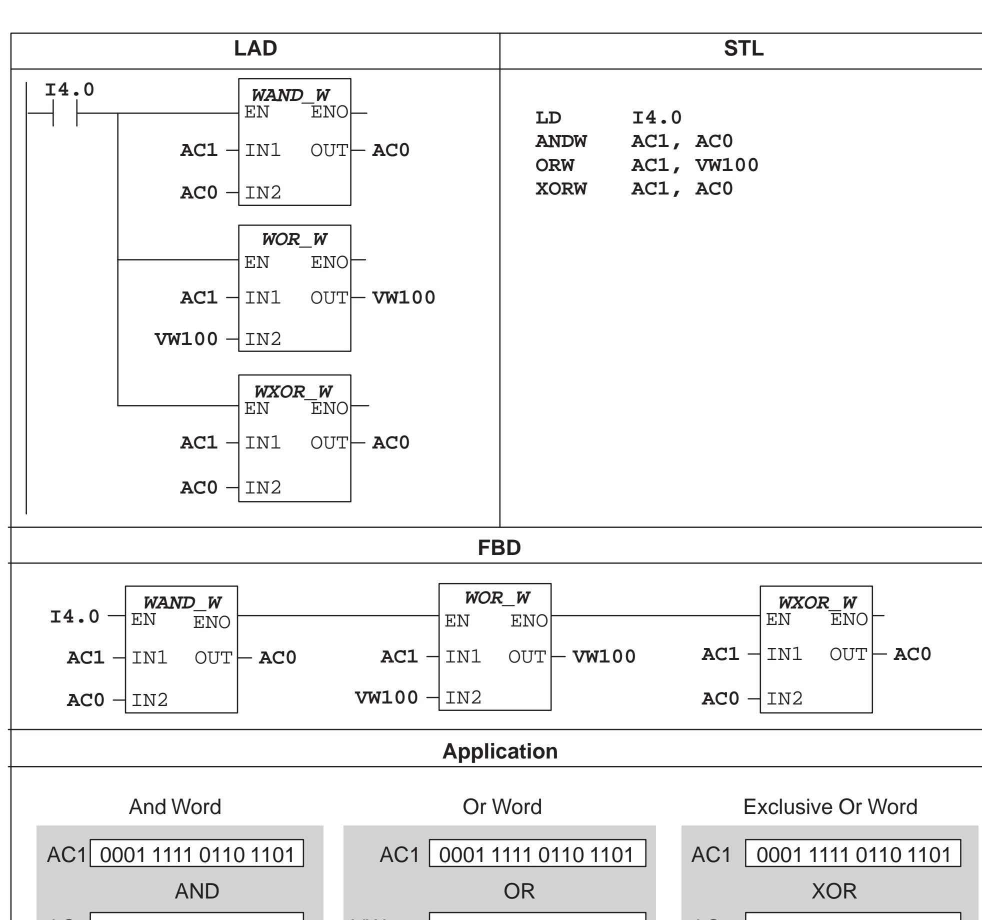
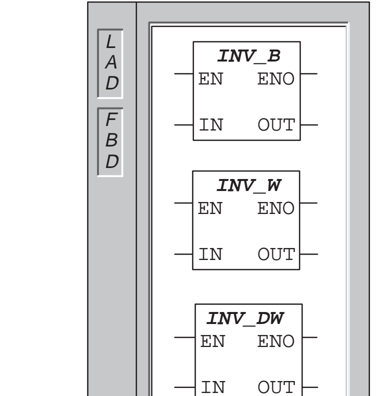


Figure 9-37 Example of the Logical Operation Instructions

Invert Byte, Invert Word, Invert Double Word Instructions



The **Invert Byte** instruction forms the ones complement of the input byte IN, and loads the result into byte value OUT.

The **Invert Word** instruction forms the ones complement of the input word IN, and loads the result in word value OUT.

The **Invert Double Word** instruction forms the ones complement of the input double word IN, and loads the result in double word value OUT.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

This instruction affects the following Special Memory bits: SM1.0 (zero)

Invert...	Inputs/Outputs	Operands	Data Types
Byte	IN	VB, IB, QB, MB,SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
	OUT	VB, IB, QB, MB, SB, SMB, LB, AC,*VD, *AC, *LD	BYTE
Word	IN	VW, IW, QW, MW, SW, SMW, T, C, AIW, LW, AC, Constant, *VD, *AC, *LD	WORD
	OUT	VW, IW, QW, MW,SW, SMW, T, C, LW, AC, *VD, *AC, *LD	WORD
Double Word	IN	VD, ID, QD, MD, SD, SMD, LD, HC, AC, Constant, *VD, *AC, *LD	DWORD
	OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	DWORD

Invert Example

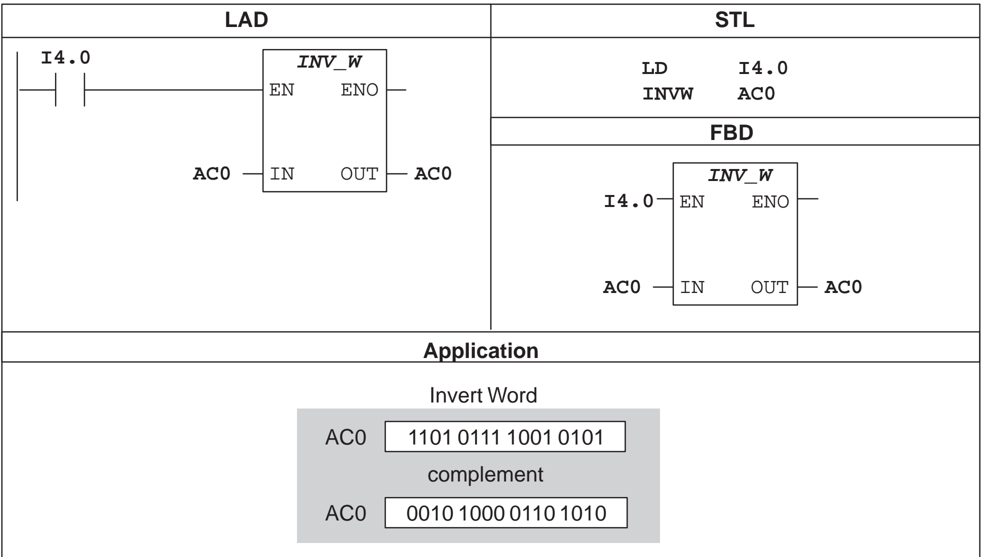
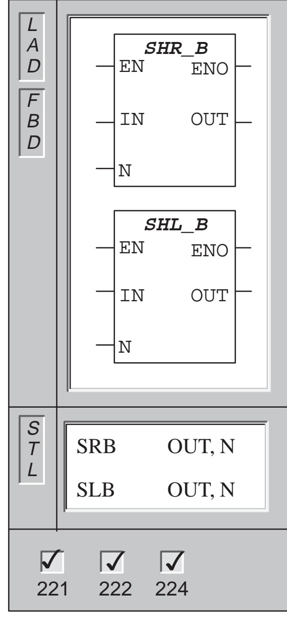


Figure 9-38 Example of Invert Instruction for LAD and STL

9.13 SIMATIC Shift and Rotate Instructions

Shift Right Byte, Shift Left Byte



The **Shift Right Byte** and **Shift Left Byte** instructions shift the input byte (IN) value right or left by the shift count (N), and load the result in the output byte (OUT).

The **Shift** instructions fill with zeros as each bit is shifted out. If the shift count (N) is greater than or equal to 8, the value is shifted a maximum of 8 times.

If the shift count is greater than 0, the overflow memory bit (SM1.1) takes on the value of the last bit shifted out. The zero memory bit (SM1.0) is set if the result of the shift operation is zero.

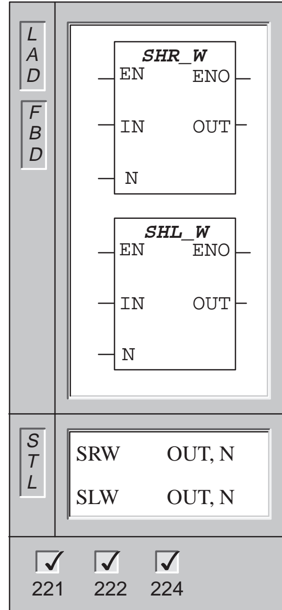
Shift right and shift left byte operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN, OUT	VB, IB, QB, MB, SB, SMB, LB, AC, *VD, *AC, *LD	BYTE
N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE

Shift Right Word, Shift Left Word



The **Shift Right Word** and **Shift Left Word** instructions shift the input word (IN) value right or left by the shift count (N), and load the result in the output word (OUT).

The **Shift** instructions fill with zeros as each bit is shifted out. If the shift count (N) is greater than or equal to 16, the value is shifted a maximum of 16 times. If the shift count is greater than 0, the overflow memory bit (SM1.1) takes on the value of the last bit shifted out. The zero memory bit (SM1.0) is set if the result of the shift operation is zero.

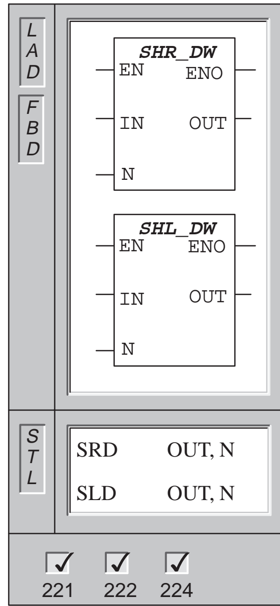
Shift right and shift left word operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, AC, Constant, *VD, *AC, *LD	WORD
N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	WORD

Shift Right Double Word, Shift Left Double Word



The **Shift Right Double Word** and **Shift Left Double Word** instructions shift the input double word value (IN) right or left by the shift count (N), and load the result in the output double word (OUT).

The **Shift** instructions fill with zeros as each bit is shifted out. If the shift count (N) is greater than or equal to 32, the value is shifted a maximum of 32 times. If the shift count is greater than 0, the overflow memory bit (SM1.1) takes on the value of the last bit shifted out. The zero memory bit (SM1.0) is set if the result of the shift operation is zero.

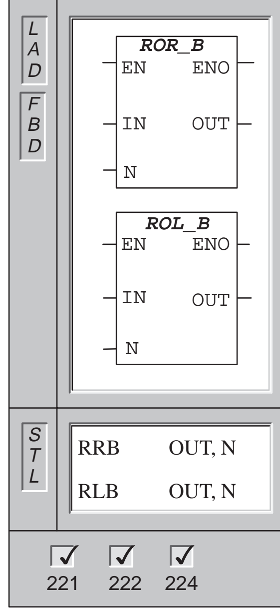
Shift right and shift left double word operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SD, SMD, LD, AC, HC, Constant, *VD, *AC, *LD	DWORD
N	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	DWORD

Rotate Right Byte, Rotate Left Byte



The **Rotate Right Byte** and **Rotate Left Byte** instructions rotate the input byte value (IN) right or left by the shift count (N), and load the result in the output byte (OUT).

If the shift count (N) is greater than or equal to 8, a modulo-8 operation is performed on the shift count (N) before the rotation is executed. This results in a shift count of 0 to 7. If the shift count is 0, a rotate is not performed. If the rotate is performed, the value of the last bit rotated is copied to the overflow bit (SM1.1).

If the shift count is not an integer multiple of 8, the last bit rotated out is copied to the overflow memory bit (SM1.1). The zero memory bit (SM1.0) is set when the value to be rotated is zero.

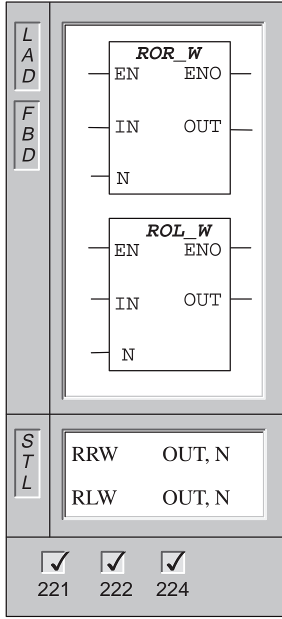
Rotate right byte and rotate left byte operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VB, IB, QB, MB, SMB, SB, LB, AC, *VD, *AC, *LD	BYTE
N	VB, IB, QB, MB, SMB, SB, LB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VB, IB, QB, MB, SMB, SB, LB, AC, *VD, *AC, *LD	BYTE

Rotate Right Word, Rotate Left Word



The **Rotate Right Word** and **Rotate Left Word** instructions rotate the input word value (IN) right or left by the shift count (N), and load the result in the output word (OUT).

If the shift count (N) is greater than or equal to 16, a modulo-16 operation is performed on the shift count (N) before the rotation is executed. This results in a shift count of 0 to 15. If the shift count is 0, a rotation is not performed. If the rotation is performed, the value of the last bit rotated is copied to the overflow bit (SM1.1).

If the shift count is not an integer multiple of 16, the last bit rotated out is copied to the overflow memory bit (SM1.1). The zero memory bit (SM1.0) is set when the value to be rotated is zero.

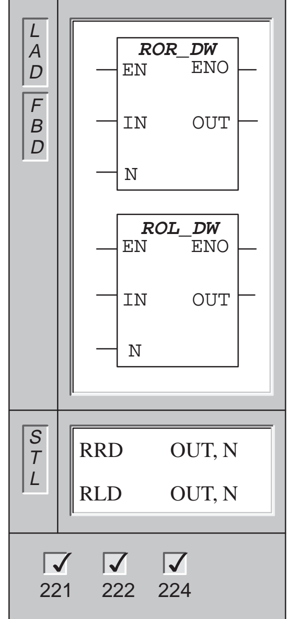
Rotate right word and rotate left word operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VW, T, C, IW, MW, SMW, AC, QW, LW, AIW, Constant, *VD, *AC, SW, *LD	WORD
N	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VW, T, C, IW, QW, MW, SMW, LW, AC, *VD, *AC, SW, *LD	WORD

Rotate Right Double Word, Rotate Left Double Word



The **Rotate Right Double Word** and **Rotate Left Double Word** instructions rotate the input double word value (IN) right or left by the shift count (N), and load the result in the output double word (OUT).

If the shift count (N) is greater than or equal to 32, a modulo-32 operation is performed on the shift count (N) before the rotation is executed. This results in a shift count of 0 to 31. If the shift count is 0, a rotation is not performed. If the rotation is performed, the value of the last bit rotated is copied to the overflow bit (SM1.1).

If the shift count is not an integer multiple of 32, the last bit rotated out is copied to the overflow memory bit (SM1.1). The zero memory bit (SM1.0) is set when the value to be rotated is zero.

Rotate right double word and rotate left double word operations are unsigned.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.0 (zero); SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, LD, AC, HC, Constant, *VD, *AC, SD, *LD	DWORD
N	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VD, ID, QD, MD, SMD, LD, AC, *VD, *AC, SD, *LD	DWORD

Shift and Rotate Examples

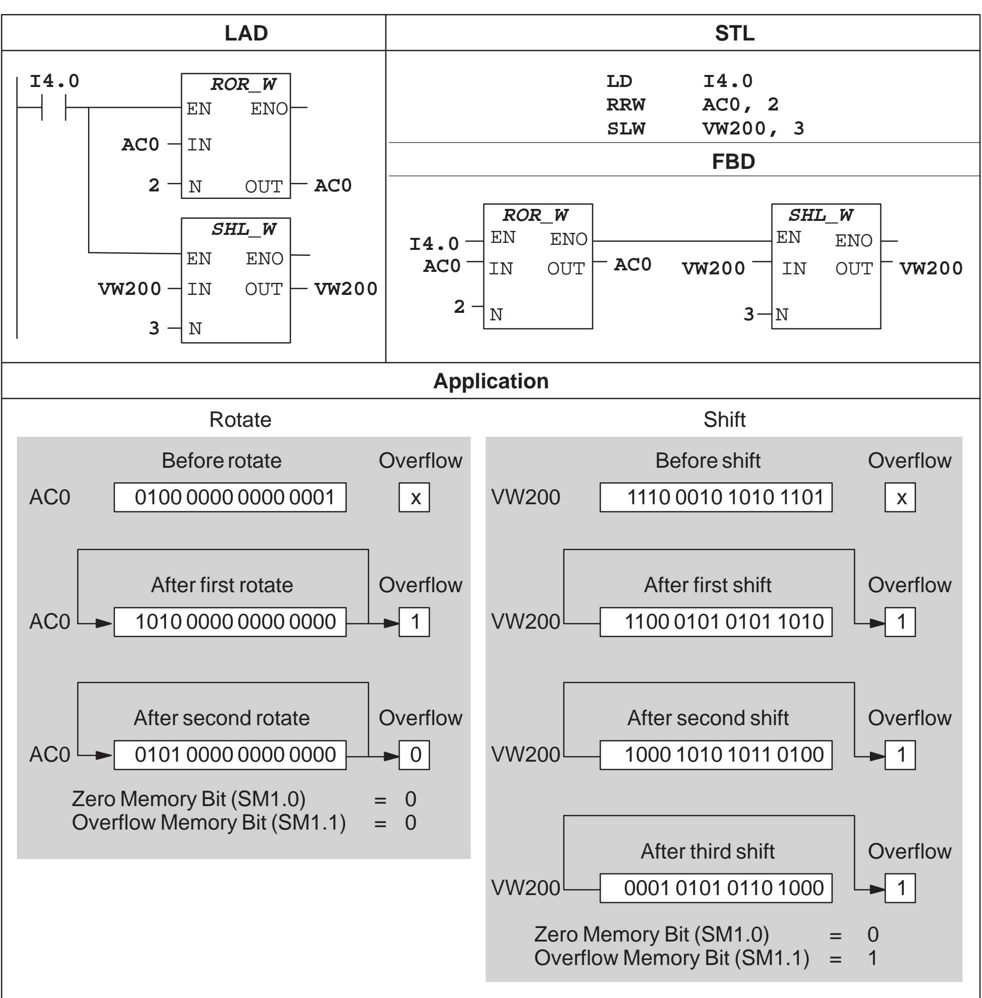
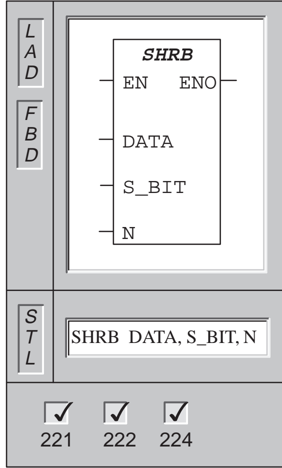


Figure 9-39 Example of Shift and Rotate Instructions for LAD, STL, and FBD

Shift Register Bit



The **Shift Register Bit** (SHRB) instruction shifts the value of DATA into the Shift Register. S_BIT specifies the least significant bit of the Shift Register. N specifies the length of the Shift Register and the direction of the shift (Shift Plus = N, Shift Minus = -N).

Each bit shifted out by the SHRB instruction is placed in the overflow memory bit (SM1.1).

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0091 (operand out of range), 0092 (error in count field)

This instruction affects the following Special Memory bit: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
DATA, S_BIT	I, Q, M, SM, T, C, V, S, L	BOOL
N	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE

Understanding the Shift Register Bit Instruction

The Shift Register Bit instruction provides an easy method for sequencing and controlling product flow or data. Use the Shift Register Bit instruction to shift the entire register one bit, once per scan. The Shift Register Bit instruction is defined by both the least significant bit (S_BIT) and the number of bits specified by the length (N). Figure 9-41 shows an example of the Shift Register Bit instruction.

The address of the most significant bit of the Shift Register (MSB.b) can be computed by the following equation:

MSB.b = [(Byte of S_BIT) + ([N] - 1 + (bit of S_BIT)) / 8] . [remainder of the division by 8]

You must subtract 1 bit because S_BIT is one of the bits of the Shift Register.

For example, if S_BIT is V33.4, and N is 14, then the MSB.b is V35.1, or:

MSB.b = V33 + ([14] - 1 +4)/8
= V33 + 17/8
= V33 + 2 with a remainder of 1
= V35.1

On a Shift Minus, indicated by a negative value of length (N), the input data shifts into the most significant bit of the Shift Register, and shifts out of the least significant bit (S_BIT).

On a Shift Plus, indicated by a positive value of length (N), the input data (DATA) shifts into the least significant bit of the Shift Register, specified by the S_BIT, and out of the most significant bit of the Shift Register.

The data shifted out is then placed in the overflow memory bit (SM1.1). The maximum length of the shift register is 64 bits, positive or negative. Figure 9-40 shows bit shifting for negative and positive values of N.

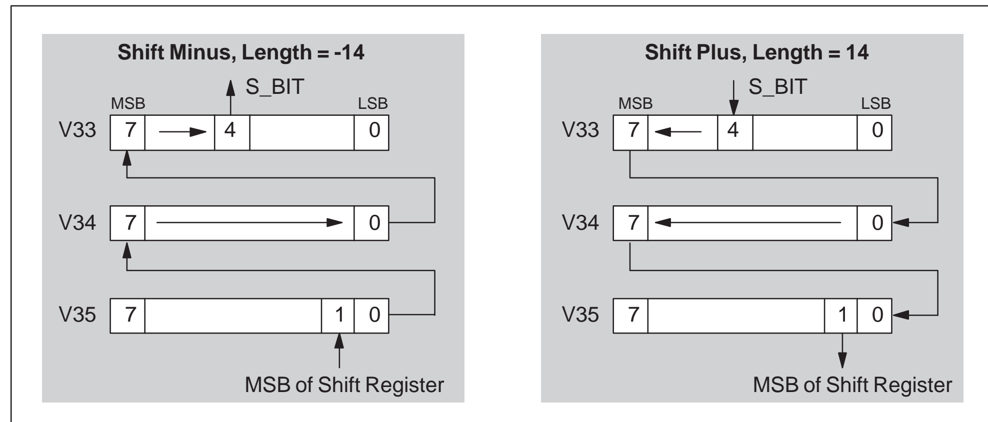


Figure 9-40 Shift Register Entry and Exit for Plus and Minus Shifts

Shift Register Bit Example

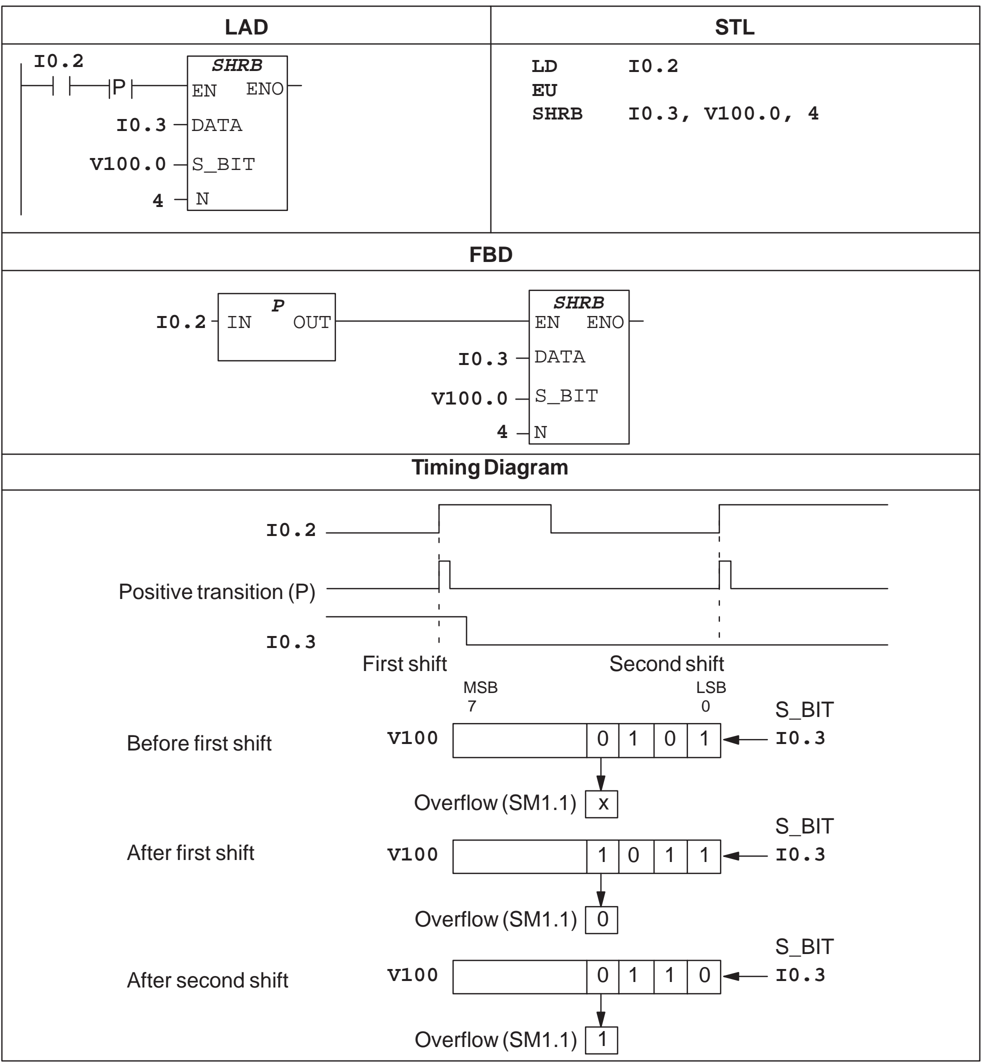
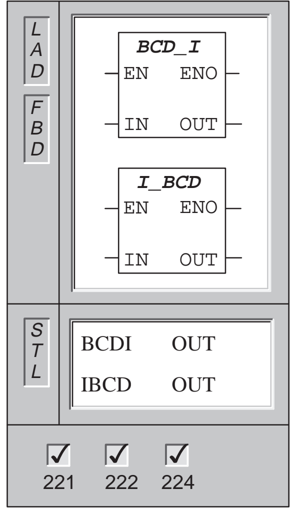


Figure 9-41 Example of Bit Shift Register Instruction for LAD, STL, and FBD

9.14 SIMATIC Conversion Instructions

BCD to Integer, Integer to BCD



The **BCD to Integer** instruction converts the input Binary-Coded Decimal (IN) to an integer value and loads the result into the variable specified by OUT. The valid range for IN is 0 to 9999 BCD.

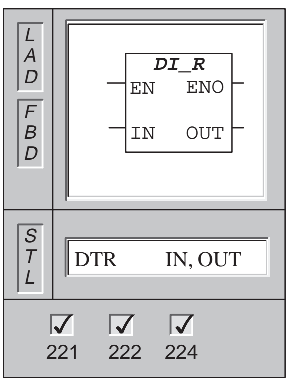
The **Integer to BCD** instruction converts the input integer value (IN) to a Binary-Coded Decimal and loads the result into the variable specified by OUT. The valid range for IN is 0 to 9999 integer.

Error conditions that set ENO = 0: SM1.6 (BCD error), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.6 (invalid BCD)

Inputs/Outputs	Operands	Data Types
IN	VW, T, C, IW, QW, MW, SMW, LW, AC, AIW, Constant, *VD, *AC, SW, *LD	WORD
OUT	VW, T, C, IW, QW, MW, SMW, LW, AC, *VD, *AC, SW, *LD	WORD

Double Integer to Real

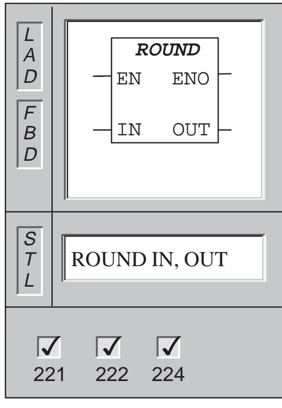


The **Double Integer to Real** instruction converts a 32-bit, signed integer (IN) into a 32-bit real number and places the result into the variable specified by OUT.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, AC, LD, HC, Constant, *VD, *AC, SD, *LD	DINT
OUT	VD, ID, QD, MD, SMD, LD, AC, *VD, *AC, SD, *LD	REAL

Round



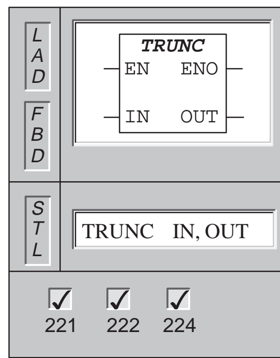
The **Round** instruction converts the real value (IN) to a double integer value and places the result into the variable specified by OUT. If the fraction portion is 0.5 or greater, the number is rounded up.

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, AC, LD, HC, Constant, *VD, *AC, SD, *LD	REAL
OUT	VD, ID, QD, MD, SMD, LD, AC, *VD, *AC, SD, *LD	DINT

Truncate



The **Truncate** instruction converts a 32-bit real number (IN) into a 32-bit signed integer and places the result into the variable specified by OUT. Only the whole number portion of the real number is converted, and the fraction is discarded.

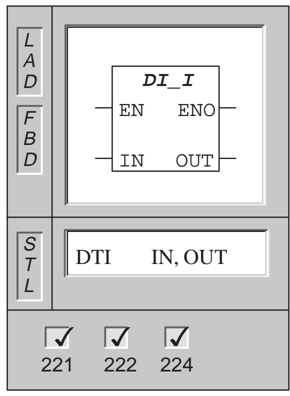
If the value that you are converting is not a valid real number or is too large to be represented in the output, then the overflow bit is set and the output is not affected.

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

This instruction affects the following Special Memory bits: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, LD, AC, Constant, *VD, *AC, SD, *LD	REAL
OUT	VD, ID, QD, MD, SMD, LD, AC, *VD, *AC, SD, *LD	DINT

Double Integer to Integer



The **Double Integer to Integer** instruction converts the double integer value (IN) to an integer value and places the result into the variable specified by OUT.

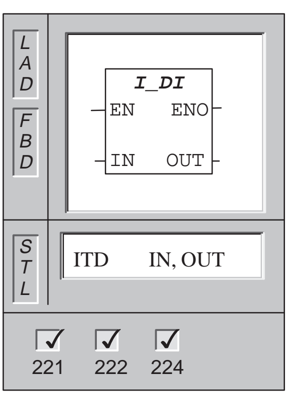
If the value that you are converting is too large to be represented in the output, then the overflow bit is set and the output is not affected.

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SMD, AC, LD, HC, Constant, *VD, *AC, SD, *LD	DINT
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *LD, *AC	INT

Integer to Double Integer

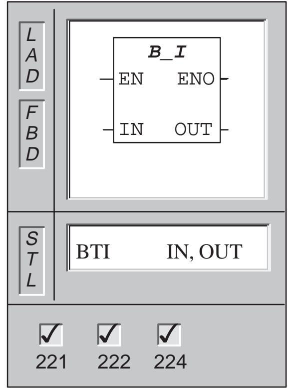


The **Integer to Double Integer** instruction converts the integer value (IN) to a double integer value and places the result into the variable specified by OUT. The sign is extended.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, AC, Constant, *AC, *VD, *LD	INT
OUT	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *LD, *AC	DINT

Byte to Integer

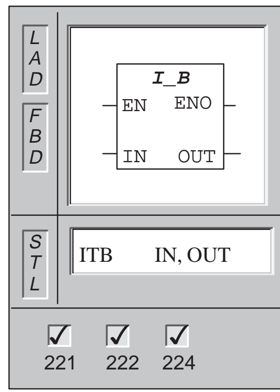


The **Byte to Integer** instruction converts the byte value (IN) to an integer value and places the result into the variable specified by OUT. The byte is unsigned, therefore there is no sign extension.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VB, IB, QB, MB, SB, SMB, LB, AC, Constant, *AC, *VD, *LD	BYTE
OUT	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *LD, *AC	INT

Integer to Byte



The **Integer to Byte** instruction converts the word value (IN) to a byte value and places the result into the variable specified by OUT.

Values 0 to 255 are converted. All other values result in overflow and the output is not affected.

Error conditions that set ENO = 0: SM1.1 (overflow), SM4.3 (run-time), 0006 (indirect address)

These instructions affect the following Special Memory bits: SM1.1 (overflow)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, T, C, AIW, AC, Constant, *VD, *LD, *AC	INT
OUT	VB, IB, QB, MB, SB, SMB, LB, AC, *VD, *AC, *LD	BYTE

Convert Example

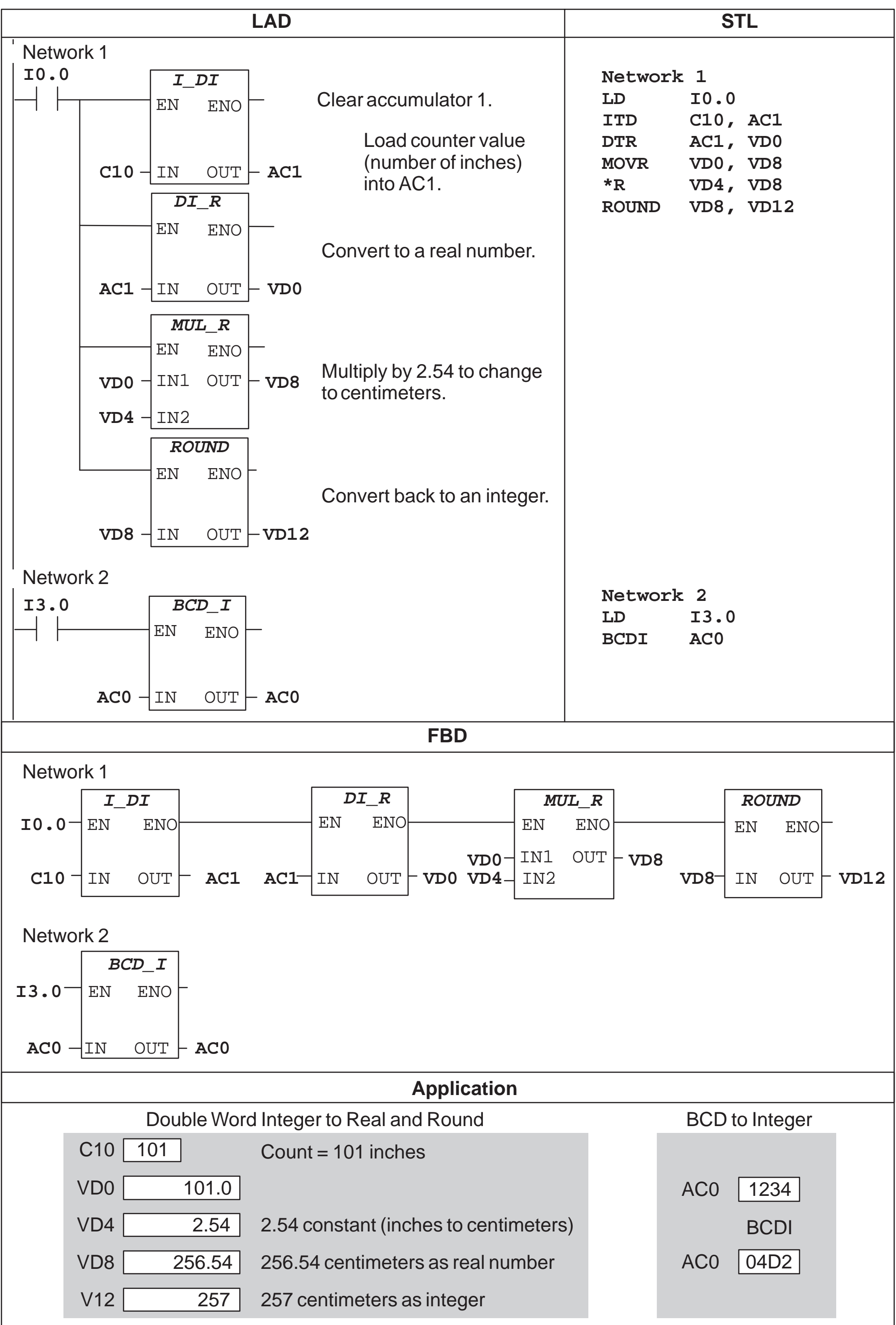
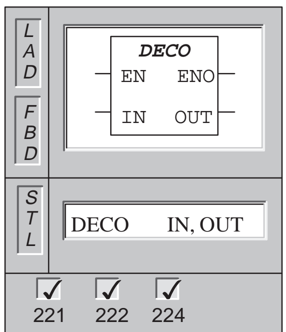


Figure 9-42 Example of Conversion Instructions

Decode

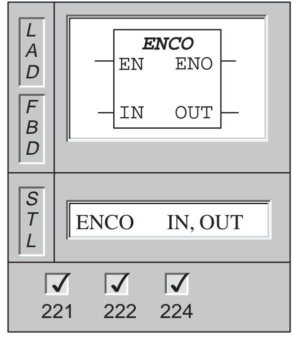


The **Decode** instruction sets the bit in the output word (OUT) that corresponds to the bit number represented by the least significant "nibble" (4 bits) of the input byte (IN). All other bits of the output word are set to 0.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VB, IB, QB, MB, SMB, LB, SB, AC, Constant, *VD, *AC, *LD	BYTE
OUT	VW, IW, QW, MW, SMW, LW, SW, AQW, T, C, AC, *VD, *AC, *LD	WORD

Encode



The **Encode** instruction writes the bit number of the least significant bit set of the input word (IN) into the least significant "nibble" (4 bits) of the output byte (OUT).

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
IN	VW, T, C, IW, QW, MW, SMW, AC, LW, AIW, Constant, *VD, *AC, SW, *LD	WORD
OUT	VB, IB, QB, MB, SMB, LB, AC, *VD, *AC, SB, *LD	BYTE

Decode, Encode Examples

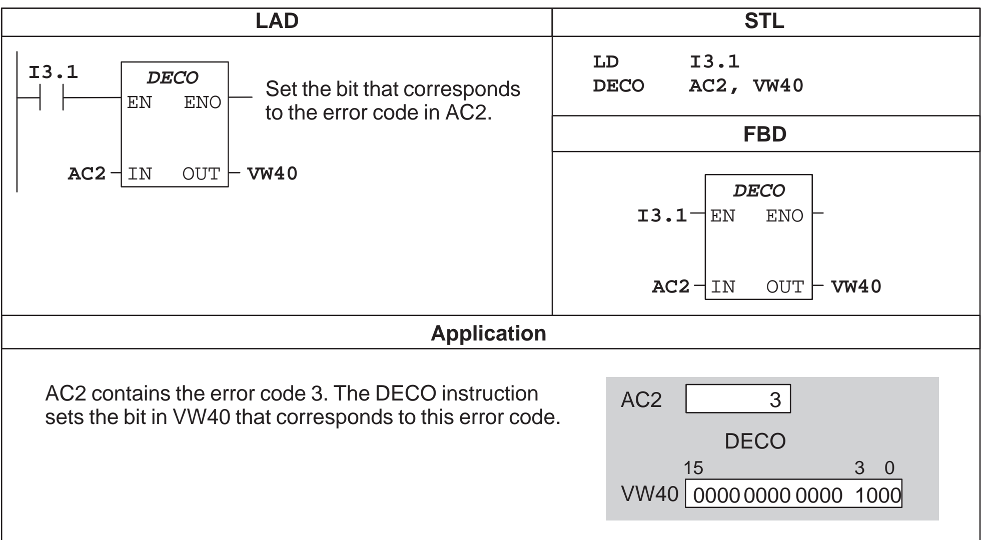


Figure 9-43 Example of Setting an Error Bit Using Decode

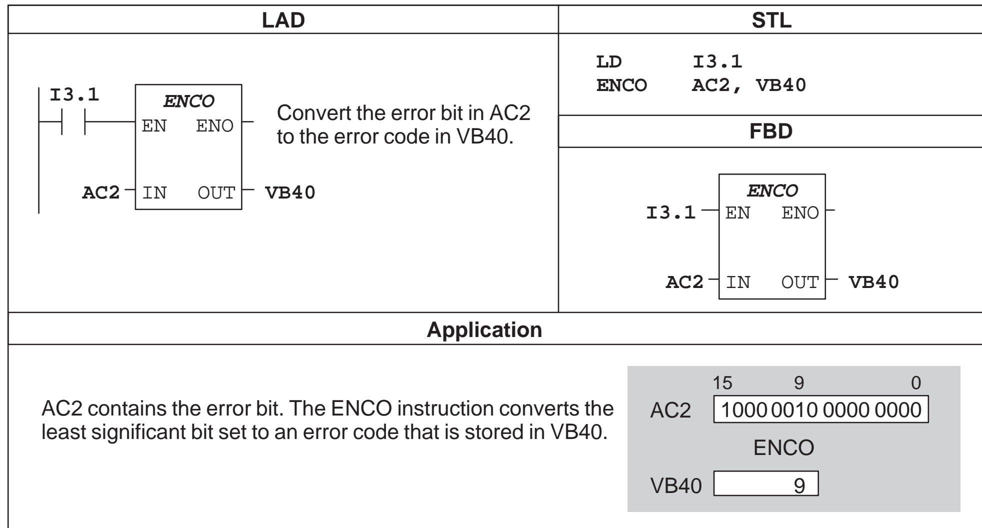
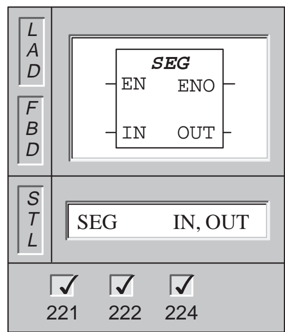


Figure 9-44 Example of Converting the Error Bit into an Error Code Using Encode

Segment

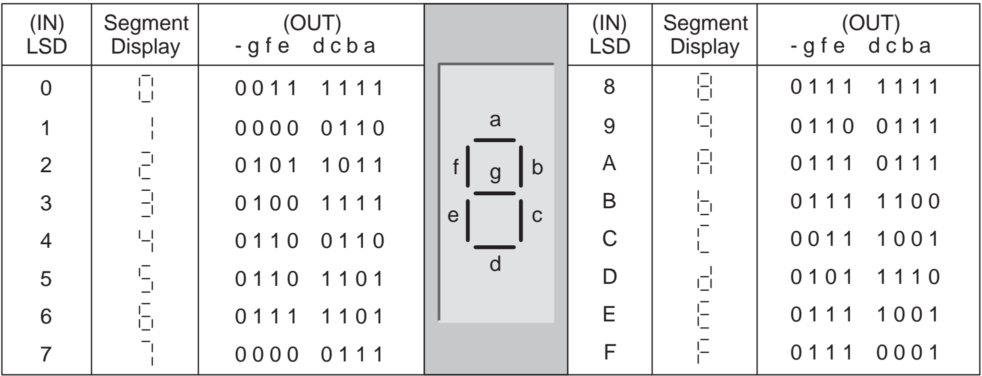


The **Segment** instruction uses the character specified by IN to generate a bit pattern (OUT) that illuminates the segments of a seven-segment display. The illuminated segments represent the character in the least significant digit of the input byte (IN).

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Figure 9-45 shows the seven segment display coding used by the Segment instruction.

Inputs/Outputs	Operands	Data Types
IN	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VB, IB, QB, MB, SMB, LB, AC, *VD, *AC, SB, *LD	BYTE



(IN) LSD	Segment Display	(OUT) - g f e d c b a	(IN) LSD	Segment Display	(OUT) - g f e d c b a
0		0 0 1 1 1 1 1 1	8		0 1 1 1 1 1 1 1
1		0 0 0 0 0 1 1 0	9		0 1 1 0 0 1 1 1
2		0 1 0 1 1 0 1 1	A		0 1 1 1 0 1 1 1
3		0 1 0 0 1 1 1 1	B		0 1 1 1 1 1 0 0
4		0 1 1 0 0 1 1 0	C		0 0 1 1 1 0 0 1
5		0 1 1 0 1 1 0 1	D		0 1 0 1 1 1 1 0
6		0 1 1 1 1 1 0 1	E		0 1 1 1 1 0 0 1
7		0 0 0 0 0 1 1 1	F		0 1 1 1 0 0 0 1

Figure 9-45 Seven Segment Display Coding

Segment Example

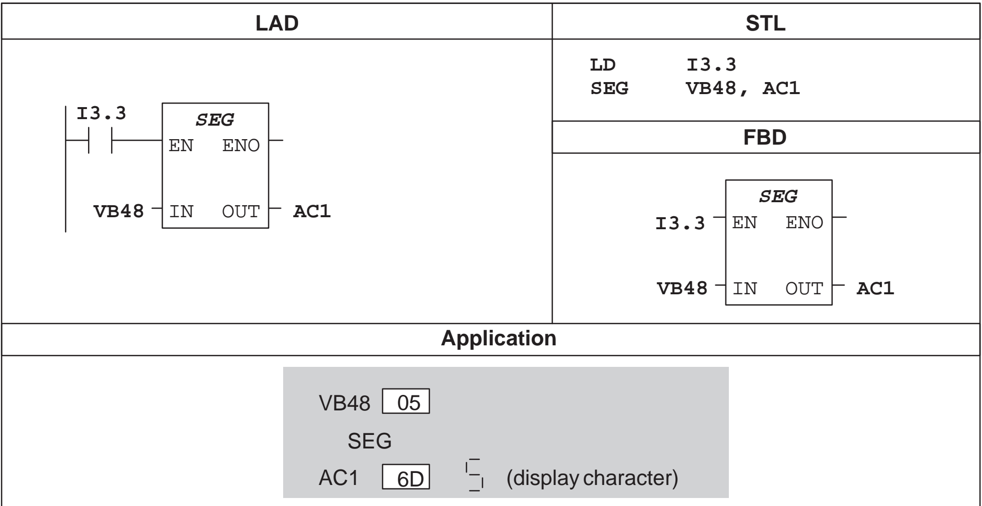
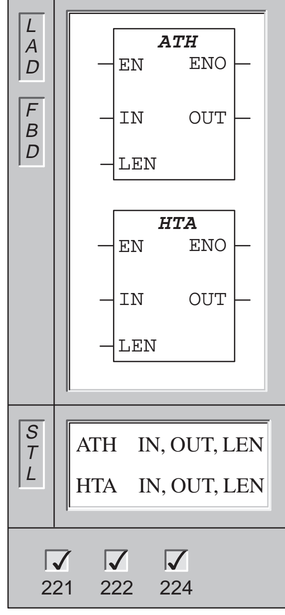


Figure 9-46 Example of Segment Instruction

ASCII to HEX, HEX to ASCII



The **ASCII to HEX** instruction converts the ASCII string of length (LEN), starting at IN, to hexadecimal digits starting at OUT. The maximum length of the ASCII string is 255 characters.

The **HEX to ASCII** instruction converts the hexadecimal digits, starting with the input byte (IN), to an ASCII string starting at OUT. The number of hexadecimal digits to be converted is specified by length (LEN). The maximum number of the hexadecimal digits that can be converted is 255.

Legal ASCII characters are the hexadecimal values 30 to 39, and 41 to 46.

ASCII to Hex: Error conditions that set ENO = 0:
SM1.7 (illegal ASCII), SM4.3 (run-time),
0006 (indirect address), 0091 (operand out of range)

Hex to ASCII: Error conditions that set ENO = 0:
SM4.3 (run-time), 0006 (indirect address),
0091 (operand out of range)

These instructions affect the following Special Memory bits: SM1.7 (illegal ASCII)

Inputs/Outputs	Operands	Data Types
IN, OUT	VB, IB, QB, MB, SMB, LB, *VD, *AC, SB, *LD	BYTE
LEN	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE

ASCII to HEX Example

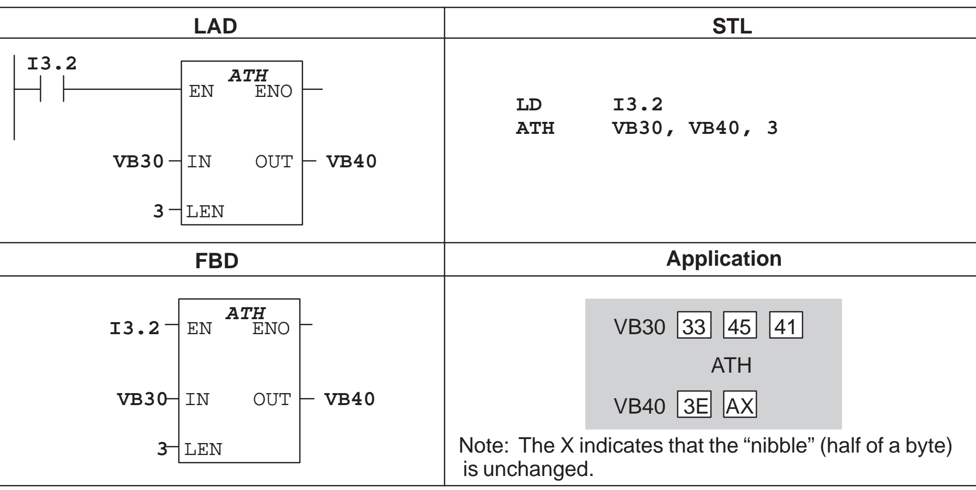
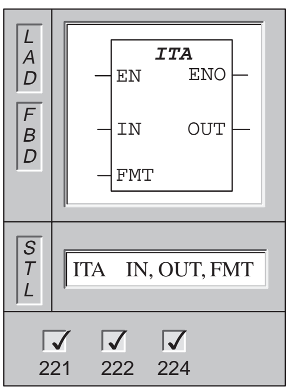


Figure 9-47 Example of ASCII to HEX Instruction

Integer to ASCII



The **Integer to ASCII** instruction converts an integer word (IN) to an ASCII string. The format (FMT) specifies the conversion precision to the right of the decimal, and whether the decimal point is to be shown as a comma or a period. The resulting conversion is placed in 8 consecutive bytes beginning with OUT. The ASCII string is always 8 characters.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), no output (illegal format)

Inputs/Outputs	Operands	Data Types
IN	VW, IW, QW, MW, SW, SMW, LW, AIW, T, C, AC, Constant, *VD, *AC, *LD	INT
FMT	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VB, IB, QB, MB, SMB, LB, *VD, *AC, SB, *LD	BYTE

The format operand (FMT) for the ITA (Integer to ASCII) instruction is defined in Figure 9-48. The size of the output buffer is always 8 bytes. The number of digits to the right of the decimal point in the output buffer is specified by the nnn field. The valid range of the nnn field is 0 to 5. Specifying 0 digits to the right of the decimal point causes the value to be displayed without a decimal point. For values of nnn bigger than 5, the output buffer is filled with ASCII spaces. The c bit specifies the use of either a comma (c=1) or a decimal point (c=0) as the separator between the whole number and the fraction. The upper 4 bits must be zero.

The output buffer is formatted in accord with the following rules:

1. Positive values are written to the output buffer without a sign.
2. Negative values are written to the output buffer with a leading minus sign (-).
3. Leading zeros to the left of the decimal point (except the digit adjacent to the decimal point) are suppressed.
4. Values are right-justified in the output buffer.

Figure 9-48 gives examples of values that are formatted using a decimal point (c = 0) with three digits to the right of the decimal point (nnn = 011).

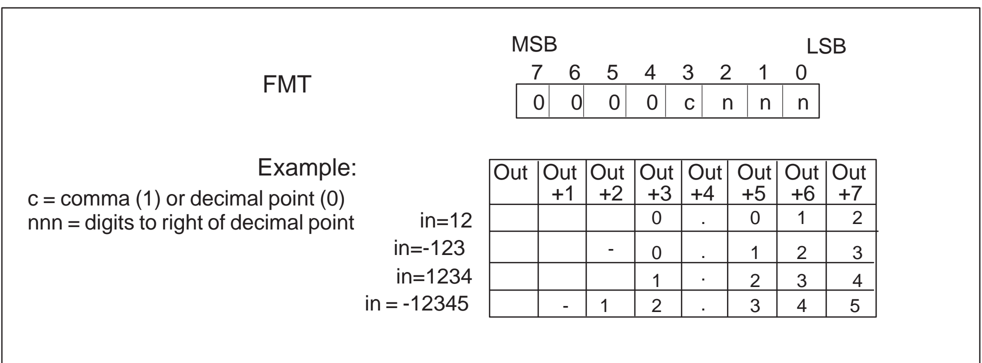
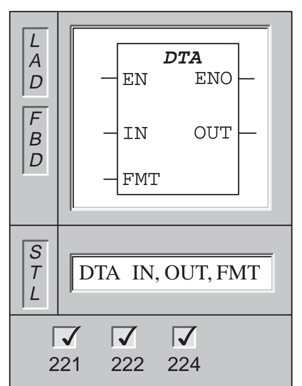


Figure 9-48 FMT Operand for the ITA Instruction

Double Integer to ASCII



The **Double Integer to ASCII** instruction converts a double word (IN) to an ASCII string. The format (FMT) specifies the conversion precision to the right of the decimal. The resulting conversion is placed in 12 consecutive bytes beginning with OUT.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), no output (illegal format)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SD, SMD, LD, HC, Constant, AC, *VD, *AC, *LD	DINT
FMT	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VB, IB, QB, MB, SMB, LB, *VD, *AC, SB, *LD	BYTE

The format operand (FMT) for the DTA instruction is defined in Figure 9-49. The size of the output buffer is always 12 bytes. The number of digits to the right of the decimal point in the output buffer is specified by the nnn field. The valid range of the nnn field is 0 to 5. Specifying 0 digits to the right of the decimal point causes the value to be displayed without a decimal point. For values of nnn bigger than 5, the output buffer is filled with ASCII spaces. The c bit specifies the use of either a comma (c=1) or a decimal point (c=0) as the separator between the whole number and the fraction. The upper 4 bits must be zero. The output buffer is formatted in accord with the following rules:

1. Positive values are written to the output buffer without a sign.
2. Negative values are written to the output buffer with a leading minus sign (-).
3. Leading zeros to the left of the decimal point (except the digit adjacent to the decimal point) are suppressed.
4. Values are right-justified in the output buffer.

Figure 9-49 gives examples of values that are formatted using a decimal point (c = 0) with four digits to the right of the decimal point (nnn = 100).

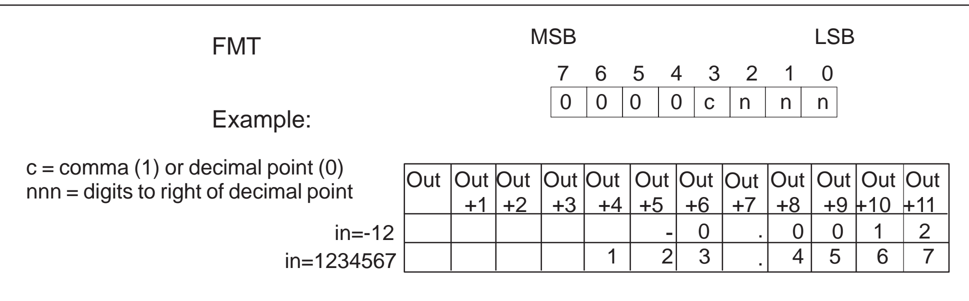
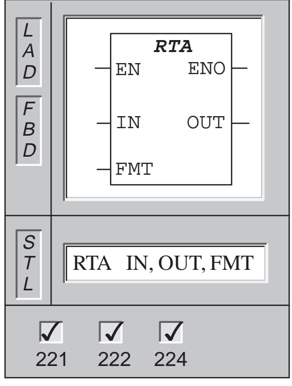


Figure 9-49 FMT Operand for DTA Instruction

Real to ASCII



The **Real to ASCI** instruction converts a floating point value (IN) to an ASCII string. The format (FMT) specifies the conversion precision to the right of the decimal, and whether the decimal point is shown as a decimal point or a period, and the output buffer size. The resulting conversion is placed in an output buffer beginning with OUT. The length of the resulting ASCII string is the size of the output buffer, and can be specified to a size ranging from 3 to 15.

Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), no output (illegal format or buffer too small)

Inputs/Outputs	Operands	Data Types
IN	VD, ID, QD, MD, SD, SMD, LD, AC, *VD, *AC, *LD	REAL
FMT	VB, IB, QB, MB, SMB, LB, AC, Constant, *VD, *AC, SB, *LD	BYTE
OUT	VB, IB, QB, MB, SMB, LB, *VD, *AC, SB, *LD	BYTE

The format operand (FMT) for the RTA instruction is defined in Figure 9-50. The size of the output buffer is specified by the ssss field. A size of 0, 1, or 2 bytes is not valid. The number of digits to the right of the decimal point in the output buffer is specified by the nnn field. The valid range of the nnn field is 0 to 5. Specifying 0 digits to the right of the decimal point causes the value to be displayed without a decimal point. The output buffer is filled with ASCII spaces for values of nnn bigger than 5 or when the specified output buffer is too small to store the converted value. The c bit specifies the use of either a comma (c=1) or a decimal point (c=0) as the separator between the whole number and the fraction. The output buffer is formatted in accord with the following rules:

1. Positive values are written to the output buffer without a sign.
2. Negative values are written to the output buffer with a leading minus sign (-).
3. Leading zeros to the left of the decimal point (except the digit adjacent to the decimal point) are suppressed.
4. Values to the right of the decimal point are rounded to fit in the specified number of digits to the right of the decimal point.
5. The size of the output buffer must be a minimum of three bytes more than the number of digits to the right of the decimal point.
6. Values are right-justified in the output buffer.

Figure 9-50 gives examples of values that are formatted using a decimal point (c=0) with one digit to the right of the decimal point (nnn=001) and a buffer size of six bytes (ssss=0110).

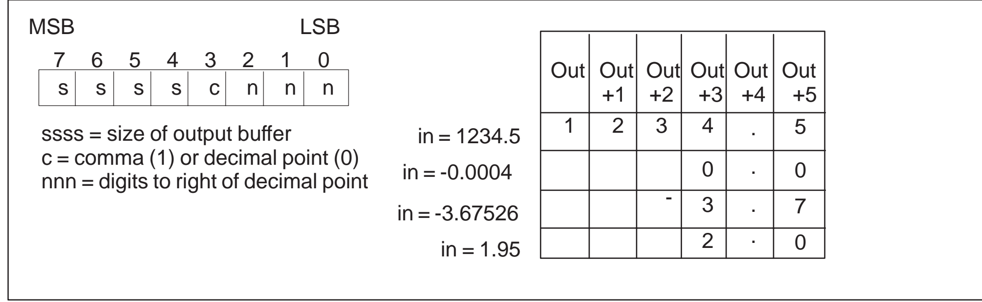


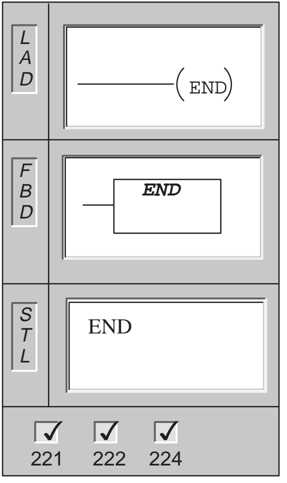
Figure 9-50 FMT Operand for RTA Instruction

Note

The floating point format used by the S7-200 CPU supports a maximum of 7 significant digits. Attempting to display more than the 7 significant digits produces a rounding error.

9.15 SIMATIC Program Control Instructions

End



The **Conditional END** instruction terminates the main user program based upon the condition of the preceding logic.

Operands: None

Data Types: None

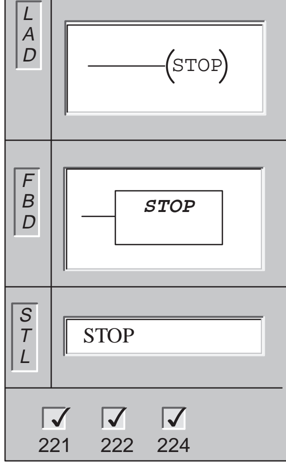
Note

You can use the Conditional END instruction in the main program, but you cannot use it in either subroutines or interrupt routines.

Note

Micro/WIN 32 automatically adds an unconditional end to the main user program.

Stop

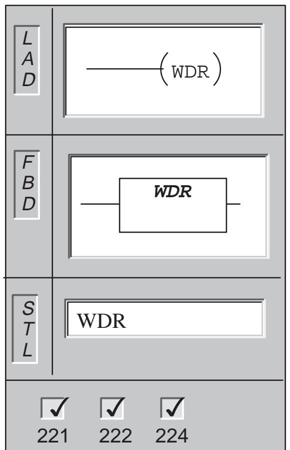


The **STOP** instruction terminates the execution of your program immediately by causing a transition of the CPU from RUN to STOP mode.

Operands: None

If the STOP instruction is executed in an interrupt routine, the interrupt routine is terminated immediately, and all pending interrupts are ignored. Remaining actions in the current scan cycle are completed, including execution of the main user program, and the transition from RUN to STOP mode is made at the end of the current scan.

Watchdog Reset



The **Watchdog Reset** instruction allows the CPU system watchdog timer to be retriggered. This extends the time that the scan is allowed to take without getting a watchdog error.

Operands: None

Considerations for Using the WDR Instruction to Reset the Watchdog Timer

You should use the Watchdog Reset instruction carefully. If you use looping instructions either to prevent scan completion, or to delay excessively the completion of the scan, the following processes are inhibited until the scan cycle is completed.

- Communication (except Freeport Mode)
- I/O updating (except Immediate I/O)
- Force updating
- SM bit updating (SM0, SM5 to SM29 are not updated)
- Run-time diagnostics
- 10-ms and 100-ms timers will not properly accumulate time for scans exceeding 25 seconds
- STOP instruction, when used in an interrupt routine

Note

If you expect your scan time to exceed 300 ms, or if you expect a burst of interrupt activity that may prevent returning to the main scan for more than 300 ms, you should use the WDR instruction to re-trigger the watchdog timer.

Changing the switch to the STOP position will cause the CPU to transition to STOP mode within 1.4 seconds.

Stop, End, and WDR Example

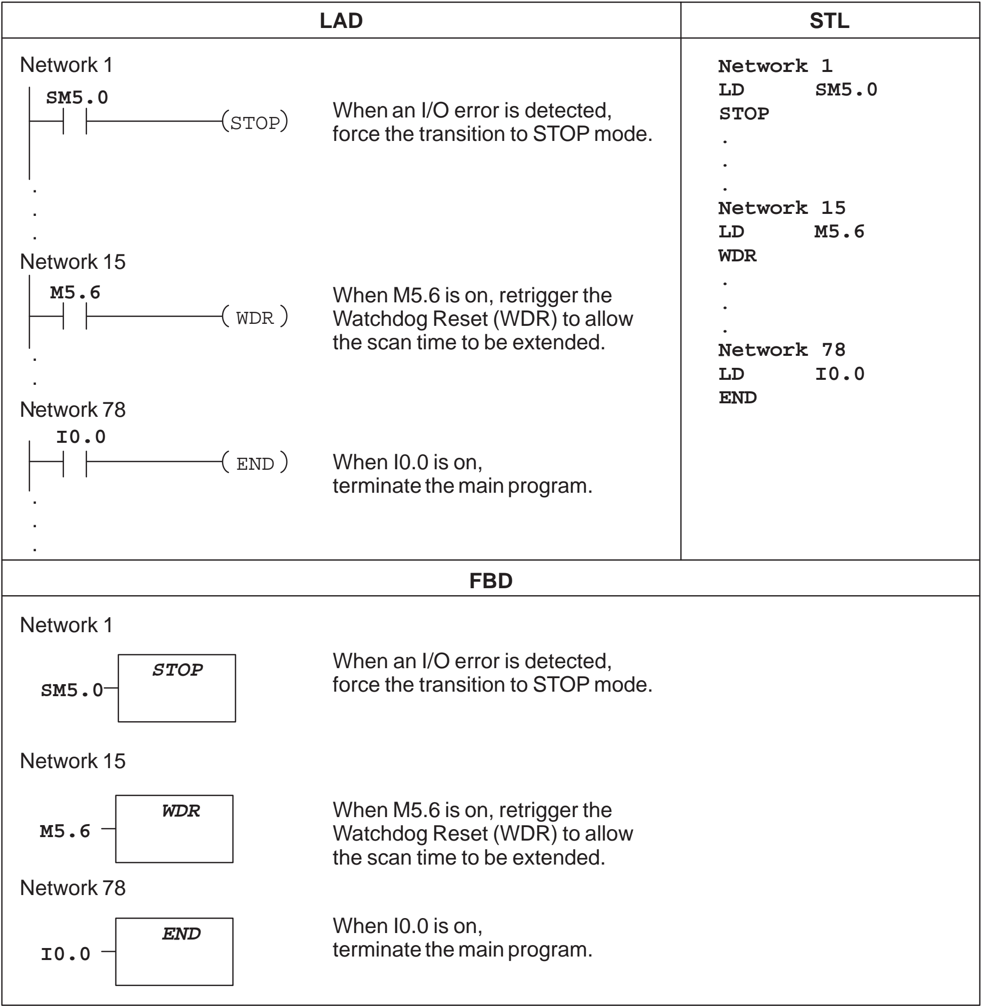
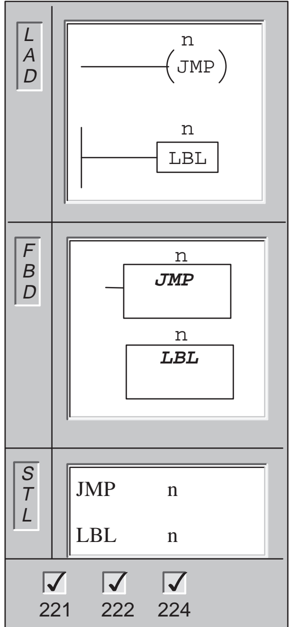


Figure 9-51 Example of Stop, End, and WDR Instructions for LAD, STL, and FBD

Jump to Label, Label



The **Jump to Label** instruction performs a branch to the specified label (n) within the program. When a jump is taken, the top of stack value is always a logical 1.

The **Label** instruction marks the location of the jump destination (n).

Operands: n: 0 to 255

Data Types: WORD

Both the Jump and corresponding Label must be in the main program, a subroutine, or an interrupt routine. You cannot jump from the main program to a label in either a subroutine or an interrupt routine. Likewise, you cannot jump from a subroutine or interrupt routine to a label outside that subroutine or interrupt routine.

Jump to Label Example

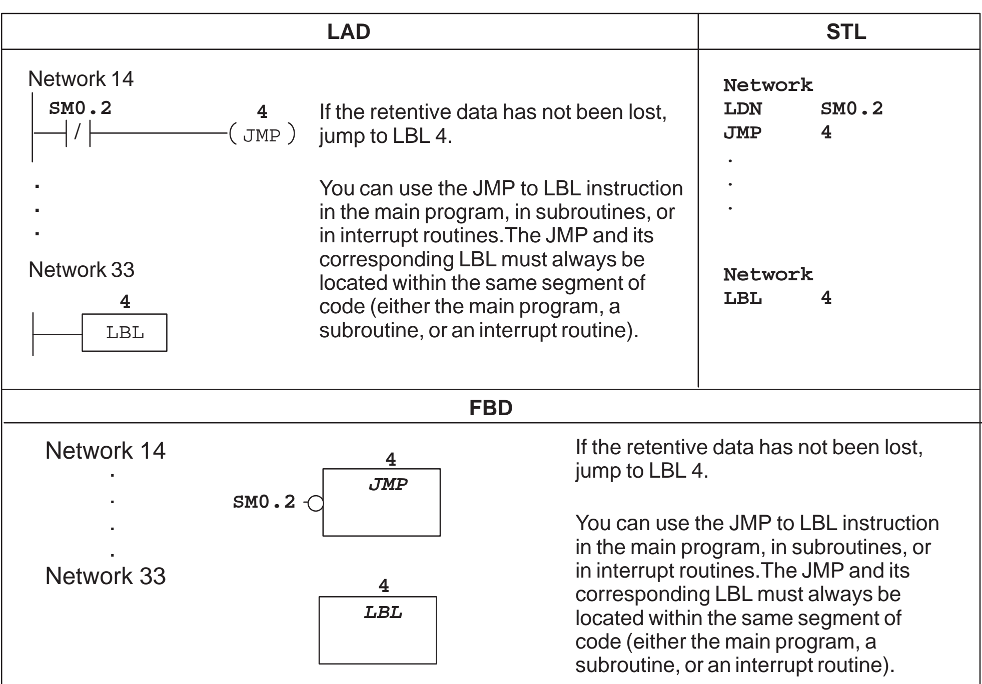
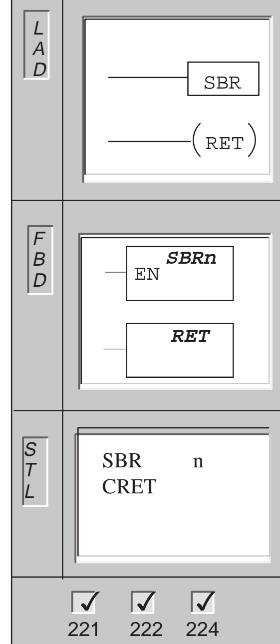


Figure 9-52 Example of Jump to Label and Label Instructions for LAD, STL, and FBD

Subroutine, Return from Subroutine



The Call **Subroutine** instruction transfers control to the subroutine (n). You can use a Call Subroutine instruction with or without parameters. To add a subroutine, select **Edit > Insert > Subroutine** from the menu.

The **Conditional Return from Subroutine** instruction is used to terminate a subroutine based upon the preceding logic.

Operands: n: Constant

Data Types: BYTE

Once the subroutine completes its execution, control returns to the instruction that follows the Call Subroutine.

Figure 9-55 shows an example of the Call Subroutine, and Return from Subroutine instructions.

Error conditions that set ENO for Call Subroutine with parameters = 0:

SM4.3 (run-time), 0008 (maximum subroutine nesting exceeded)

Note

Micro/WIN 32 automatically adds a return from each subroutine.

You can nest subroutines (place a subroutine call within a subroutine), to a depth of eight. Recursion (a subroutine that calls itself) is not prohibited, but you should use caution when using recursion with subroutines.

When a subroutine is called, the entire logic stack is saved, the top of stack is set to one, all other stack locations are set to zero, and control is transferred to the called subroutine. When this subroutine is completed, the stack is restored with the values saved at the point of call, and control is returned to the calling routine.

Accumulators are common to subroutines and the calling routine. No save or restore operation is performed on accumulators due to subroutine use.

Calling a Subroutine With Parameters

Subroutines may contain passed parameters. The parameters are defined in the local variable table of the subroutine (Figure 9-53). The parameters must have a symbol name (maximum of 8 characters), a variable type, and a data type. Sixteen parameters can be passed to or from a subroutine.

The variable type field in the local variable table defines whether the variable is passed into the subroutine (IN), passed into and out of the subroutine (IN_OUT), or passed out of the subroutine (OUT). The characteristics of the parameter types are as follows:

- IN: parameters are passed into the subroutine. If the parameter is a direct address (such as VB10), the value at the specified location is passed into the subroutine. If the parameter is an indirect address (such as *AC1), the value at the location pointed to is passed into the subroutine. If the parameter is a data constant (16#1234), or an address (VB100), the constant or address value is passed into the subroutine.
- IN_OUT: the value at the specified parameter location is passed into the subroutine and the result value from the subroutine is returned to the same location. Constants (such as 16#1234) and addresses (such as &VB100) are not allowed for input/output parameters.
- OUT: The result value from the subroutine is returned to the specified parameter location. Constants (such as 16#1234) and addresses (such as &VB100) are not allowed as output.
- TEMP:
  Any local memory that is not used for passed parameters may be used for temporary storage within the subroutine.

To add a parameter entry, place the cursor on the variable type field of the type (IN, IN_OUT<OUT) that you want to add. Click the right mouse button to get a menu of options. Select the Insert option and then the Row Below option. Another parameter entry of the selected type appears below the current entry.

	Name	Var. Type	Data Type	Comment
	EN	IN	BOOL	
L0.0	IN1	IN	BOOL	
LB1	IN2	IN	BYTE	
LB2.0	IN3	IN	BOOL	
LD3	IN4	IN	DWORD	
LW7	IN/OUT1	IN/OUT	WORD	
LD9	OUT1	OUT	DWORD	
		TEMP		

Figure 9-53 STEP 7-Micro/WIN 32 Local Variable Table

The data type field in the local variable table defines the size and format of the parameter. The parameter types are:

- Power Flow: Boolean power flow is allowed only for bit (Boolean) inputs. This declaration tells STEP 7-Micro/WIN 32 that this input parameter is the result of power flow based on a combination of bit logic instructions. Boolean power flow inputs must appear first in the local variable table before any other type input. Only input parameters are allowed to be used this way. The enable input (EN) and the IN1 inputs in Figure 9-54 use Boolean logic.
- Boolean - This data type is used for single bit inputs and outputs. IN2 in Figure 9-54 is a Boolean input.
- Byte, Word, Dword - These data types identify an unsigned input or output parameter of 1, 2, or 4 bytes respectively.
- INT, DINT - These data types identify signed input or output parameters of 2 or 4 bytes respectively.
- Real - This data type identifies a single precision (4 byte) IEEE floating point value.

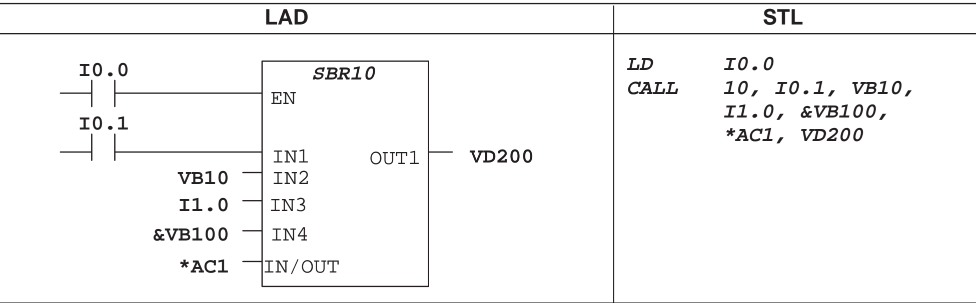


Figure 9-54 Subroutine Call in LAD and STL

Address parameters such as IN4 in Figure 9-54 (&VB100) are passed into a subroutine as a Dword (unsigned double word) value. The type of a constant parameter must be specified for the parameter in the calling routine with a constant describer in front of the constant value. For example, to pass an unsigned double word constant with a value of 12,345 as a parameter, the constant parameter must be specified as DW#12345. If the constant describer is omitted from parameter, the constant may be assumed to be a different type.

There are no automatic data type conversions performed on the input or output parameters. For example, if the local variable table specifies that a parameter has the data type Real, and in the calling routine a double word (Dword) is specified for that parameter, the value in the subroutine will be a double word.

When values are passed to a subroutine, they are placed into the local memory of the subroutine. The left-most column of the local variable table (see Figure 9-53) shows the local memory address for each passed parameter. Input parameter values are copied to the subroutine's local memory when the subroutine is called. Output parameter values are copied form the subroutine's local memory to the specified output parameter addresses when the subroutine execution is complete.

The data element size and type are represented in the coding of the parameters. Assignment of parameter values to local memory in the subroutine is as follows:

- Parameter values are assigned to local memory in the order specified by the call subroutine instruction with parameters starting at L.0.
- One to eight consecutive bit parameter values are assigned to a single byte starting with Lx.0 and continuing to Lx.7.
- Byte, word, and double word values are assigned to local memory on byte boundaries (LBx, LWx, or LDx).

In the Call Subroutine instruction with parameters, parameters must be arranged in order with input parameters first, followed by input/output parameters, and then followed by output parameters.

If you are programming in STL, the format of the CALL instruction is:

```
CALL    subroutine number, parameter 1, parameter 2, ... , parameter
```

Error conditions that set ENO for Call Subroutine with parameters = 0:
SM4.3 (run-time), 0008 (maximum subroutine nesting exceeded)

Subroutine, and Return from Subroutine Example

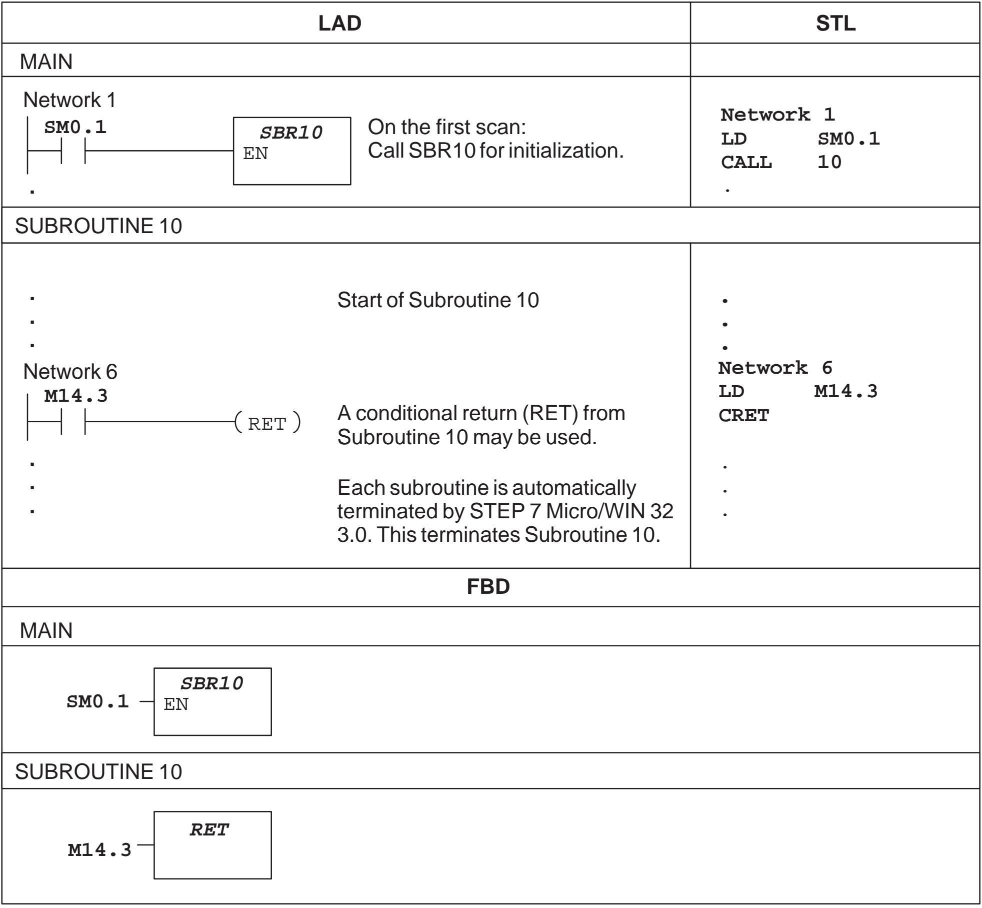
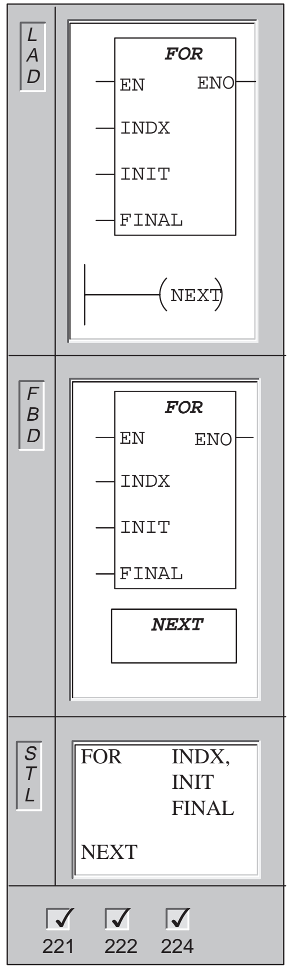


Figure 9-55 Example of Subroutine Instructions for LAD, FBD, and STL

For, Next



The **FOR** instruction executes the instructions between the FOR and the NEXT. You must specify the index value or current loop count (INDX), the starting value (INIT), and the ending value (FINAL).

The **NEXT** instruction marks the end of the FOR loop, and sets the top of the stack to 1.

For example, given an INIT value of 1 and a FINAL value of 10, the instructions between the FOR and the NEXT are executed 10 times with the INDX value being incremented: 1, 2, 3, ...10.

If the starting value is greater than the final value, the loop is not executed. After each execution of the instructions between the FOR and the NEXT instruction, the INDX value is incremented and the result is compared to the final value. If the INDX is greater than the final value, the loop is terminated.

For: Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
INDX	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, *VD, *AC, *LD	INT
INIT	VW, IW, QW, MW, SW, SMW, T, C, AC, LW, AIW, Constant, *VD, *AC, *LD	INT
FINAL	VW, IW, QW, MW, SW, SMW, LW, T, C, AC, AIW, Constant, *VD, *AC, *LD	INT

Here are some guidelines for using the FOR/NEXT loop:

- If you enable the FOR/NEXT loop, it continues the looping process until it finishes the iterations, unless you change the final value from within the loop itself. You can change the values while the FOR/NEXT is in the looping process.
- When the loop is enabled again, it copies the initial value into the index value (current loop number). The FOR/NEXT instruction resets itself the next time it is enabled.

Use the FOR/NEXT instructions to delineate a loop that is repeated for the specified count. Each FOR instruction requires a NEXT instruction. You can nest FOR/NEXT loops (place a FOR/NEXT loop within a FOR/NEXT loop) to a depth of eight.

For/Next Example

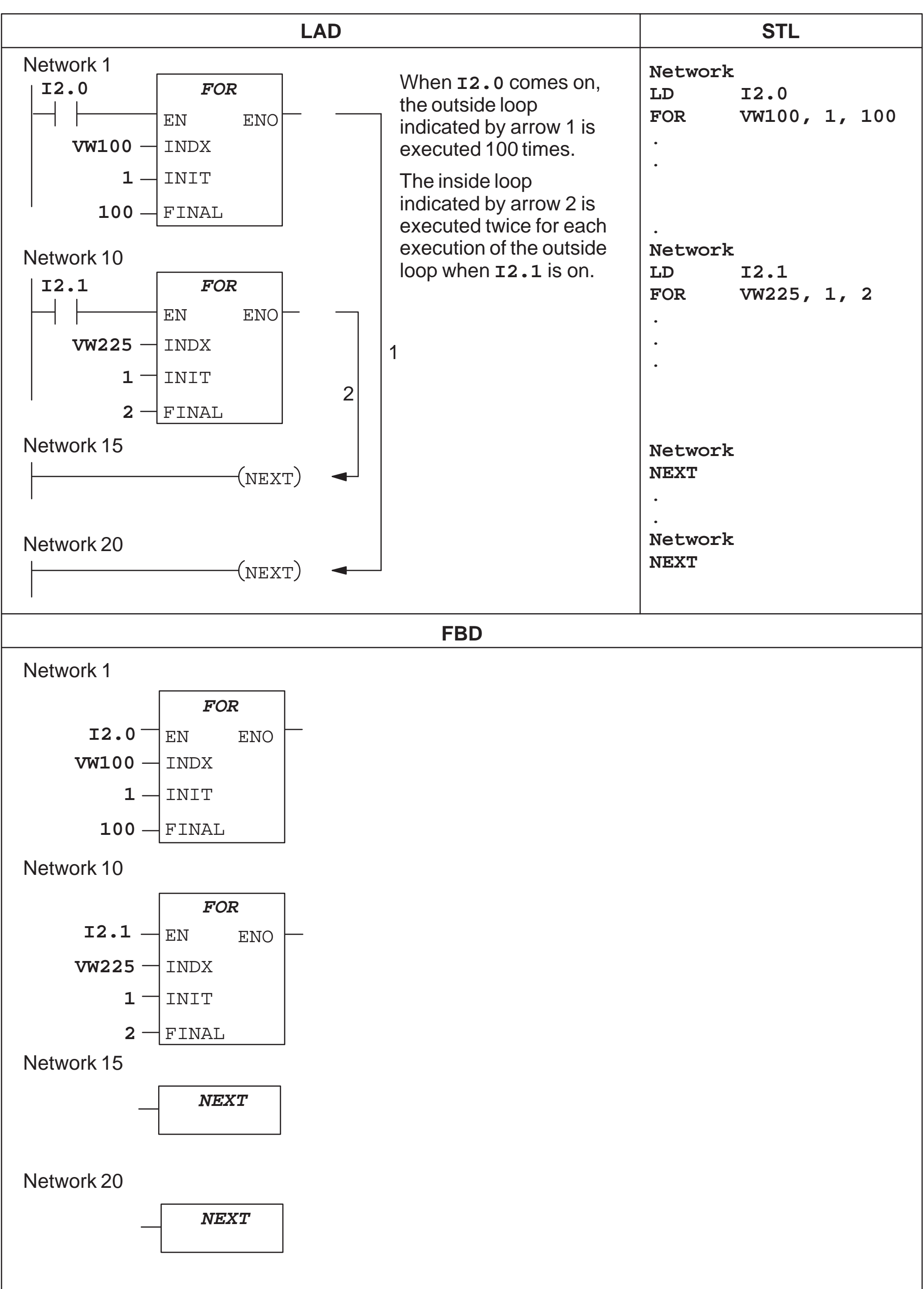
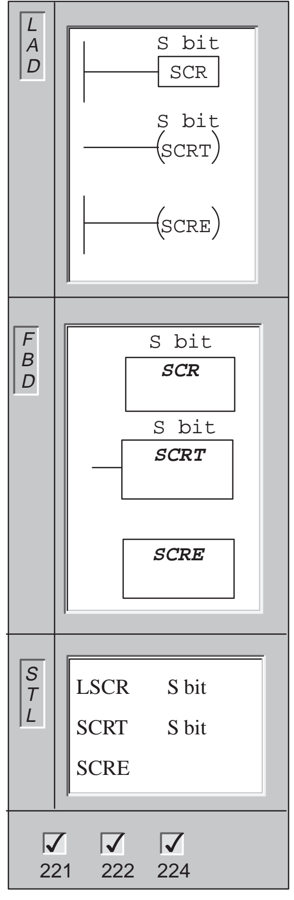


Figure 9-56 Example of For/Next Instructions for LAD and STL

Sequence Control Relay



The **Load Sequence Control Relay** instruction marks the beginning of an SCR segment. When the S bit is on, power flow is enabled to the SCR segment. The SCR segment must be terminated with an SCRE instruction.

The **Sequence Control Relay Transition** instruction identifies the SCR bit to be enabled (the next S bit to be set). When power flows to the coil or FBD box, the referenced S bit is turned on and the S bit of the LSCR instruction (that enabled this SCR segment) is turned off.

The **Sequence Control Relay End** instruction marks the end of an SCR segment.

Inputs/Outputs	Operands	Data Types
n	S	BOOL

Understanding SCR Instructions

In LAD and STL, Sequence Control Relays (SCRs) are used to organize machine operations or steps into equivalent program segments. SCRs allow logical segmentation of the control program.

The LSCR instruction loads the SCR and logic stacks with the value of the S bit referenced by the instruction. The SCR segment is energized or de-energized by the resulting value of the SCR stack. The top of the logic stack is loaded to the value of the referenced S bit so that boxes or output coils can be tied directly to the left power rail without an intervening contact. Figure 9-57 shows the S stack and the logic stack and the effect of executing the LSCR instruction.

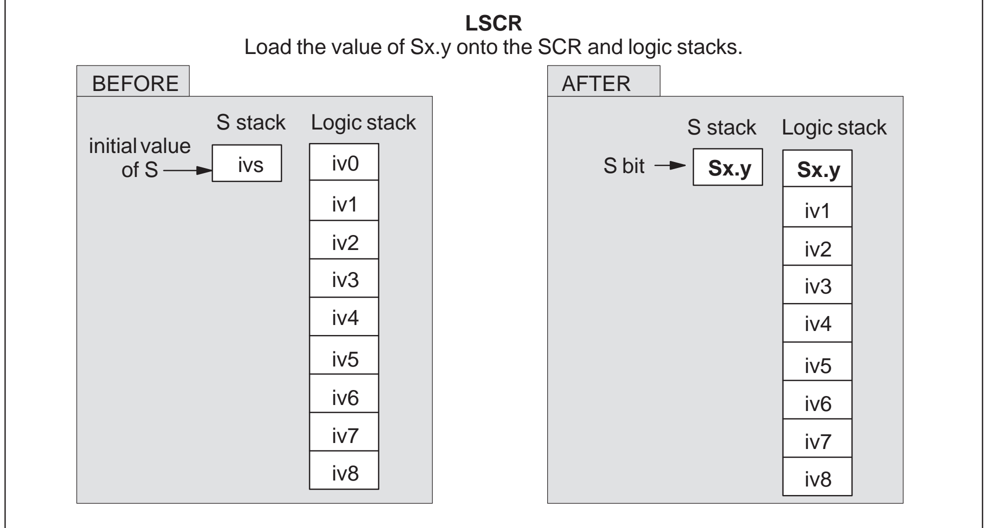


Figure 9-57 Effect of LSCR on the Logic Stack

The following is true of Sequence Control Relay instructions:

- All logic between the LSCR and the SCRE instructions make up the SCR segment and are dependent upon the value of the S stack for its execution. Logic between the SCRE and the next LSCR instruction have no dependency upon the value of the S stack.
- The SCRT instruction sets an S bit to enable the next SCR and also resets the S bit that was loaded to enable this section of the SCR segment.

Restrictions

Restrictions for using SCRs follow:

- You cannot use the same S bit in more than one routine. For example, if you use S0.1 in the main program, do not use it in the subroutine.
- You cannot use the JMP and LBL instructions in an SCR segment. This means that jumps into, within, or out of an SCR segment are not allowed. You can use jump and label instructions to jump around SCR segments.
- You cannot use the FOR, NEXT, and END instructions in an SCR segment.

SCR Example

Figure 9-58 shows an example of the operation of SCRs.

- In this example, the first scan bit SM0.1 is used to set S0.1, which will be the active State 1 on the first scan.
- After a 2-second delay, T37 causes a transition to State 2. This transition deactivates the State 1 SCR (S0.1) segment and activates the State 2 SCR (S0.2) segment.

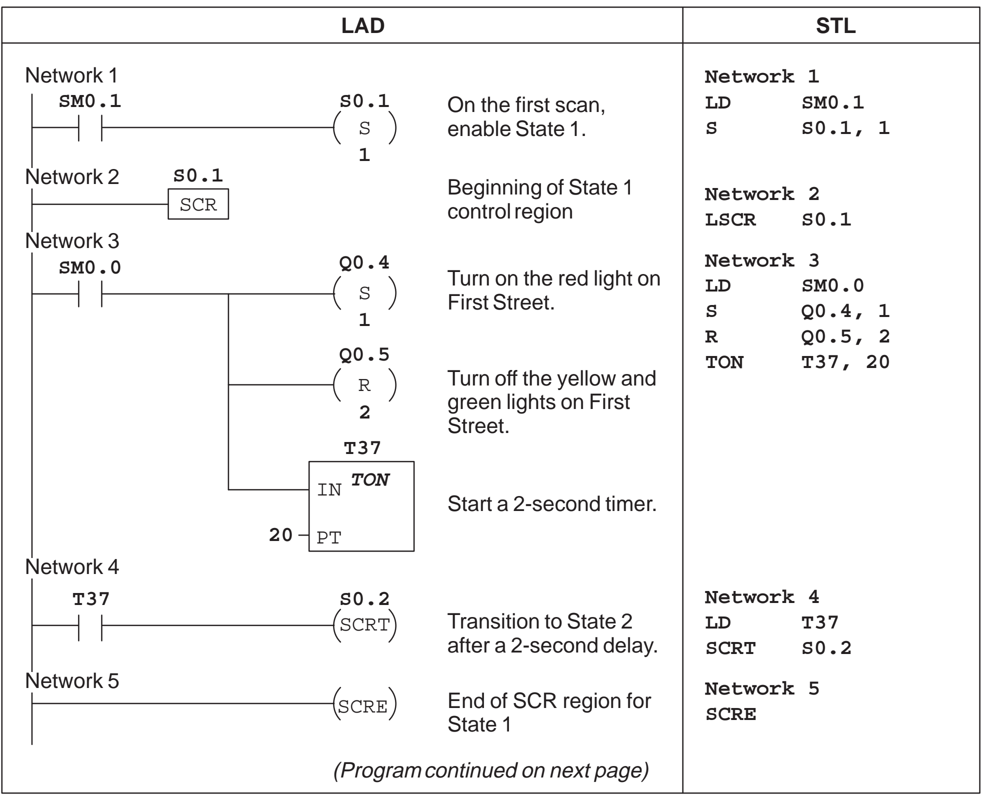


Figure 9-58 Example of Sequence Control Relays (SCRs)

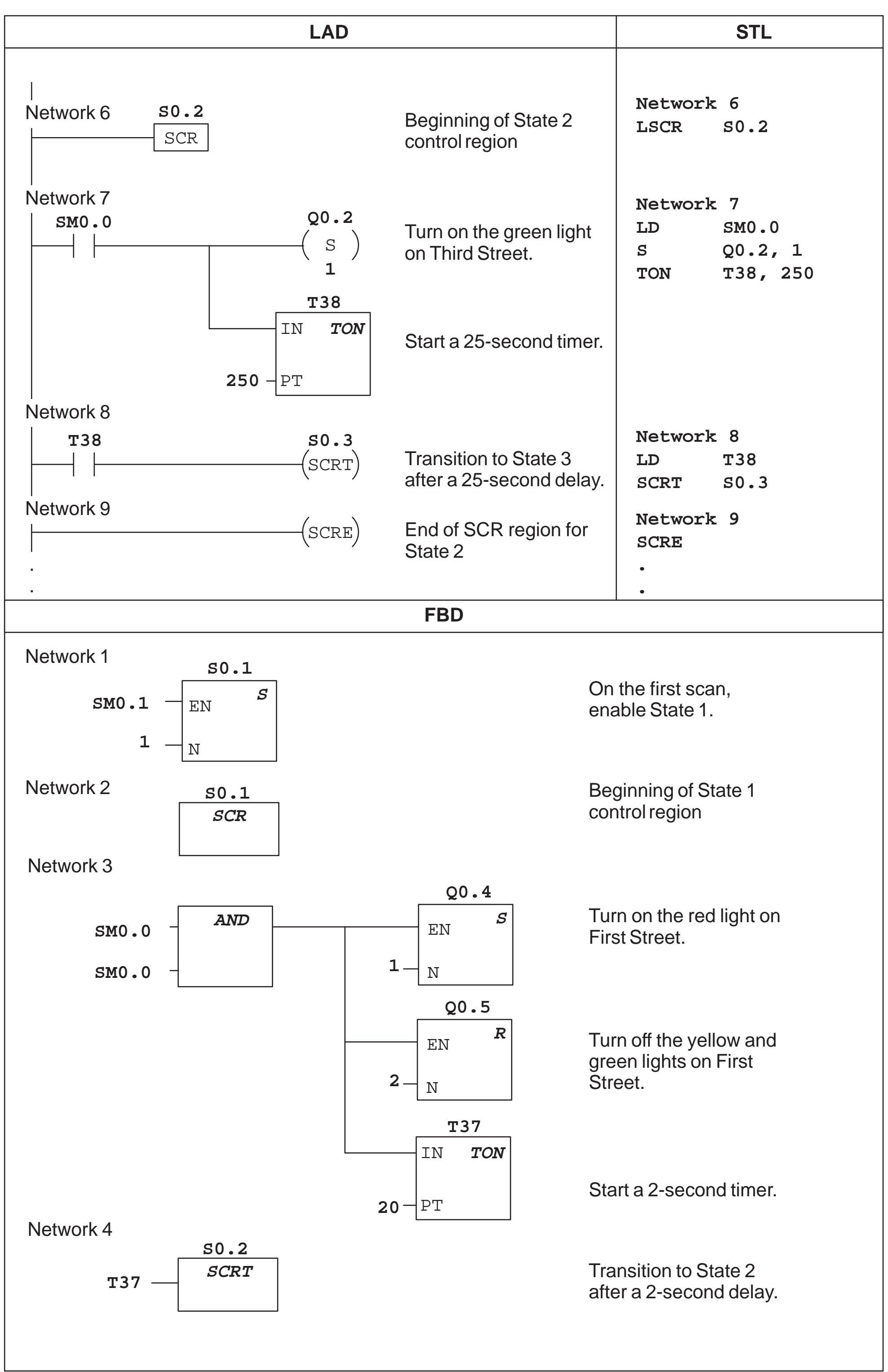


Figure 9-58 Example of Sequence Control Relays (SCRs), continued

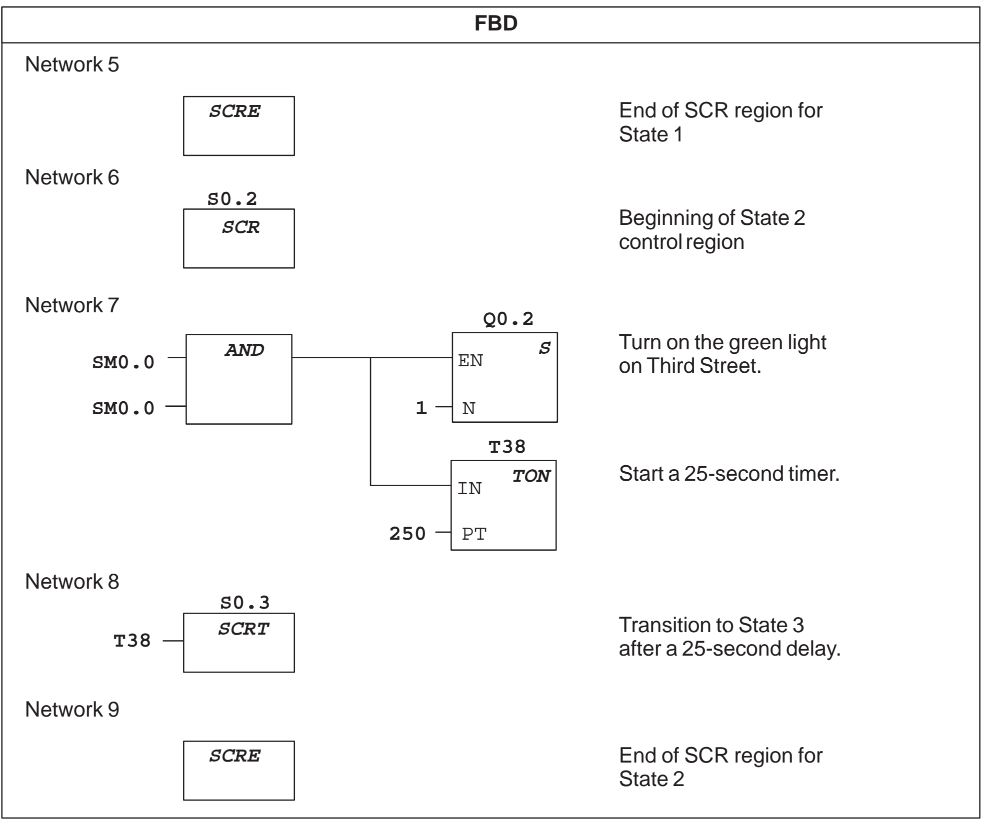


Figure 9-58 Example of Sequence Control Relays (SCRs), continued

Divergence Control

In many applications, a single stream of sequential states must be split into two or more different streams. When a stream of control diverges into multiple streams, all outgoing streams must be activated simultaneously. This is shown in Figure 9-59.

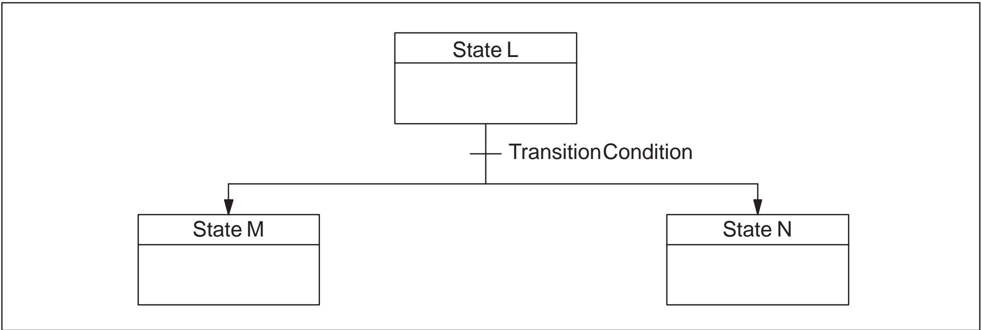


Figure 9-59 Divergence of Control Stream

The divergence of control streams can be implemented in an SCR program by using multiple SCRT instructions enabled by the same transition condition, as shown in Figure 9-60.

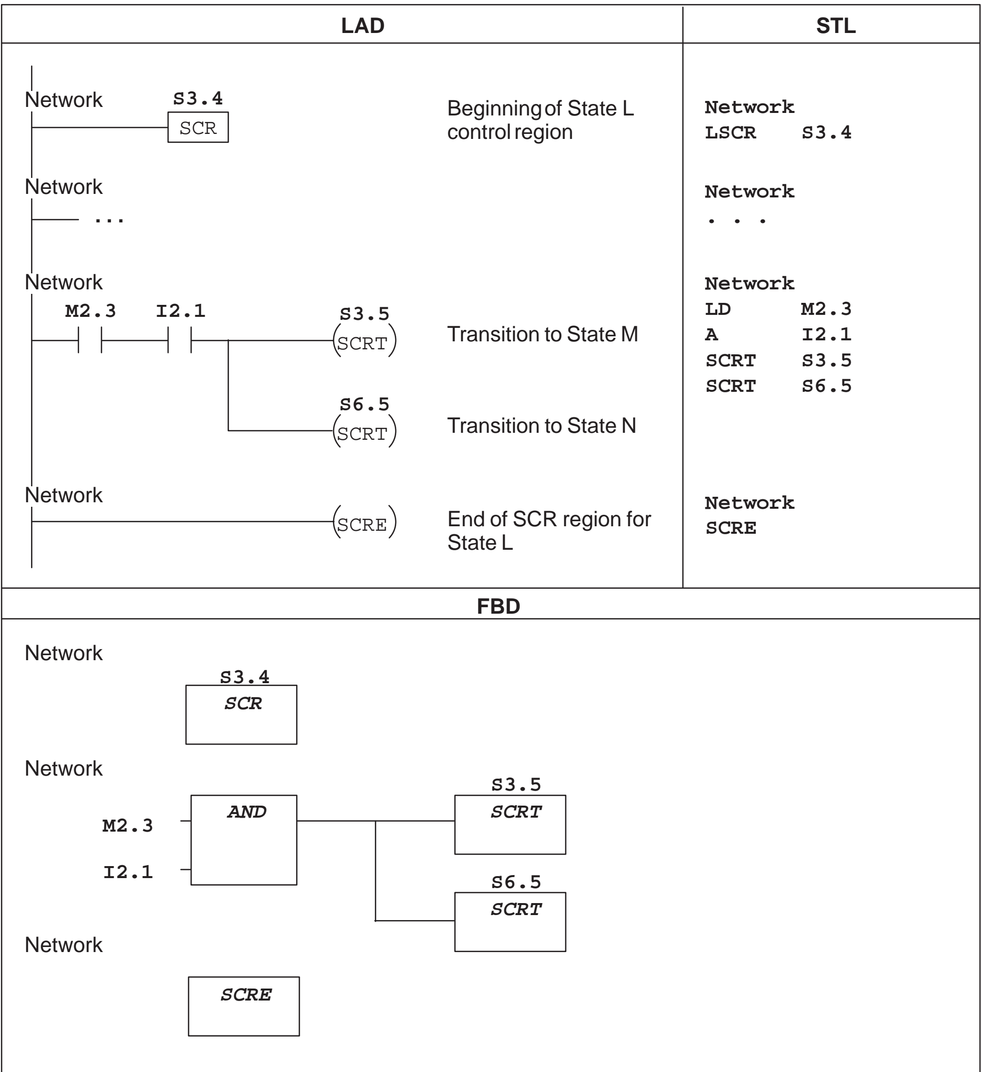


Figure 9-60 Example of Divergence of Control Streams

Convergence Control

A similar situation arises when two or more streams of sequential states must be merged into a single stream. When multiple streams merge into a single stream, they are said to converge. When streams converge, all incoming streams must be complete before the next state is executed. Figure 9-61 depicts the convergence of two control streams.

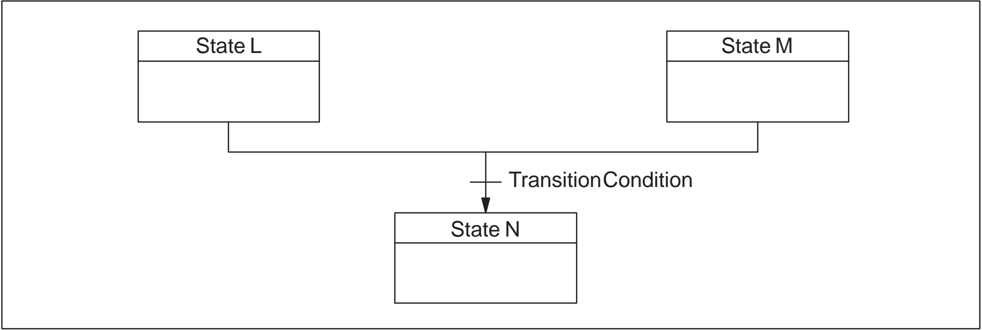


Figure 9-61 Convergence of Control Streams

The convergence of control streams can be implemented in an SCR program by making the transition from state L to state L’ and by making the transition from state M to state M’. When both SCR bits representing L’ and M’ are true, state N can the enabled as shown in Figure 9-62.

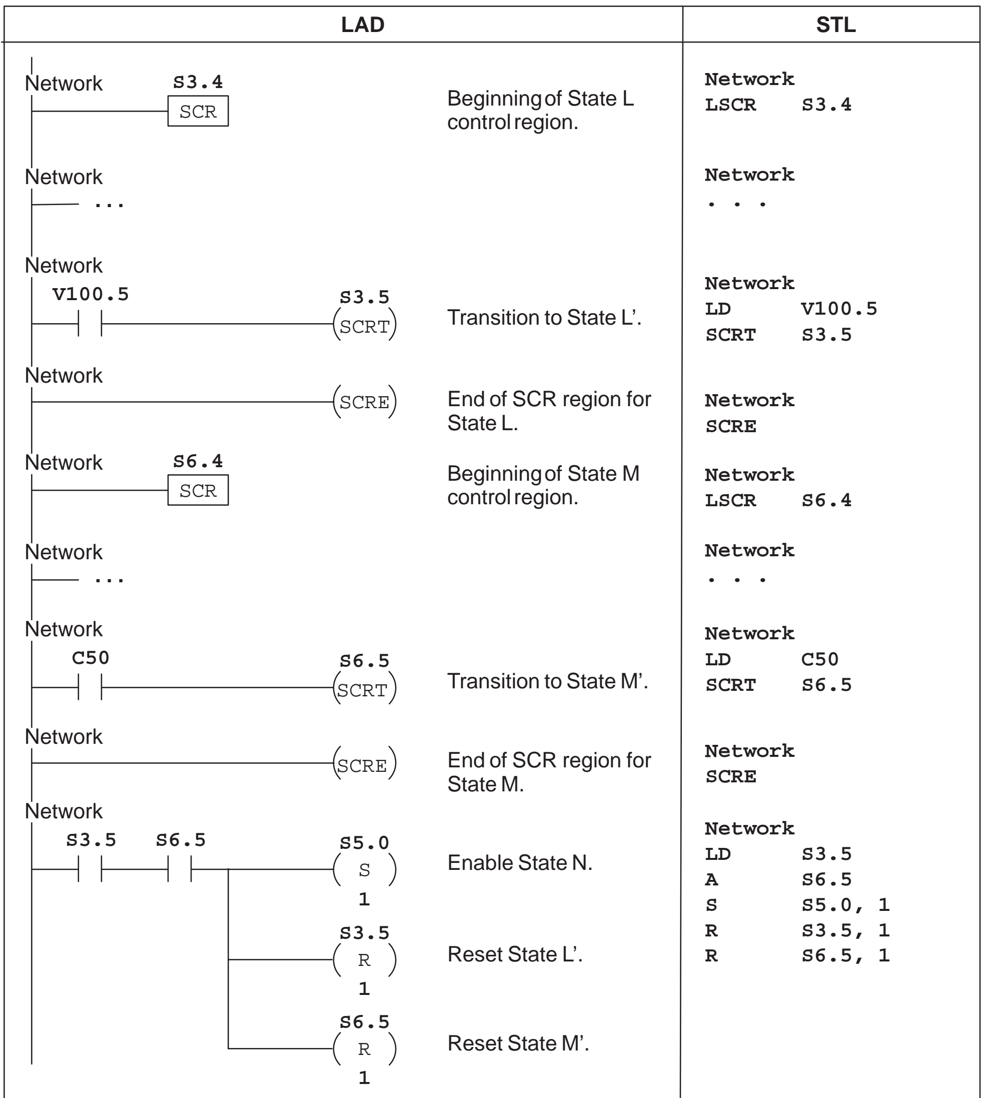


Figure 9-62 Example of Convergence of Control Streams

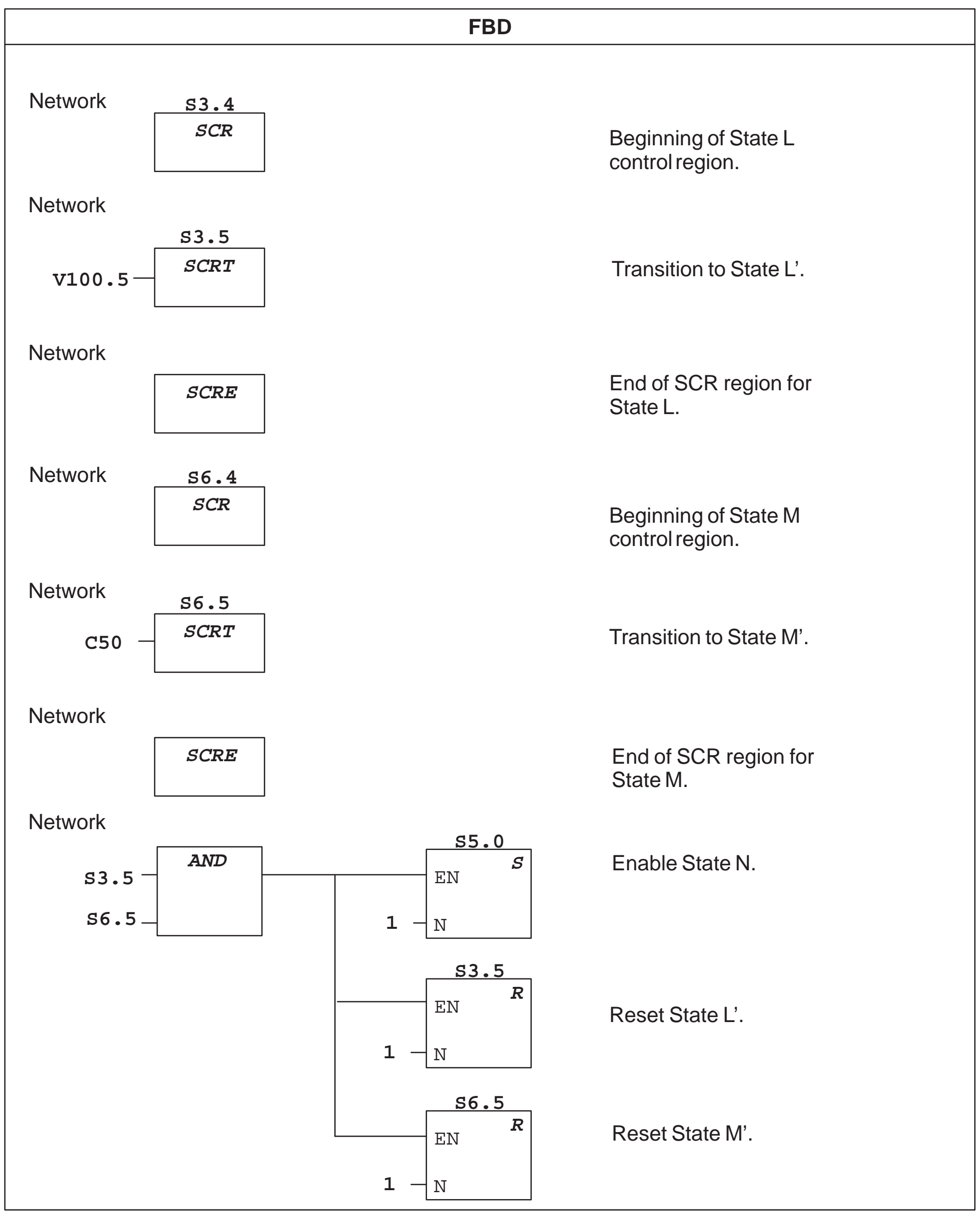


Figure 9-62 Example of Convergence of Control Streams, continued

In other situations, a control stream may be directed into one of several possible control streams, depending upon which transition condition comes true first. Such a situation is depicted in Figure 9-63.

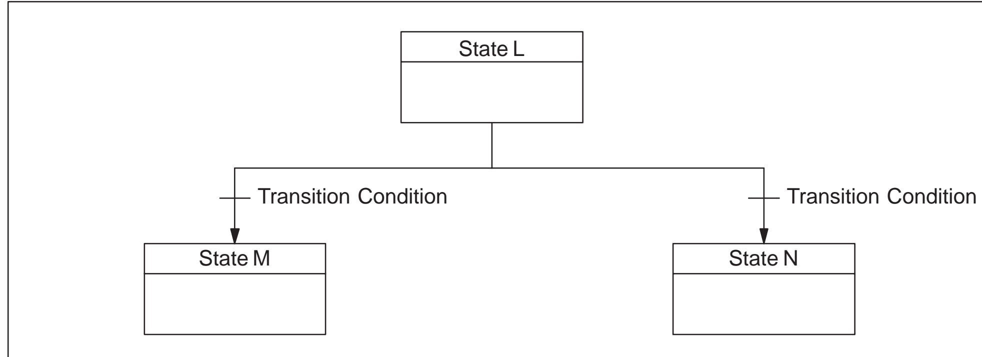


Figure 9-63 Divergence of Control Stream, Depending on Transition Condition

An equivalent SCR program is shown in Figure 9-64.

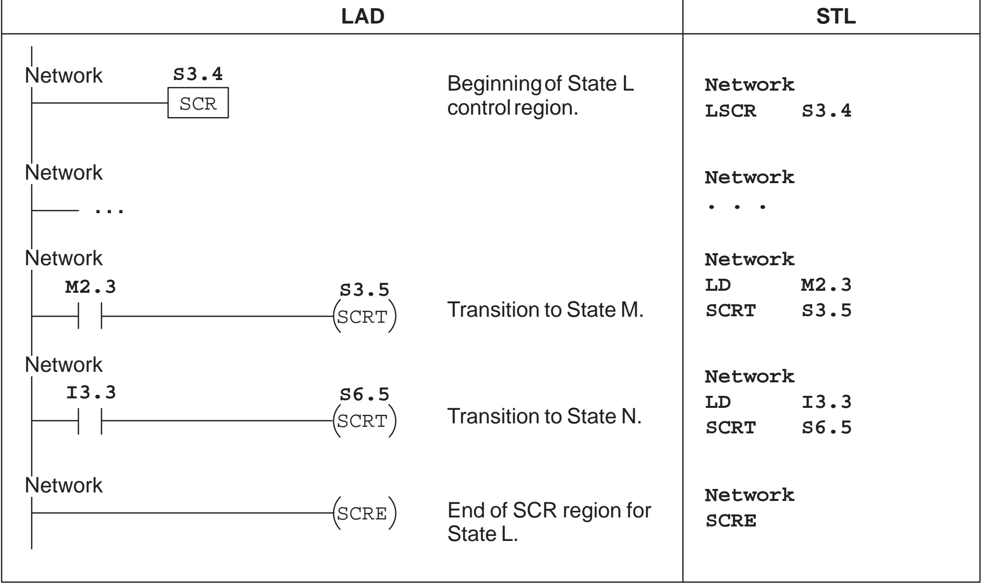


Figure 9-64 Example of Conditional Transitions

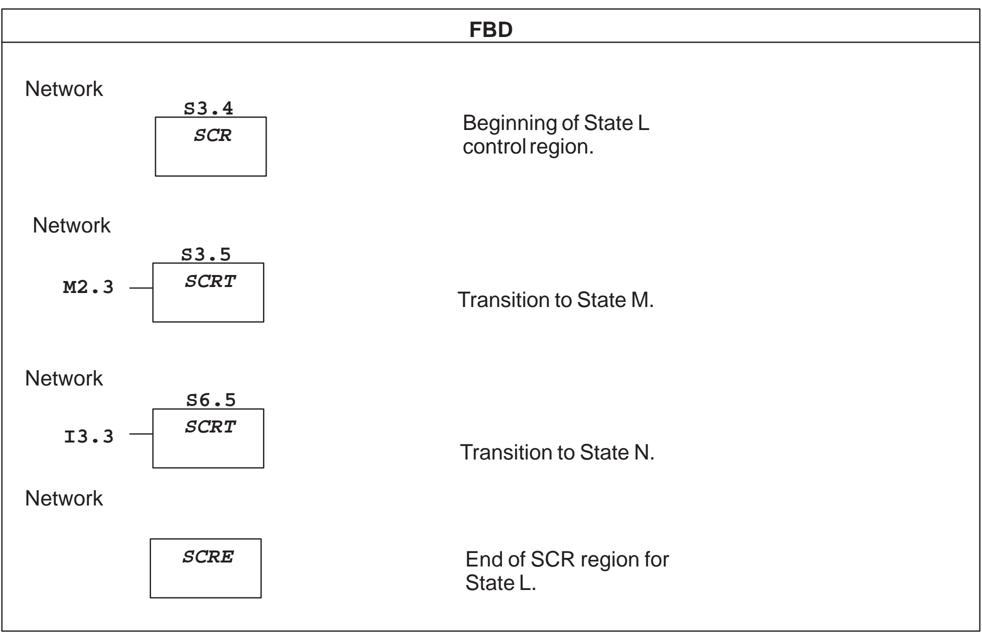
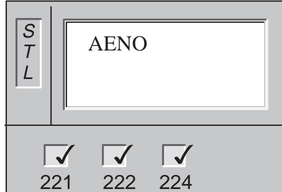


Figure 9-64 Example of Conditional Transitions (continued)

ENO



ENO is a Boolean output for boxes in LAD and FBD. If a box has power flow at the EN input and is executed without error, the ENO output passes power flow to the next element. ENO can be used as an enable bit that indicates the successful completion of an instruction.

The ENO bit is used with the top of stack to affect power flow for execution of subsequent instructions.

STL instructions do not have an EN input; the top of the stack must be a logic 1 for the instruction to be executed.

In STL there is no ENO output, but the STL instructions that correspond to LAD and FBD instructions with ENO outputs do set a special ENO bit. This bit is accessible with the **And ENO** (AENO) instruction. AENO can be used to generate the same effect as the ENO bit of a box. The AENO instruction is only available in STL.

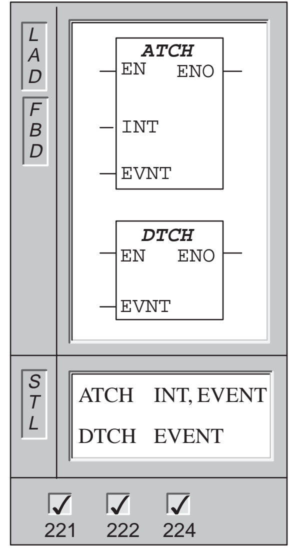
AENO will perform a logical AND of the ENO bit and the top of stack. The result of the AND operation is the new top of stack.

Operands: None

Data Types: None

9.16 SIMATIC Interrupt and Communications Instructions

Attach Interrupt, Detach Interrupt



The **Attach Interrupt** instruction associates an interrupt event (EVNT) with an interrupt routine number (INT), and enables the interrupt event.

The **Detach Interrupt** instruction disassociates an interrupt event (EVNT) from all interrupt routines, and disables the interrupt event.

Attach Interrupt: Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
INT	Constant (CPU 222: 0-12, 19-23, 27-33; CPU 224: 0-23, 27-33)	BYTE
EVNT	Constant (CPU 222: 0-12, 19-23, 27-33; CPU 224: 0-23, 27-33)	BYTE

Understanding Attach and Detach Interrupt Instructions

Before an interrupt routine can be invoked, an association must be established between the interrupt event and the program segment that you want to execute when the event occurs. Use the Attach Interrupt instruction (ATCH) to associate an interrupt event (specified by the interrupt event number) and the program segment (specified by an interrupt routine number). You can attach multiple interrupt events to one interrupt routine, but one event cannot be concurrently attached to multiple interrupt routines. When an event occurs with interrupts enabled, only the last interrupt routine attached to this event is executed.

When you attach an interrupt event to an interrupt routine, that interrupt is automatically enabled. If you disable all interrupts using the global disable interrupt instruction, each occurrence of the interrupt event is queued until interrupts are re-enabled, using the global enable interrupt instruction.

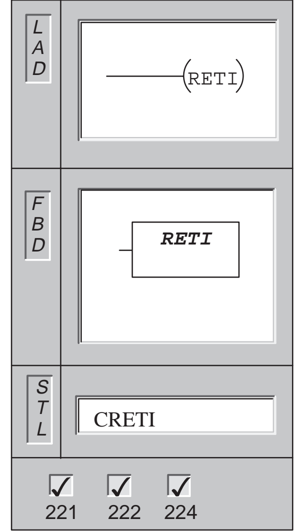
You can disable individual interrupt events by breaking the association between the interrupt event and the interrupt routine with the Detach Interrupt instruction (DTCH). The Detach instruction returns the interrupt to an inactive or ignored state.

Table 9-20 lists the different types of interrupt events.

Table 9-20 Interrupt Events

Event Number	Interrupt Description	CPU 221	CPU 222	CPU 224
0	Rising edge, I0.0	Y	Y	Y
1	Falling edge, I0.0	Y	Y	Y
2	Rising edge, I0.1	Y	Y	Y
3	Falling edge, I0.1	Y	Y	Y
4	Rising edge, I0.2	Y	Y	Y
5	Falling edge, I0.2	Y	Y	Y
6	Rising edge, I0.3	Y	Y	Y
7	Falling edge, I0.3	Y	Y	Y
8	Port 0: Receive character	Y	Y	Y
9	Port 0: Transmit complete	Y	Y	Y
10	Timed interrupt 0, SMB34	Y	Y	Y
11	Timed interrupt 1, SMB35	Y	Y	Y
12	HSC0 CV=PV (current value = preset value)	Y	Y	Y
13	HSC1 CV=PV (current value = preset value)			Y
14	HSC1 direction changed			Y
15	HSC1 external reset			Y
16	HSC2 CV=PV (current value = preset value)			Y
17	HSC2 direction changed			Y
18	HSC2 external reset			Y
19	PLS0 pulse count complete interrupt	Y	Y	Y
20	PLS1 pulse count complete interrupt	Y	Y	Y
21	Timer T32 CT=PT interrupt	Y	Y	Y
22	Timer T96 CT=PT interrupt	Y	Y	Y
23	Port 0: Receive message complete	Y	Y	Y
24	Port 1: Receive message complete			
25	Port 1: Receive character			
26	Port 1: Transmit complete			
27	HSC0 direction changed	Y	Y	Y
28	HSC0 external reset	Y	Y	Y
29	HSC4 CV=PV (current value = preset value)	Y	Y	Y
30	HSC4 direction changed	Y	Y	Y
31	HSC4 external reset	Y	Y	Y
32	HSC3 CV=PV (current value = preset value)	Y	Y	Y
33	HSC5 CV=PV (current value = preset value)	Y	Y	Y

Return from Interrupt



The **Conditional Return from Interrupt** instruction may be used to return from an interrupt, based upon the condition of the preceding logic. To add an interrupt, select **Edit** **Insert ▸ Interrupt** from the menu.

Operands: None

Data Types: None

The Return from Interrupt routines are identified by separate program tabs in the STEP 7-Micro/WIN 32 screen.

Interrupt Routines

The interrupt routine is executed in response to an associated internal or external event. Once the last instruction of the interrupt routine has been executed, control is returned to the main program. You can exit the routine by executing a conditional return from interrupt instruction (CRETI).

Interrupt Use Guidelines

Interrupt processing provides quick reaction to special internal or external events. You should optimize interrupt routines to perform a specific task, and then return control to the main routine. By keeping the interrupt routines short and to the point, execution is quick and other processes are not deferred for long periods of time. If this is not done, unexpected conditions can cause abnormal operation of equipment controlled by the main program. For interrupts, the axiom, “the shorter, the better,” is definitely true.

Restrictions

You cannot use the DISI, ENI, HDEF, LSCR, and END instructions in an interrupt routine.

System Support for Interrupt

Because contact, coil, and accumulator logic may be affected by interrupts, the system saves and reloads the logic stack, accumulator registers, and the special memory bits (SM) that indicate the status of accumulator and instruction operations. This avoids disruption to the main user program caused by branching to and from an interrupt routine.

Calling Subroutine From Interrupt Routines

You can call one nesting level of subroutines from an interrupt routine. The accumulators and the logic stack are shared between an interrupt routine and a subroutine that is called.

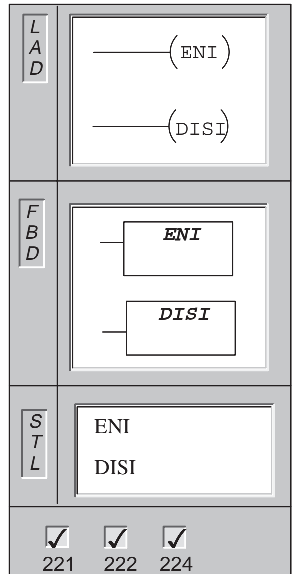
Sharing Data Between the Main Program and Interrupt Routines

You can share data between the main program and one or more interrupt routines. For example, a part of your main program may provide data to be used by an interrupt routine, or vice versa. If your program is sharing data, you must also consider the effect of the asynchronous nature of interrupt events, which can occur at any point during the execution of your main program. Problems with the consistency of shared data can result due to the actions of interrupt routines when the execution of instructions in your main program is interrupted by interrupt events.

There are a number of programming techniques you can use to ensure that data is correctly shared between your main program and interrupt routines. These techniques either restrict the way access is made to shared memory locations, or prevent interruption of instruction sequences using shared memory locations.

- For an STL program that is sharing a single variable: If the shared data is a single byte, word, or double-word variable and your program is written in STL, then correct shared access can be ensured by storing the intermediate values from operations on shared data only in non-shared memory locations or accumulators.
- For a LAD program that is sharing a single variable: If the shared data is a single byte, word, or double-word variable and your program is written in LAD, then correct shared access can be ensured by establishing the convention that access to shared memory locations be made using only Move instructions (MOVB, MOVW, MOVD, MOVR). While many LAD instructions are composed of interruptible sequences of STL instructions, these Move instructions are composed of a single STL instruction whose execution cannot be affected by interrupt events.
- For an STL or LAD program that is sharing multiple variables: If the shared data is composed of a number of related bytes, words, or double-words, then the interrupt disable/enable instructions (DISI and ENI) can be used to control interrupt routine execution. At the point in your main program where operations on shared memory locations are to begin, disable the interrupts. Once all actions affecting the shared locations are complete, re-enable the interrupts. During the time that interrupts are disabled, interrupt routines cannot be executed and therefore cannot access shared memory locations; however, this approach can result in delayed response to interrupt events.

Enable Interrupt, Disable Interrupt



The **Enable Interrupt** instruction globally enables processing of all attached interrupt events.

The **Disable Interrupt** instruction globally disables processing of all interrupt events.

Operands: None

Data Types: None

When you make the transition to the RUN mode, interrupts are initially disabled. Once in RUN mode, you can enable all interrupts by executing the global Enable Interrupt instruction. The global Disable Interrupt instruction allows interrupts to be queued, but does not allow the interrupt routines to be invoked.

Communication Port Interrupts

The serial communications port of the programmable logic controller can be controlled by the LAD or STL program. This mode of operating the communications port is called Freeport mode. In Freeport mode, your program defines the baud rate, bits per character, parity, and protocol. The receive and transmit interrupts are available to facilitate your program-controlled communications. Refer to the transmit/receive instructions for more information.

I/O Interrupts

I/O interrupts include rising/falling edge interrupts, high-speed counter interrupts, and pulse train output interrupts. The CPU can generate an interrupt on rising and/or falling edges of an input. See Table 9-21 for the inputs available for the interrupts. The rising edge and the falling edge events can be captured for each of these input points. These rising/falling edge events can be used to signify a condition that must receive immediate attention when the event happens.

Table 9-21 Rising/Falling Edge Interrupts Supported

I/O Interrupts	S7-200 CPU
I/O Points	I0.0 to I0.3

The high-speed counter interrupts allow you to respond to conditions such as the current value reaching the preset value, a change in counting direction that might correspond to a reversal in the direction in which a shaft is turning, or an external reset of the counter. Each of these high-speed counter events allows action to be taken in real time in response to high-speed events that cannot be controlled at programmable logic controller scan speeds.

The pulse train output interrupts provide immediate notification of completion of outputting the prescribed number of pulses. A typical use of pulse train outputs is stepper motor control.

You can enable each of the above interrupts by attaching an interrupt routine to the related I/O event.

Time-Based Interrupts

Time-based interrupts include timed interrupts and the Timer T32/T96 interrupts. The CPU can support timed interrupts. You can specify actions to be taken on a cyclic basis using a timed interrupt. The cycle time is set in 1-ms increments from 1 ms to 255 ms. You must write the cycle time in SMB34 for timed interrupt 0, and in SMB35 for timed interrupt 1.

The timed interrupt event transfers control to the appropriate interrupt routine each time the timer expires. Typically, you use timed interrupts to control the sampling of analog inputs at regular intervals or to execute a PID loop at a timed interrupt.

A timed interrupt is enabled and timing begins when you attach an interrupt routine to a timed interrupt event. During the attachment, the system captures the cycle time value, so subsequent changes do not affect the cycle time. To change the cycle time, you must modify the cycle time value, and then re-attach the interrupt routine to the timed interrupt event. When the re-attachment occurs, the timed interrupt function clears any accumulated time from the previous attachment, and begins timing with the new value.

Once enabled, the timed interrupt runs continuously, executing the attached interrupt routine on each expiration of the specified time interval. If you exit the RUN mode or detach the timed interrupt, the timed interrupt is disabled. If the global disable interrupt instruction is executed, timed interrupts continue to occur. Each occurrence of the timed interrupt is queued (until either interrupts are enabled, or the queue is full). See Figure 9-66 for an example of using a timed interrupt.

The timer T32/T96 interrupts allow timely response to the completion of a specified time interval. These interrupts are only supported for the 1-ms resolution on-delay (TON) and off-delay (TOF) timers T32 and T96. The T32 and T96 timers otherwise behave normally. Once the interrupt is enabled, the attached interrupt routine is executed when the active timer's current value becomes equal to the preset time value during the normal 1-ms timer update performed in the CPU. You enable these interrupts by attaching an interrupt routine to the T32/T96 interrupt events.

Understanding the Interrupt Priority and Queuing

Interrupts are prioritized according to the fixed priority scheme shown below:

- Communication (highest priority)
- I/O interrupts
- Time-based interrupts (lowest priority)

Interrupts are serviced by the CPU on a first-come-first-served basis within their respective priority assignments. Only one user-interrupt routine is ever being executed at any point in time. Once the execution of an interrupt routine begins, the routine is executed to completion. It cannot be pre-empted by another interrupt routine, even by a higher priority routine. Interrupts that occur while another interrupt is being processed are queued for later processing.

The three interrupt queues and the maximum number of interrupts they can store are shown in Table 9-22.

Table 9-22 Interrupt Queues and Maximum Number of Entries per Queue

Queue	CPU 221	CPU 222	CPU 224
Communications queue	4	4	4
I/O Interrupt queue	16	16	16
Timed Interrupt queue	8	8	8

Potentially, more interrupts can occur than the queue can hold. Therefore, queue overflow memory bits (identifying the type of interrupt events that have been lost) are maintained by the system. The interrupt queue overflow bits are shown in Table 9-23. You should use these bits only in an interrupt routine because they are reset when the queue is emptied, and control is returned to the main program.

Table 9-23 Special Memory Bit Definitions for Interrupt Queue Overflow Bits

Description (0 = no overflow, 1 = overflow)	SM Bit
Communication interrupt queue overflow	SM4.0
I/O interrupt queue overflow	SM4.1
Timed interrupt queue overflow	SM4.2

Table 9-24 shows the interrupt event, priority, and assigned event number.

Table 9-24 Interrupt Events in Priority Order

Event Number	Interrupt Description	Priority Group	Priority in Group
8	Port 0: Receive character	Communications (highest)	0
9	Port 0: Transmit complete		0
23	Port 0: Receive message complete		0
24	Port 1: Receive message complete		1
25	Port 1: Receive character		1
26	Port 1: Transmit complete		1
19	PTO 0 complete interrupt	Discrete (middle)	0
20	PTO 1 complete interrupt		1
0	Rising edge, I0.0		2
2	Rising edge, I0.1		3
4	Rising edge, I0.2		4
6	Rising edge, I0.3		5
1	Falling edge, I0.0		6
3	Falling edge, I0.1		7
5	Falling edge, I0.2		8
7	Falling edge, I0.3		9
12	HSC0 CV=PV (current value = preset value)		10
27	HSC0 direction changed		11
28	HSC0 external reset		12
13	HSC1 CV=PV (current value = preset value)		13
14	HSC1 direction input changed		14
15	HSC1 external reset		15
16	HSC2 CV=PV		16
17	HSC2 direction changed		17
18	HSC2 external reset		18
32	HSC3 CV=PV (current value = preset value)		19
29	HSC4 CV=PV (current value = preset value)		20
30	HSC4 direction changed		21
31	HSC4 external reset		22
33	HSC5 CV=PV (current value = preset value)		23
10	Timed interrupt 0	Timed (lowest)	0
11	Timed interrupt 1		1
21	Timer T32 CT=PT interrupt		2
22	Timer T96 CT=PT interrupt		3

Interrupt Examples

Figure 9-65 shows an example of the Interrupt Routine instructions.

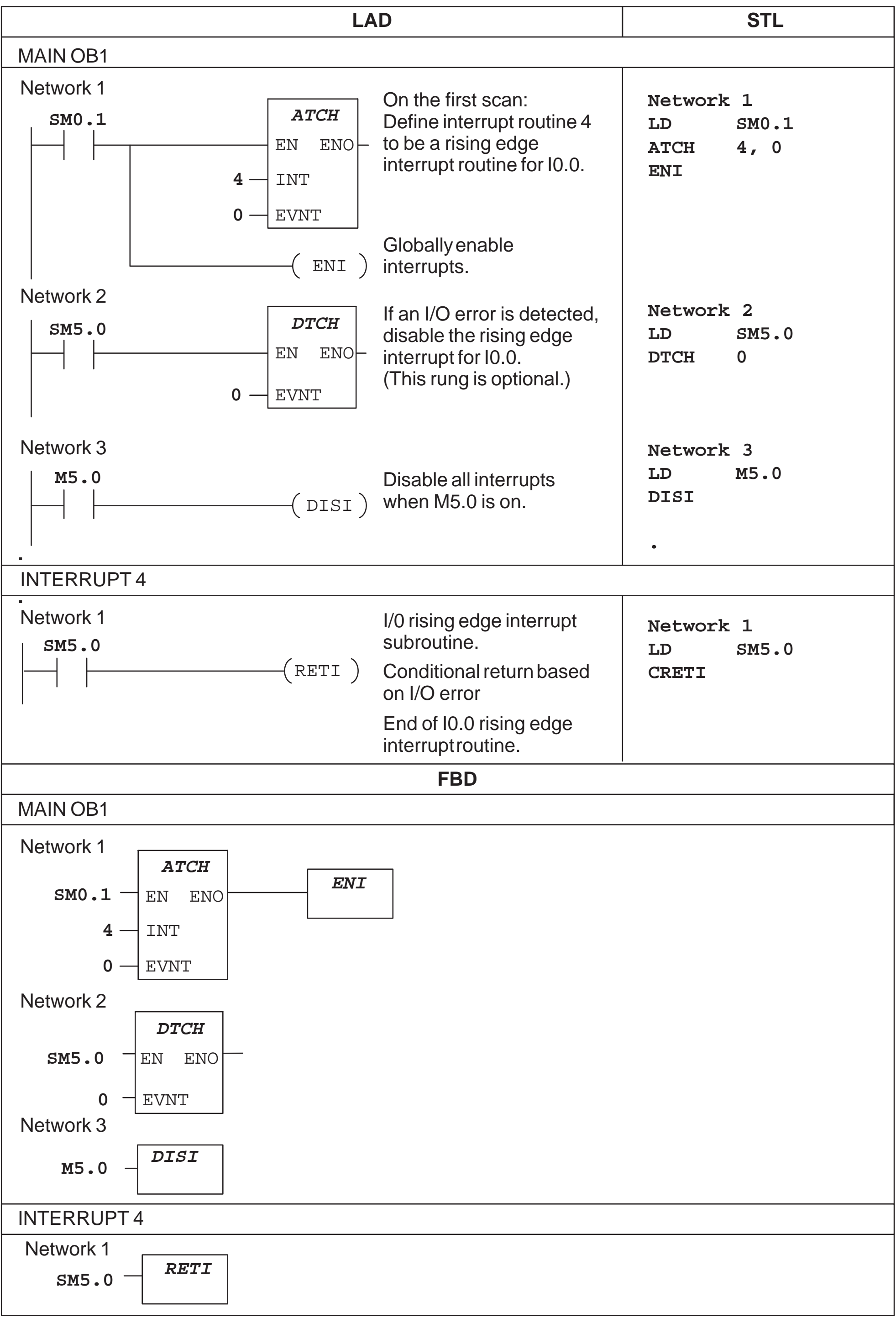


Figure 9-65 Example of Interrupt Instructions

Figure 9-66 shows how to set up a timed interrupt to read the value of an analog input.

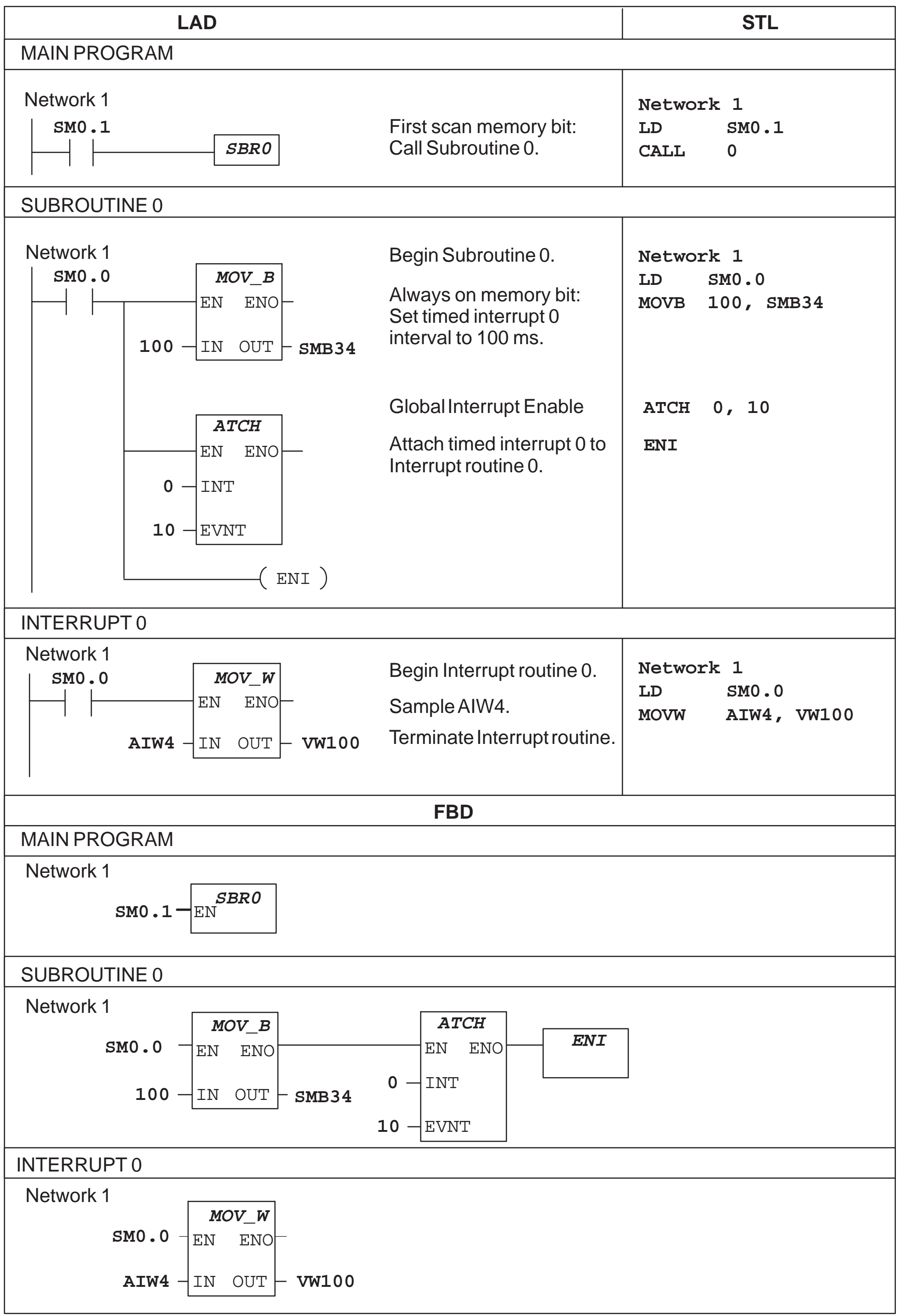
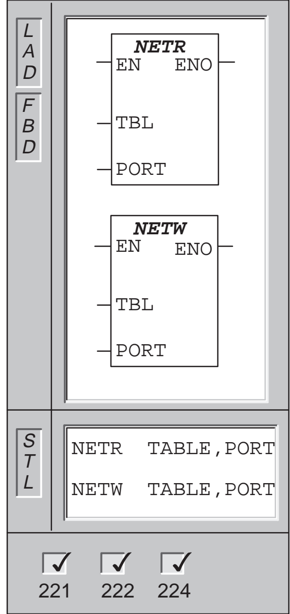


Figure 9-66 Example of How to Set Up a Timed Interrupt to Read the Value of an Analog Input

Network Read, Network Write



The **Network Read** instruction initiates a communication operation to gather data from a remote device through the specified port (PORT), as defined by the table (TBL).

The **Network Write** instruction initiates a communication operation to write data to a remote device through the specified port (PORT), as defined by the table (TBL).

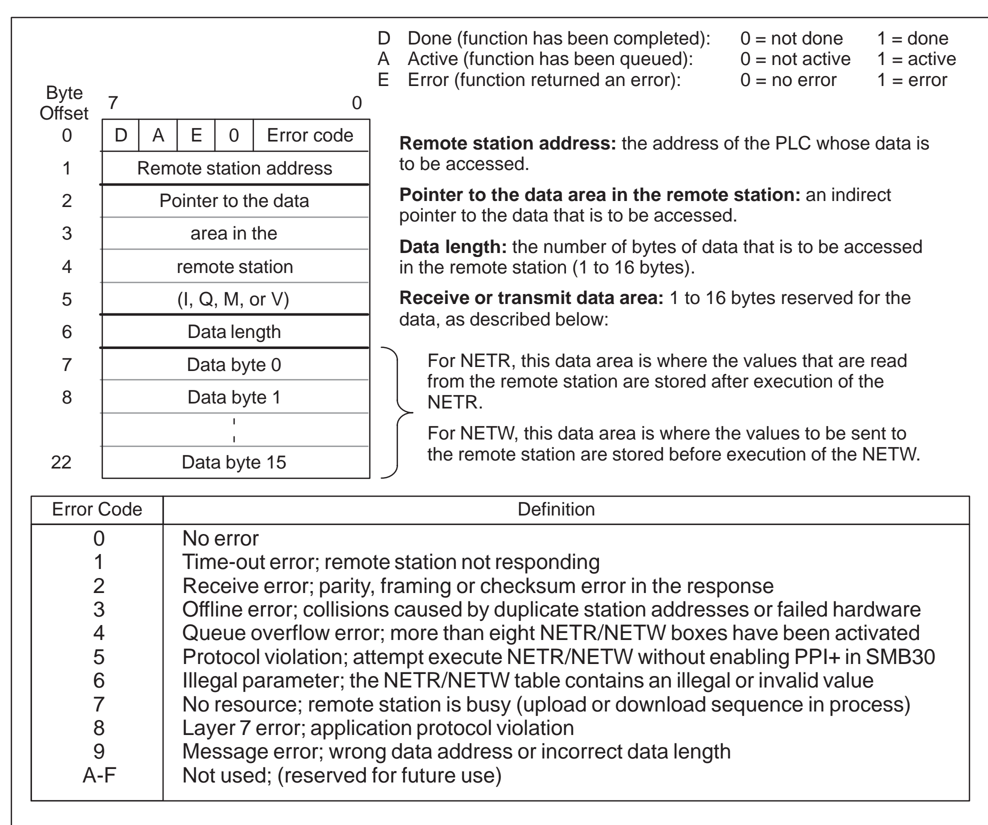
The NETR instruction can read up to 16 bytes of information from a remote station, and the NETW instruction can write up to 16 bytes of information to a remote station. You may have any number of NETR/NETW instructions in the program, but only a maximum of eight NETR and NETW instructions may be activated at any one time. For example, you can have four NETRs and four NETWs, or two NETRs and six NETWs active at the same time in a given S7-200.

Figure 9-67 defines the table that is referenced by the TBL parameter in the NETR and NETW instructions.

NETR: Error conditions that set ENO = 0:
SM4.3 (run-time), 0006 (indirect address)

NETW: Error conditions that set ENO = 0:
SM4.3 (run-time), 0006 (indirect address)

Inputs/Outputs	Operands	Data Types
TBL	I, Q, M, S, V, VB, MB, *VD, *AC, *LD	BYTE
PORT	Constant	BYTE



Error Code	Definition
0	No error
1	Time-out error; remote station not responding
2	Receive error; parity, framing or checksum error in the response
3	Offline error; collisions caused by duplicate station addresses or failed hardware
4	Queue overflow error; more than eight NETR/NETW boxes have been activated
5	Protocol violation; attempt execute NETR/NETW without enabling PPI+ in SMB30
6	Illegal parameter; the NETR/NETW table contains an illegal or invalid value
7	No resource; remote station is busy (upload or download sequence in process)
8	Layer 7 error; application protocol violation
9	Message error; wrong data address or incorrect data length
A-F	Not used; (reserved for future use)

Figure 9-67 Definition of TABLE for NETR and NETW

Example of Network Read and Network Write

Figure 9-68 shows an example to illustrate the utility of the NETR and NETW instructions. For this example, consider a production line where tubs of butter are being filled and sent to one of four boxing machines (case packers). The case packer packs eight tubs of butter into a single cardboard box. A diverter machine controls the flow of butter tubs to each of the case packers. Four CPU 221 modules are used to control the case packers and a CPU 222 module equipped with a TD 200 operator interface is used to control the diverter. Figure 9-68 shows the network setup.

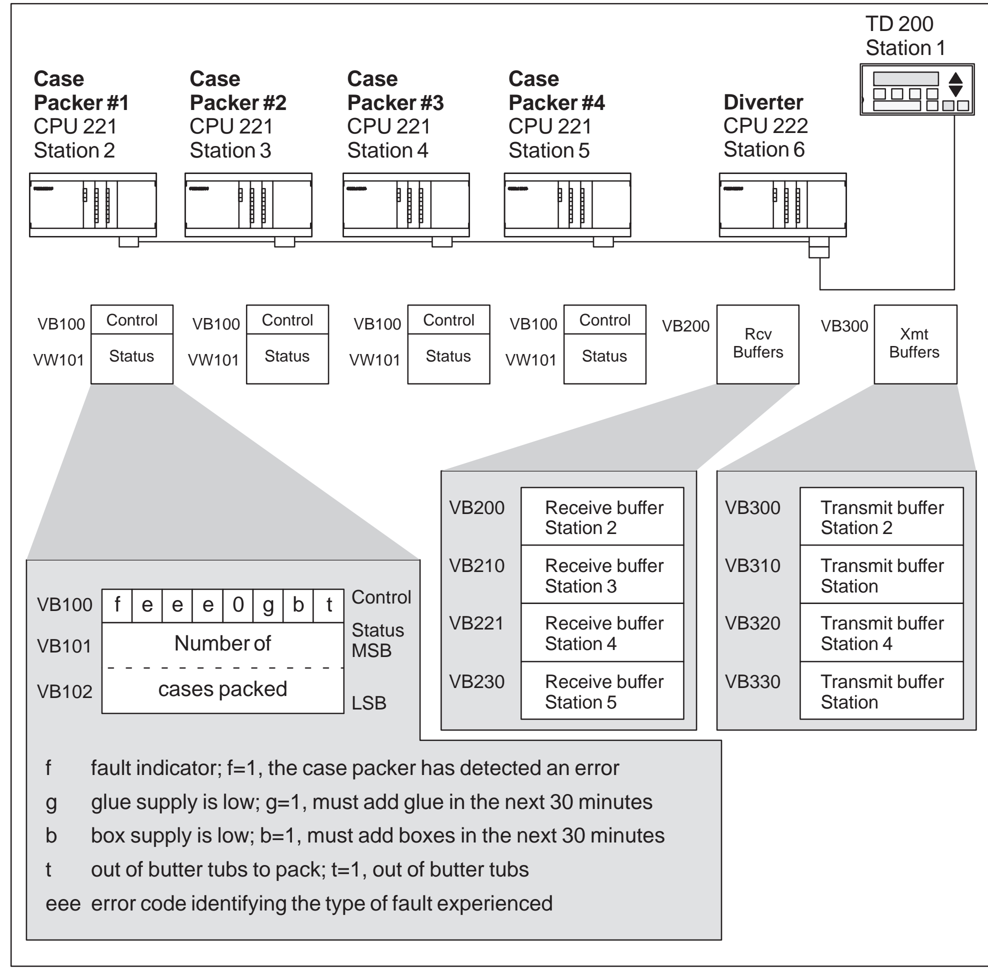


Figure 9-68 Example of NETR and NETW Instructions

The receive and transmit buffers for accessing the data in station 2 (located at VB200 and VB300, respectively) are shown in detail in Figure 9-69.

The CPU 224 uses a NETR instruction to read the control and status information on a continuous basis from each of the case packers. Each time a case packer has packed 100 cases, the diverter notes this and sends a message to clear the status word using a NETW instruction.

The program required to read the control byte, the number of cases packed and to reset the number of cases packed for a single case packer (case packer #1) is shown in Figure 9-70.

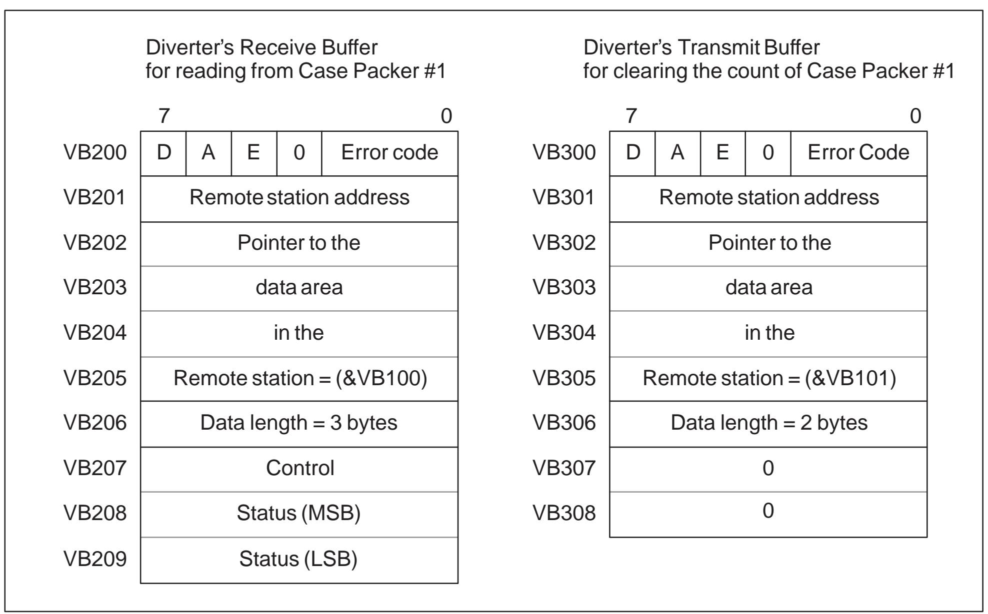


Figure 9-69 Sample TABLE Data for NETR and NETW Example

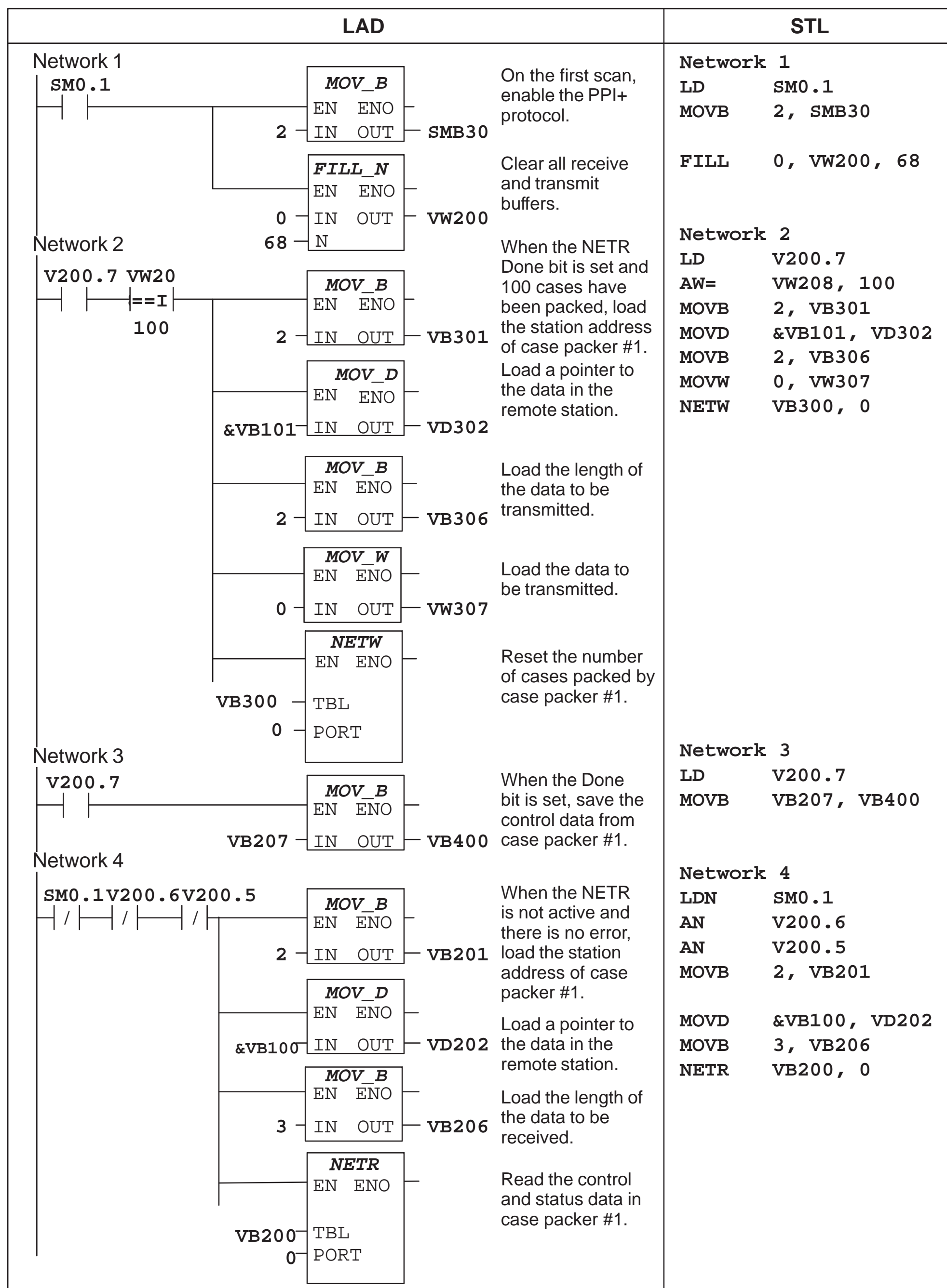


Figure 9-70 Example of NETR and NETW Instructions for LAD and STL

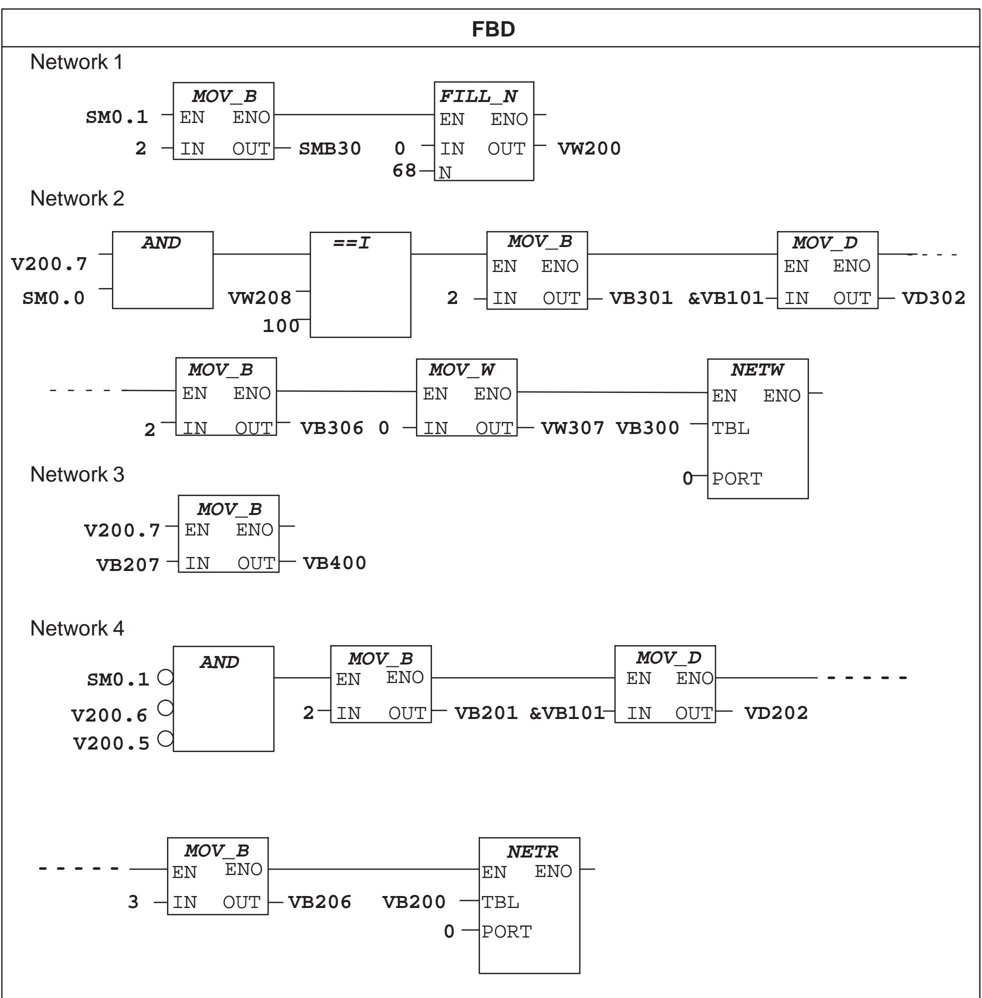
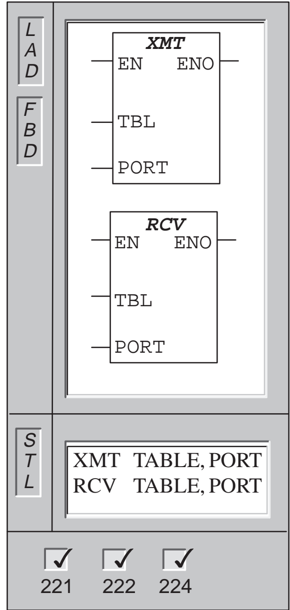


Figure 9-71   Example of NETR and NETW instructions for FBD

Transmit, Receive



The **Transmit** instruction invokes the transmission of the data buffer (TBL). The first entry in the data buffer specifies the number of bytes to be transmitted. PORT specifies the communication port to be used for transmission.

The XMT instruction is used in Freeport mode to transmit data by means of the communication port(s).

The format of the XMT buffer is:

The **Receive** instruction initiates or terminates the Receive Message service. You must specify a start and an end condition for the Receive box to operate. Messages received through the specified port (PORT) are stored in the data buffer (TBL). The first entry in the data buffer specifies the number of bytes received.

Transmit: Error conditions that set ENO = 0: SM4.3 (run-time), 0006 (indirect address), 0009 (simultaneous XMT/RCV on port 0), 000B (simultaneous XMT/RCV on port 1)

Receive: Error conditions that set ENO = 0: SM86.6 and SM186.6 (RCV parameter error), SM4.3 (run-time), 0006 (indirect address), 0009 (simultaneous XMT/RCV on port 0), 000B (simultaneous XMT/RCV on port 1)

Inputs/Outputs	Operands	Data Types
TABLE	VB, IB, QB, MB, SB, SMB, *VD, *AC, *LD	BYTE
PORT	Constant (0)	BYTE

Understanding Freeport Mode

You can select the Freeport mode to control the serial communication port of the CPU by means of the user program. When you select Freeport mode, the LAD program controls the operation of the communication port through the use of the receive interrupts, the transmit interrupts, the transmit instruction (XMT), and the receive instruction (RCV). The communication protocol is entirely controlled by the ladder program while in Freeport mode. SMB30 (for port 0) and SMB130 (for port 1 if your CPU has two ports) are used to select the baud rate and parity.

The Freeport mode is disabled and normal communication is re-established (for example, programming device access) when the CPU is in the STOP mode.

In the simplest case, you can send a message to a printer or a display using only the Transmit (XMT) instruction. Other examples include a connection to a bar code reader, a weighing scale, and a welder. In each case, you must write your program to support the protocol that is used by the device with which the CPU communicates while in Freeport mode.

Freeport communication is possible only when the CPU is in the RUN mode. Enable the Freeport mode by setting a value of 01 in the protocol select field of SMB30 (Port 0) or SMB130 (Port 1). While in Freeport mode, communication with the programming device is not possible.

Note

Entering Freeport mode can be controlled using special memory bit SM0.7, which reflects the current position of the operating mode switch. When SM0.7 is equal to 0, the switch is in TERM position; when SM0.7 = 1, the operating mode switch is in RUN position. If you enable Freeport mode only when the switch is in RUN position, you can use the programming device to monitor or control the CPU operation by changing the switch to any other position.

Freeport Initialization

SMB30 and SMB130 configure the communication ports, 0 and 1, respectively, for Freeport operation and provide selection of baud rate, parity, and number of data bits. The Freeport control byte(s) description is shown in Table 9-25.

Table 9-25 Special Memory Bytes SMB30 and SMB130

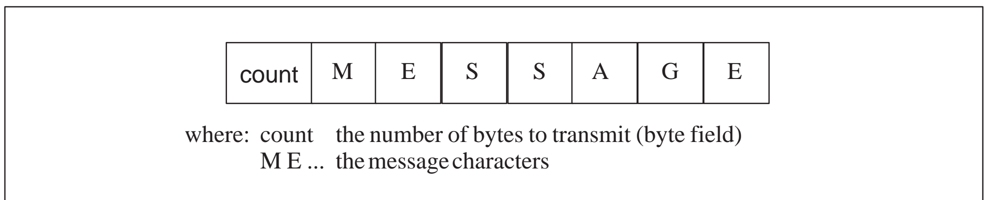
Port 0	Port 1	Description
Format of SMB30	Format of SMB130	MSB 7 … LSB 0: p p d b b b m m — Freeport mode control byte
SM30.6 and SM30.7	SM130.6 and SM130.7	pp Parity select 00 = no parity 01 = even parity 10 = no parity 11 = odd parity
SM30.5	SM130.5	d Data bits per character 0 = 8 bits per character 1 = 7 bits per character
SM30.2 to SM30.4	SM130.2 to SM130.4	bbb Freeport Baud rate 000 = 38,400 baud 001 = 19,200 baud 010 = 9,600 baud 011 = 4,800 baud 100 = 2,400 baud 101 = 1,200 baud 110 = 600 baud 111 = 300 baud
SM30.0 and SM30.1	SM130.0 and SM130.1	mm Protocol selection 00 = Point-to-Point Interface protocol (PPI/slave mode) 01 = Freeport protocol 10 = PPI/master mode 11 = Reserved (defaults to PPI/slave mode)
Note: One stop bit is generated for all configurations.		

Using the XMT Instruction to Transmit Data

The XMT instruction lets you send a buffer of one or more characters, up to a maximum of 255. An interrupt is generated (interrupt event 9 for port 0 and interrupt event 26 for port 1) after the last character of the buffer is sent, if an interrupt routine is attached to the transmit complete event. You can make transmissions without using interrupts (for example, sending a message to a printer) by monitoring SM4.5 or SM4.6 to signal when transmission is complete.

The XMT instruction can be used to generate a BREAK condition by setting the number of characters to zero and then executing the XMT instruction. This generates a BREAK condition on the line for 16-bit times at the current baud rate. Transmitting a BREAK is handled in the same manner as transmitting any other message, in that a XMT interrupt is generated when the BREAK is complete and SM4.5 or SM4.6 signal the current status of the XMT.

The format of the XMT buffer is shown in Figure 9-72.



where: count the number of bytes to transmit (byte field)
M E ... the message characters

Figure 9-72 XMT Buffer Format

Using the RCV Instruction to Receive Data

The RCV instruction lets you receive a buffer of one or more characters, up to a maximum of 255. An interrupt is generated (interrupt event 23 for port 0 and interrupt event 24 for port 1) after the last character of the buffer is received, if an interrupt routine is attached to the receive message complete event.

You can receive messages without using interrupts by monitoring SMB86. SMB86 (or SMB186) will be non-zero when the RCV box is inactive or has been terminated. It will be zero when a receive is in progress.

The RCV instruction allows you to select the message start and message end conditions. See Table 9-26 (SM86 through SM94 for port 0, and SM186 through SM194 for port 1) for descriptions of the start and end message conditions.The format of the RCV buffer is shown in Figure 9-73.

Note

The Receive Message function is automatically terminated by an overrun or a parity error. You must define a start condition (x or z), and an end condition (y, t, or maximum character count) for the Receive Message function to operate.

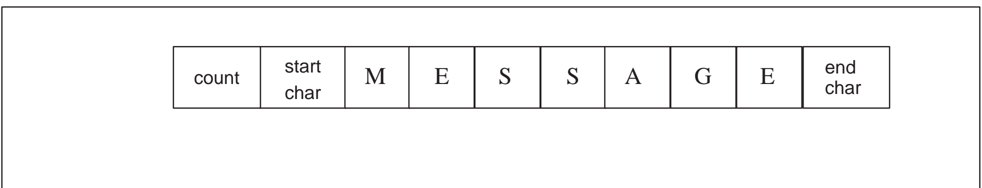


Figure 9-73 RCV Buffer Format

Table 9-26 Special Memory Bytes SMB86 to SMB94, and SMB186 to SMB194

Port 0	Port 1	Description
SMB86	SMB186	MSB 7 … LSB 0: n | r | e | 0 | 0 | t | c | p — Receive message status byte n: 1 = Receive message terminated by user disable command r: 1 = Receive message terminated: error in input parameters or missing start or end condition e: 1 = End character received t: 1 = Receive message terminated: timer expired c: 1 = Receive message terminated: maximum character count achieved p 1 = Receive message terminated because of a parity error

Table 9-26 Special Memory Bytes SMB86 to SMB94, and SMB186 to SMB194

Port 0	Port 1	Description
SMB87	SMB187	MSB 7 ... LSB 0 n | x | y | z | m | t | 0 | 0 — Receive message control byte n: 0 = Receive Message function is disabled. 1 = Receive Message function is enabled . The enable/disable receive message bit is checked each time the RCV instruction is executed. x: 0 = Ignore SMB88 or SMB188. 1 = Use the value of SMB88 or SMB188 to detect start of message. y; 0 = Ignore SMB89 or SMB189. 1 = Use the value of SMB89 or SMB189 to detect end of message. z: 0 = Ignore SMW90 or SMB190. 1 = Use the value of SMW90 to detect an idle line condition. m: 0 = Timer is an inter-character timer. 1 = Timer is a message timer. t: 0 = Ignore SMW92 or SMW192. 1 = Terminate receive if the time period in SMW92 or SMW192 is exceeded. The bits of the message interrupt control byte are used to define the criteria by which the message is identified. Both start of message and end of message criteria are defined. To determine the start of a message, either of two sets of logically ANDed start of message criteria must be true and must occur in sequence (idle line followed by start character, or break followed by start character). To determine the end of a message, the enabled end of the message criteria is logically ORed. The equations for start and stop criteria are given below: Start of Message = il * sc + bk * sc End of Message = ec + tmr + maximum character count reached Programming the start of message criteria for: 1. Idle line detection: il=1, sc=0, bk=0, SMW90>0 2. Start character detection: il=0, sc=1, bk=0, SMW90 is a don't care 3. Break Detection: il=0, sc=0, bk=1, SMW90 is a don't care 4. Any response to a request: il=1, sc=0, bk=0, SMW90=0 (Message timer can be used to terminate receive if there is no response.) 5. Break and a start character: il=0, sc=1, bk=1, SMW90 is a don't care 6. Idle line and a start character: il=1, sc=1, bk=0, SMW90 >0 7. Idle line and start character (Illegal): il=1, sc=1, bk=0, SMW90=0 Note: Receive will automatically be terminated by an overrun or a parity error (if enabled).
SMB88	SMB188	Start of message character
SMB89	SMB189	End of message character

Table 9-26 Special Memory Bytes SMB86 to SMB94, and SMB186 to SMB194

Port 0	Port 1	Description
SMB90 SMB91	SMB190 SMB191	Idle line time period given in milliseconds. The first character received after idle line time has expired is the start of a new message. SM90 (or SM190) is the most significant byte and SM91 (or SM191) is the least significant byte.
SMB92 SMB93	SMB192 SMB193	Inter-character/message timer time-out value given in milliseconds. If the time period is exceeded, the receive message is terminated. SM92 (or SM192) is the most significant byte, and SM93 (or SM193) is the least significant byte.
SMB94	SMB194	Maximum number of characters to be received (1 to 255 bytes). Note: This range must be set to the expected maximum buffer size, even if the character count message termination is not used.

Using Character Interrupt Control to Receive Data

To allow complete flexibility in protocol support, you can also receive data using character interrupt control. Each character received generates an interrupt. The received character is placed in SMB2, and the parity status (if enabled) is placed in SM3.0 just prior to execution of the interrupt routine attached to the receive character event.

- SMB2 is the Freeport receive character buffer. Each character received while in Freeport mode is placed in this location for easy access from the user program.
- SMB3 is used for Freeport mode and contains a parity error bit that is turned on when a parity error is detected on a received character. All other bits of the byte are reserved. Use this bit either to discard the message or to generate a negative acknowledge to the message.

Note

SMB2 and SMB3 are shared between Port 0 and Port 1. When the reception of a character on Port 0 results in the execution of the interrupt routine attached to that event (interrupt event 8), SMB2 contains the character received on Port 0, and SMB3 contains the parity status of that character. When the reception of a character on Port 1 results in the execution of the interrupt routine attached to that event (interrupt event 25), SMB2 contains the character received on Port 1 and SMB3 contains the parity status of that character.

Receive and Transmit Example

This sample program shows the use of Receive and Transmit. This program will receive a string of characters until a line feed character is received. The message is then transmitted back to the sender.

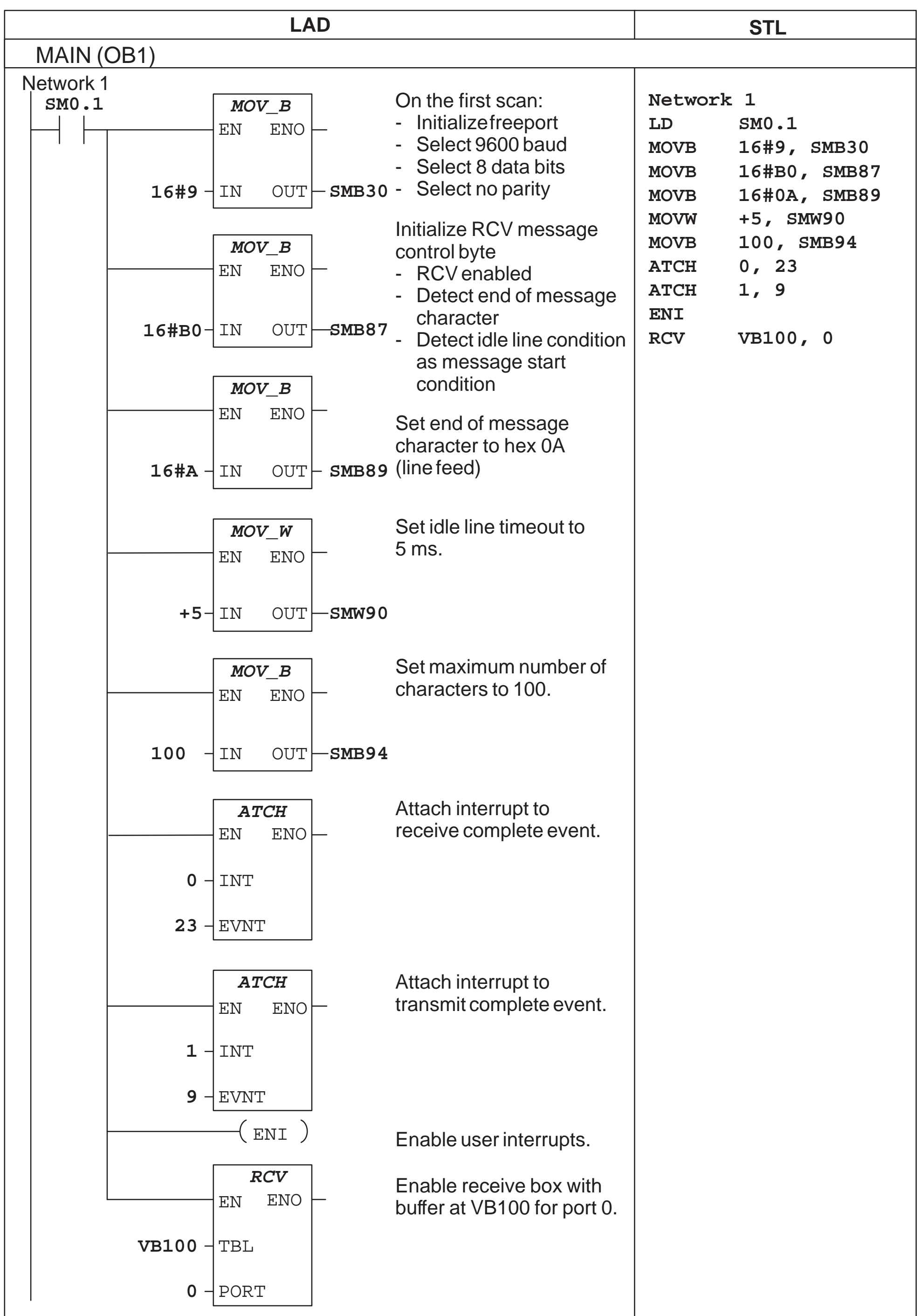


Figure 9-74 Example of Transmit Instruction

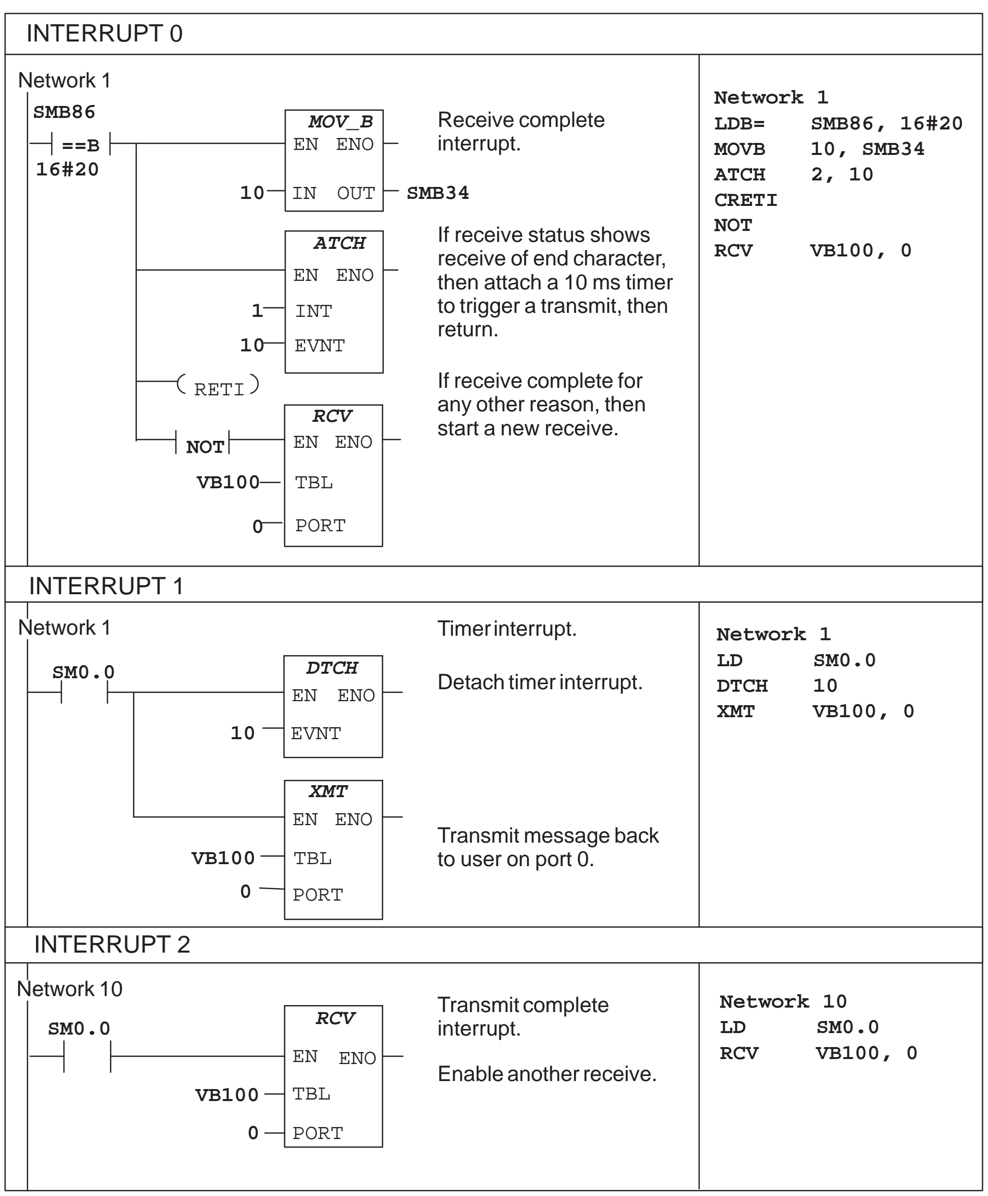


Figure 9-74 Example of Transmit Instruction (continued)

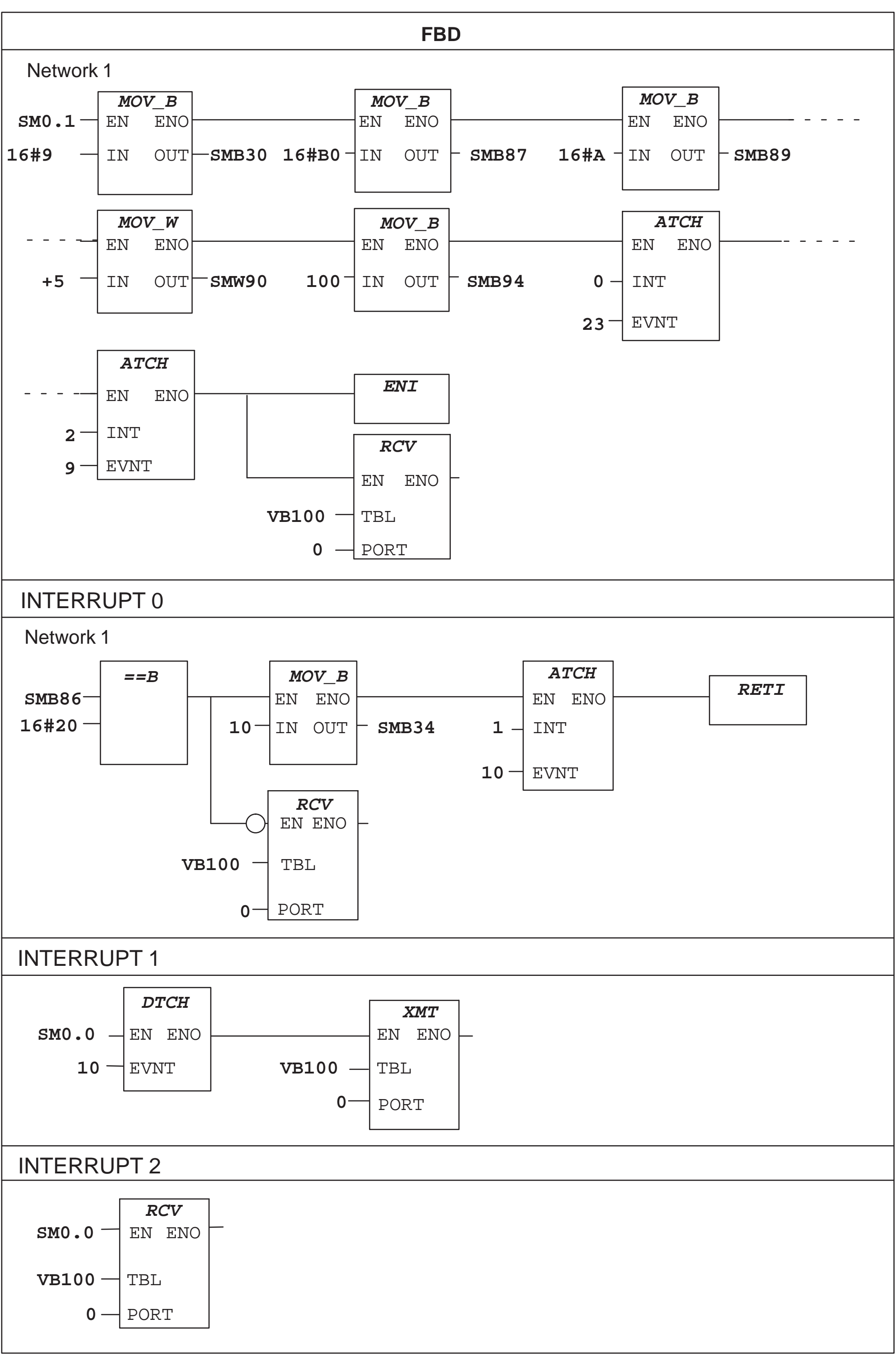
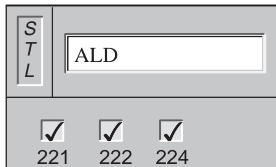


Figure 9-74 Example of Transmit Instruction (continued)

9.17 SIMATIC Logic Stack Instructions

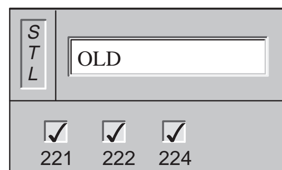
And Load



The **And Load** instruction combines the values in the first and second levels of the stack using a logical And operation. The result is loaded in the top of stack. After the ALD is executed, the stack depth is decreased by one.

Operands: none

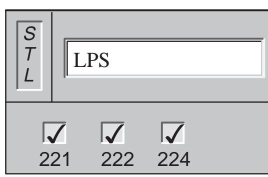
Or Load



The **Or Load** instruction combines the values in the first and second levels of the stack, using a logical Or operation. The result is loaded in the top of stack. After the OLD is executed, the stack depth is decreased by one.

Operands: none

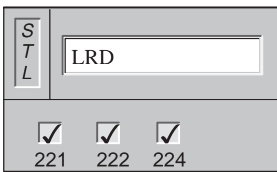
Logic Push



The **Logic Push** instruction duplicates the top value on the stack and pushes this value onto the stack. The bottom of the stack is pushed off and lost.

Operands: none

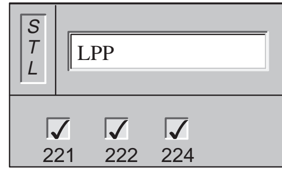
Logic Read



The **Logic Read** instruction copies the second stack value to the top of stack. The stack is not pushed or popped, but the old top of stack value is destroyed by the copy.

Operands: none

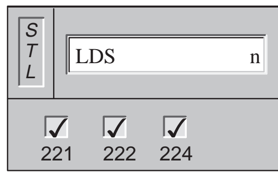
Logic Pop



The **Logic Pop** instruction pops one value off of the stack. The second stack value becomes the new top of stack value.

Operands: none

Load Stack



The **Load Stack** instruction duplicates the stack bit n on the stack and places this value on top of the stack. The bottom of the stack is pushed off and lost.

Operands: n (1 to 8)

Logic Stack Operations

Figure 9-75 illustrates the operation of the And Load and Or Load instructions.

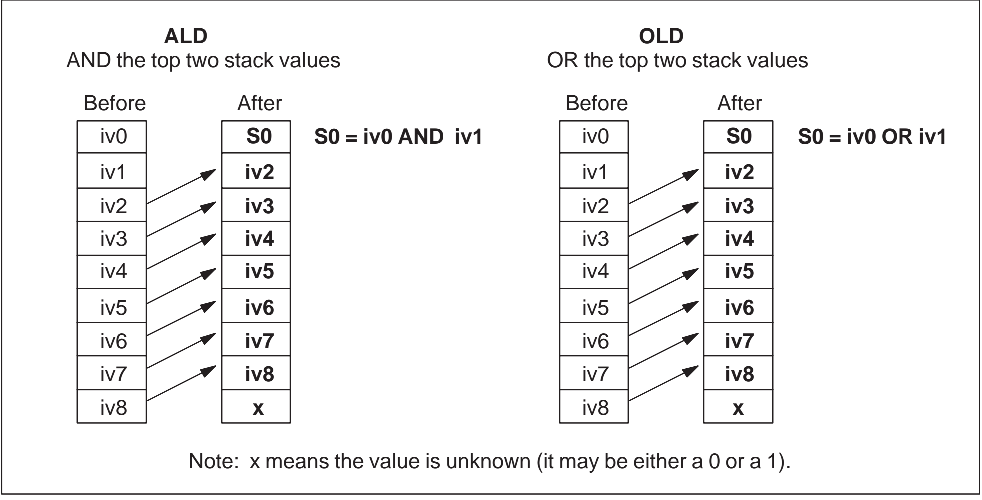


Figure 9-75 And Load and Or Load Instructions

Figure 9-76 illustrates the operation of the Logic Push, Logic Read, and Logic Pop instructions.

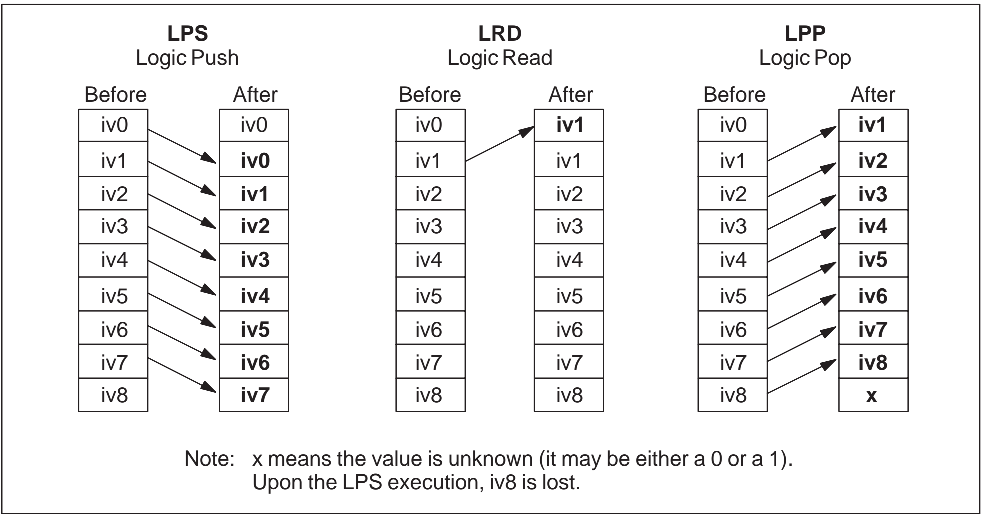


Figure 9-76 Logic Push, Logic Read, and Logic Pop Instructions

Figure 9-77 illustrates the operation of the Load Stack instructions.

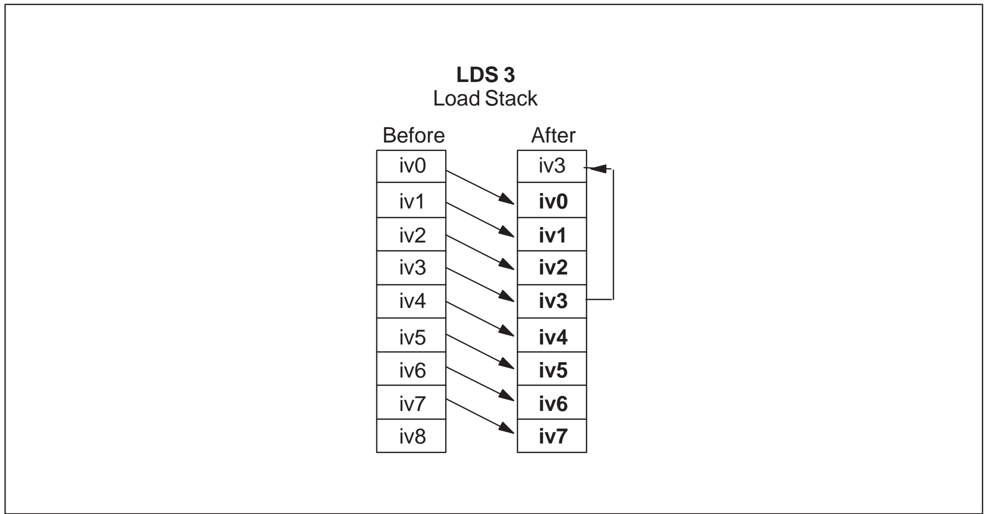


Figure 9-77 Load Stack Instructions

Logic Stack Example

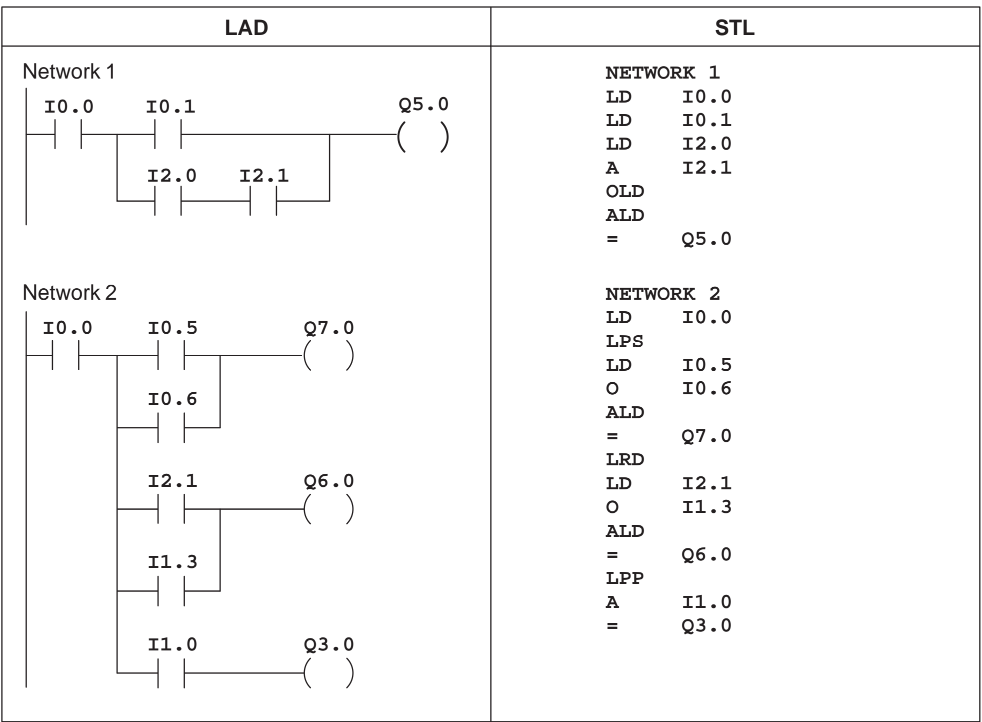


Figure 9-78 Example of Logic Stack Instructions for LAD and STL

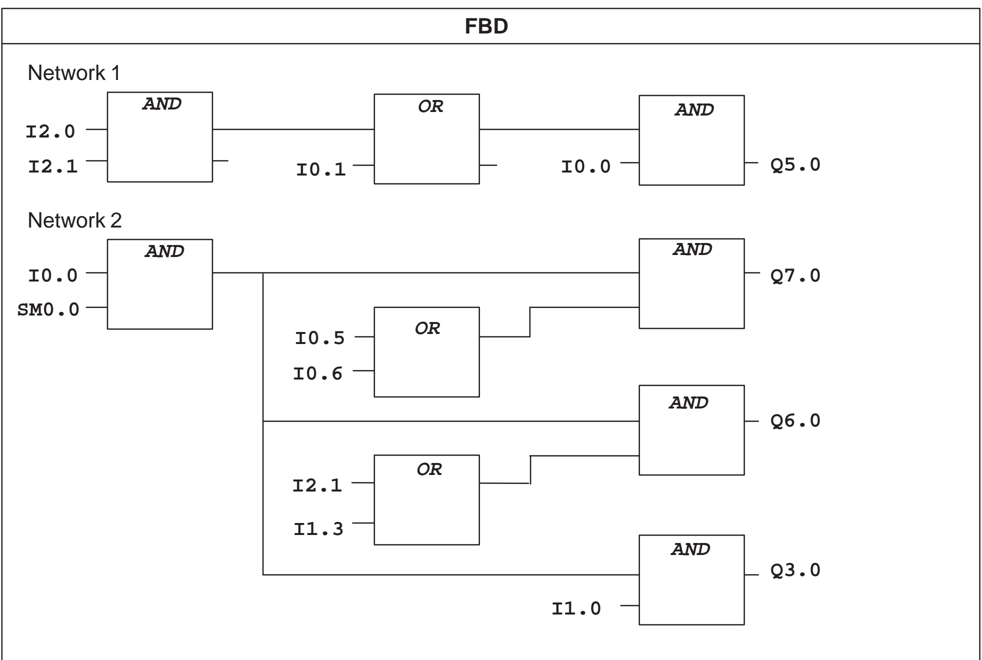


Figure 9-79 Example of Logic Stack Instructions for FBD