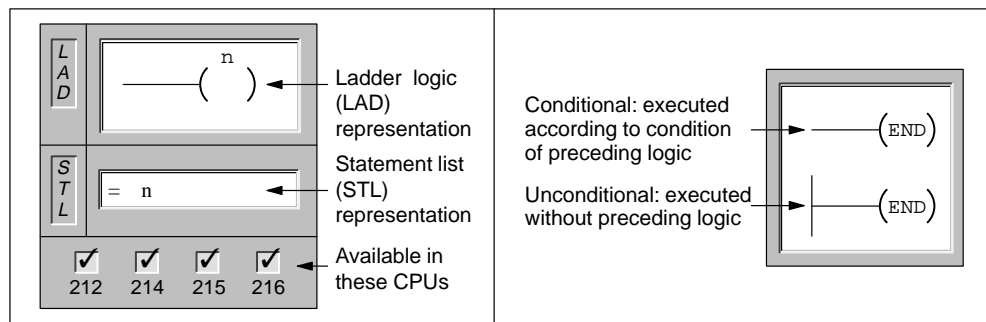


# Instruction Set

# 10

The following conventions are used in this chapter to illustrate the equivalent ladder logic and statement list instructions and the CPUs in which the instructions are available:



## Chapter Overview

Section	Description	Page
10.1	Valid Ranges for the S7-200 CPUs	10-2
10.2	Contact Instructions	10-4
10.3	Comparison Contact Instructions	10-7
10.4	Output Instructions	10-10
10.5	Timer, Counter, High-Speed Counter, High-Speed Output, Clock, and Pulse Instructions	10-13
10.6	Math and PID Loop Control Instructions	10-50
10.7	Increment and Decrement Instructions	10-66
10.8	Move, Fill, and Table Instructions	10-68
10.9	Shift and Rotate Instructions	10-78
10.10	Program Control Instructions	10-84
10.11	Logic Stack Instructions	10-99
10.12	Logic Operations	10-102
10.13	Conversion Instructions	10-108
10.14	Interrupt and Communications Instructions	10-114

## 10.1 Valid Ranges for the S7-200 CPUs

Table 10-1 Summary of S7-200 CPU Memory Ranges and Features

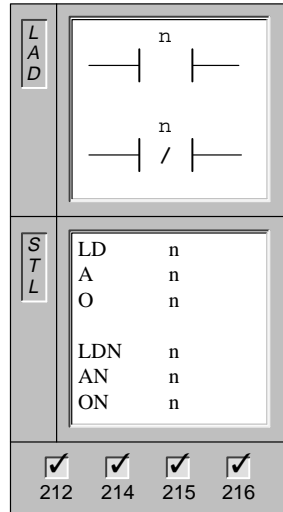
Description	CPU 212	CPU 214	CPU 215	CPU 216
User program size	512 words	2 Kwords	4 Kwords	4 Kwords
User data size	512 words	2 Kwords	2.5 Kwords	2.5 Kwords
Process-image input register	I0.0 to I7.7	I0.0 to I7.7	I0.0 to I7.7	I0.0 to I7.7
Process-image output register	Q0.0 to Q7.7	Q0.0 to Q7.7	Q0.0 to Q7.7	Q0.0 to Q7.7
Analog inputs (read only)	AIW0 to AIW30	AIW0 to AIW30	AIW0 to AIW30	AIW0 to AIW30
Analog outputs (write only)	AQW0 to AQW30	AQW0 to AQW30	AQW0 to AQW30	AQW0 to AQW30
Variable memory (V) Permanent area (max.)	V0.0 to V1023.7 V0.0 to V199.7	V0.0 to V4095.7 V0.0 to V1023.7	V0.0 to V5119.7 V0.0 to V5119.7	V0.0 to V5119.7 V0.0 to V5119.7
Bit memory (M) Permanent area (max.)	M0.0 to M15.7 MB0 to MB13	M0.0 to M31.7 MB0 to MB13	M0.0 to M31.7 MB0 to MB13	M0.0 to M31.7 MB0 to MB13
Special Memory (SM) Read only	SM0.0 to SM45.7 SM0.0 to SM29.7	SM0.0 to SM85.7 SM0.0 to SM29.7	SM0.0 to SM194.7 SM0.0 to SM29.7	SM0.0 to SM194.7 SM0.0 to SM29.7
Timers	64 (T0 to T63)	128 (T0 to T127)	256 (T0 to T255)	256 (T0 to T255)
Retentive on-delay 1 ms	T0	T0, T64	T0, T64	T0, T64
Retentive on-delay 10 ms	T1 to T4	T1 to T4, T65 to T68	T1 to T4, T65 to T68	T1 to T4, T65 to T68
Retentive on-delay 100 ms	T5 to T31	T5 to T31, T69 to T95	T5 to T31, T69 to T95	T5 to T31, T69 to T95
On-delay 1 ms	T32	T32, T96	T32, T96	T32, T96
On-delay 10 ms	T33 to T36	T33 to T36, T97 to T100	T33 to T36, T97 to T100	T33 to T36, T97 to T100
On-delay 100 ms	T37 to T63	T37 to T63, T101 to T127	T37 to T63, T101 to T255	T37 to T63, T101 to T255
Counters	C0 to C63	C0 to C127	C0 to C255	C0 to C255
High speed counter	HC0	HC0 to HC2	HC0 to HC2	HC0 to HC2
Sequential control relays	S0.0 to S7.7	S0.0 to S15.7	S0.0 to S31.7	S0.0 to S31.7
Accumulator registers	AC0 to AC3	AC0 to AC3	AC0 to AC3	AC0 to AC3
Jumps/Labels	0 to 63	0 to 255	0 to 255	0 to 255
Call/Subroutine	0 to 15	0 to 63	0 to 63	0 to 63
Interrupt routines	0 to 31	0 to 127	0 to 127	0 to 127
Interrupt events	0, 1, 8 to 10, 12	0 to 20	0 to 23	0 to 26
PID loops	Not supported	Not supported	0 to 7	0 to 7
Ports	0	0	0	0 and 1

Table 10-2 S7-200 CPU Operand Ranges

Access Method	CPU 212	CPU 214	CPU 215	CPU 216
Bit access (byte.bit)	V 0.0 to 1023.7 I 0.0 to 7.7 Q 0.0 to 7.7 M 0.0 to 15.7 SM 0.0 to 45.7 T 0 to 63 C 0 to 63 S 0.0 to 7.7	V 0.0 to 4095.7 I 0.0 to 7.7 Q 0.0 to 7.7 M 0.0 to 31.7 SM 0.0 to 85.7 T 0 to 127 C 0 to 127 S 0.0 to 15.7	V 0.0 to 5119.7 I 0.0 to 7.7 Q 0.0 to 7.7 M 0.0 to 31.7 SM 0.0 to 194.7 T 0 to 255 C 0 to 255 S 0.0 to 31.7	V 0.0 to 5119.7 I 0.0 to 7.7 Q 0.0 to 7.7 M 0.0 to 31.7 SM 0.0 to 194.7 T 0 to 255 C 0 to 255 S 0.0 to 31.7
Byte access	VB 0 to 1023 IB 0 to 7 QB 0 to 7 MB 0 to 15 SMB 0 to 45 AC 0 to 3 SB 0 to 7 Constant	VB 0 to 4095 IB 0 to 7 QB 0 to 7 MB 0 to 31 SMB 0 to 85 AC 0 to 3 SB 0 to 15 Constant	VB 0 to 5119 IB 0 to 7 QB 0 to 7 MB 0 to 31 SMB 0 to 194 AC 0 to 3 SB 0 to 31 Constant	VB 0 to 5119 IB 0 to 7 QB 0 to 7 MB 0 to 31 SMB 0 to 194 AC 0 to 3 SB 0 to 31 Constant
Word access	VW 0 to 1022 T 0 to 63 C 0 to 63 IW 0 to 6 QW 0 to 6 MW 0 to 14 SMW 0 to 44 AC 0 to 3 AIW 0 to 30 AQW 0 to 30 SW 0 to 6 Constant	VW 0 to 4094 T 0 to 127 C 0 to 127 IW 0 to 6 QW 0 to 6 MW 0 to 30 SMW 0 to 84 AC 0 to 3 AIW 0 to 30 AQW 0 to 30 SW 0 to 14 Constant	VW 0 to 5118 T 0 to 255 C 0 to 255 IW 0 to 6 QW 0 to 6 MW 0 to 30 SMW 0 to 193 AC 0 to 3 AIW 0 to 30 AQW 0 to 30 SW 0 to 30 Constant	VW 0 to 5118 T 0 to 255 C 0 to 255 IW 0 to 6 QW 0 to 6 MW 0 to 30 SMW 0 to 193 AC 0 to 3 AIW 0 to 30 AQW 0 to 30 SW 0 to 30 Constant
Double word access	VD 0 to 1020 ID 0 to 4 QD 0 to 4 MD 0 to 12 SMD 0 to 42 AC 0 to 3 HC 0 SD 0 to 4 Constant	VD 0 to 4092 ID 0 to 4 QD 0 to 4 MD 0 to 28 SMD 0 to 82 AC 0 to 3 HC 0 to 2 SD 0 to 12 Constant	VD 0 to 5116 ID 0 to 4 QD 0 to 4 MD 0 to 28 SMD 0 to 191 AC 0 to 3 HC 0 to 2 SD 0 to 28 Constant	VD 0 to 5116 ID 0 to 4 QD 0 to 4 MD 0 to 28 SMD 0 to 191 AC 0 to 3 HC 0 to 2 SD 0 to 28 Constant

## 10.2 Contact Instructions

### Standard Contacts



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

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 address *n* to the top of the stack.

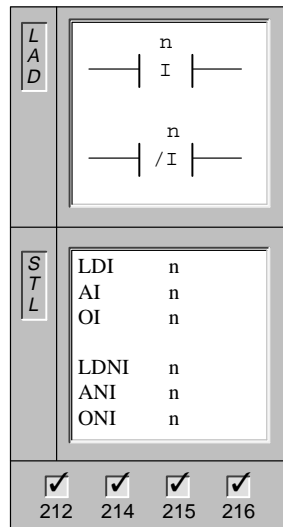
The **Normally Closed** contact is closed (on) when the bit value of address *n* is equal to 0.

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 address *n* to the top of the stack.

Operands:      *n*:            I, Q, M, SM, T, C, V, S

These instructions obtain the referenced value from the process-image register when it is updated at the beginning of each CPU scan.

### Immediate Contacts



The **Normally Open Immediate** contact is closed (on) when the bit value of the referenced physical input point *n* is equal to 1.

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 bit value of the referenced physical input point *n* to the top of the stack immediately.

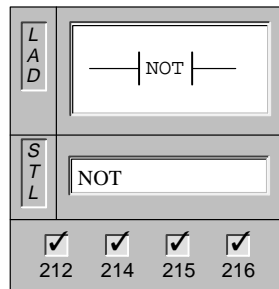
The **Normally Closed Immediate** contact is closed (on) when the bit value of the referenced physical input point *n* is equal to 0.

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

Operands:      *n*:            I

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

## Not

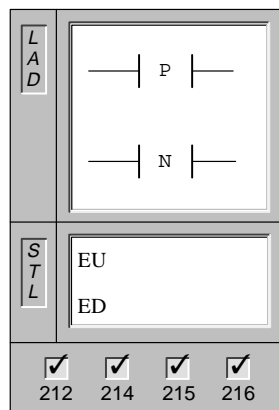


The **Not** contact changes the state of power flow. When power flow reaches the Not contact, it stops. When power flow does not reach the Not contact, it supplies power flow.

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

Operands: none

## Positive, Negative Transition



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

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.

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

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.

Operands: none

Contact Examples

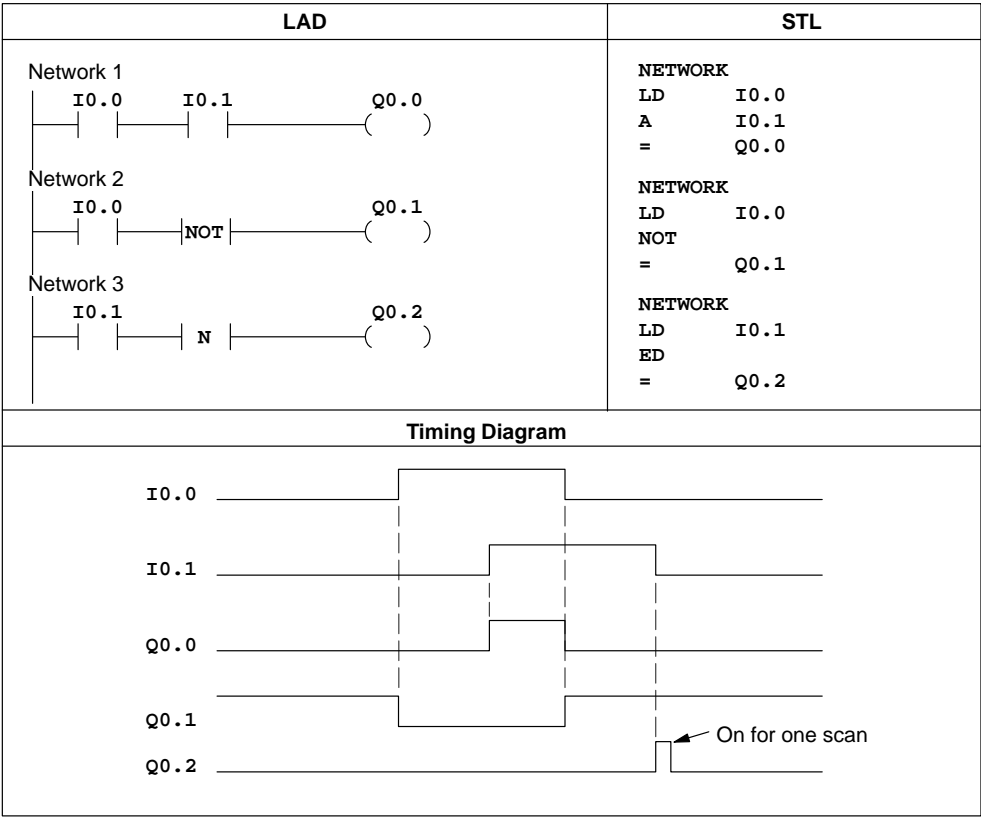
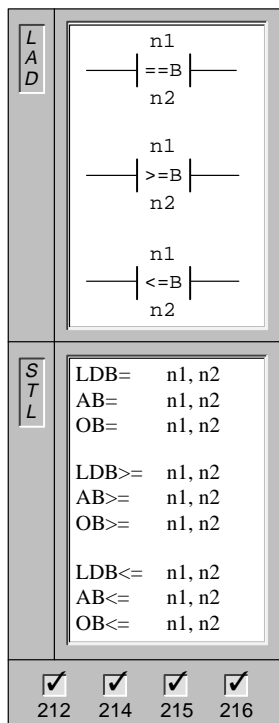


Figure 10-1 Examples of Boolean Contact Instructions for LAD and STL

## 10.3 Comparison Contact Instructions

### Compare Byte



The **Compare Byte** instruction is used to compare two values: n1 to n2. A comparison of  $n1 = n2$ ,  $n1 \geq n2$ , or  $n1 \leq n2$  can be made.

Operands: n1, n2: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

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

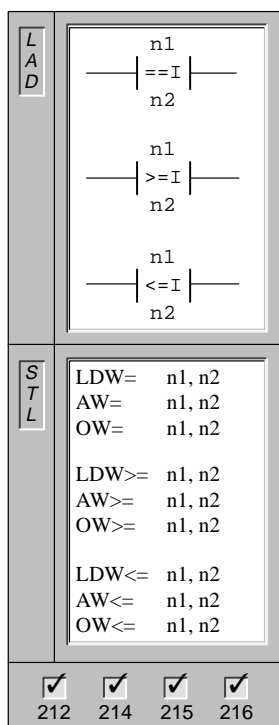
In STL, the instructions Load, AND, or OR a 1 with the top of stack when the comparison is true.

Byte comparisons are unsigned.

Note: You can create a  $<>$ ,  $<$ , or  $>$  comparison by using the Not instruction with the  $=$ ,  $\geq$ , or  $\leq$  Compare instruction. The following sequence is equivalent to a  $<>$  comparison of VB100 to 50:

```
LDB=  VB100, 50
NOT
```

### Compare Word Integer



The **Compare Word Integer** instruction is used to compare two values: n1 to n2. A comparison of  $n1 = n2$ ,  $n1 \geq n2$ , or  $n1 \leq n2$  can be made.

Operands: n1, n2: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

In LAD, the contact 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.

Word comparisons are signed ( $16\#7FFF > 16\#8000$ ).

Note: You can create a  $<>$ ,  $<$ , or  $>$  comparison by using the Not instruction with the  $=$ ,  $\geq$ , or  $\leq$  Compare instruction. The following sequence is equivalent to a  $<>$  comparison of VW100 to 50:

```
LDW=  VW100, 50
NOT
```

### Compare Double Word Integer

L A D			
S T L	LDD= n1, n2		
	AD= n1, n2		
	OD= n1, n2		
	LDD>= n1, n2		
	AD>= n1, n2		
	OD>= n1, n2		
	LDD<= n1, n2		
	AD<= n1, n2		
	OD<= n1, n2		
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
212	214	215	216

The **Compare Double Word** instruction is used to compare two values: n1 to n2. A comparison of n1 = n2, n1 >= n2, or n1 <= n2 can be made.

Operands: n1, n2: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD

In LAD, the contact 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.

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

Note: You can create a <>, <, or > comparison by using the Not instruction with the =, >=, or <= Compare instruction. The following sequence is equivalent to a <> comparison of VD100 to 50:

```
LDD= VD100, 50
NOT
```

### Compare Real

L A D			
S T L	LDR= n1, n2		
	AR= n1, n2		
	OR= n1, n2		
	LDR>= n1, n2		
	AR>= n1, n2		
	OR>= n1, n2		
	LDR<= n1, n2		
	AR<= n1, n2		
	OR<= n1, n2		
<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
212	214	215	216

The **Compare Real** instruction is used to compare two values: n1 to n2. A comparison of n1 = n2, n1 >= n2, or n1 <= n2 can be made.

Operands: n1, n2: VD, ID, QD, MD, SMD, AC, Constant, \*VD, \*AC, SD

In LAD, the contact 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.

Real comparisons are signed.

Note: You can create a <>, <, or > comparison by using the Not instruction with the =, >=, or <= Compare instruction. The following sequence is equivalent to a <> comparison of VD100 to 50:

```
LDR= VD100, 50.0
NOT
```



### Comparison Contact Examples

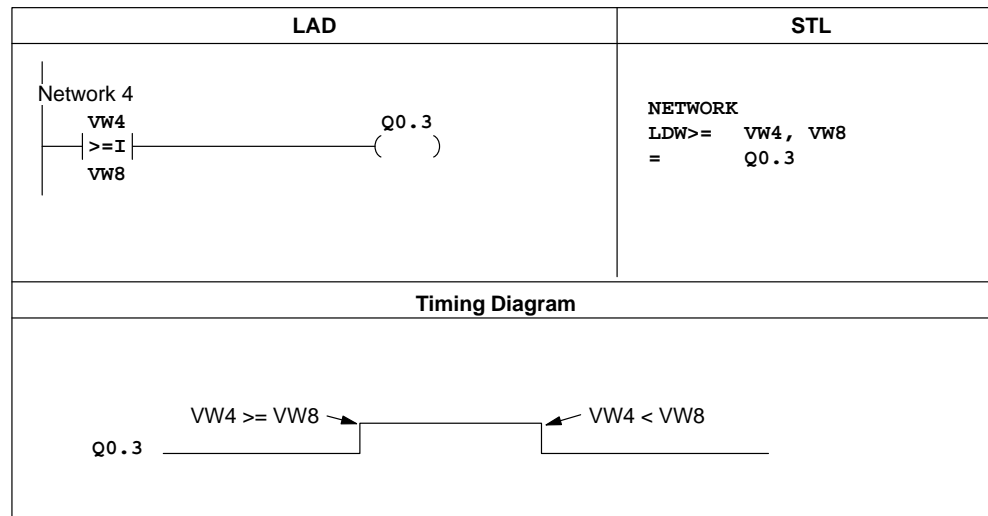
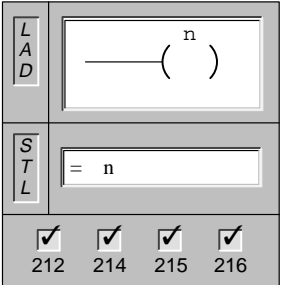


Figure 10-2 Examples of Comparison Contact Instructions for LAD and STL

## 10.4 Output Instructions

### Output

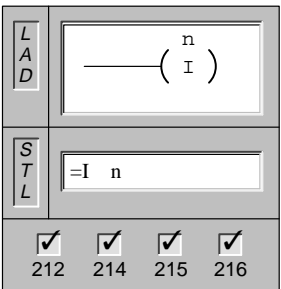


When the **Output** instruction is executed, the specified parameter (n) is turned on.

In STL, the output instruction copies the top of the stack to the specified parameter (n).

Operands:      n:            I, Q, M, SM, T, C, V, S

### Output Immediate



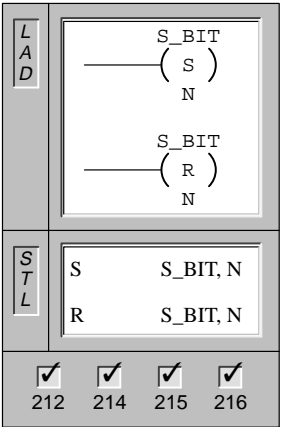
When the **Output Immediate** instruction is executed, the specified physical output point (n) is turned on immediately.

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

Operands:      n:            Q

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.

### Set, Reset

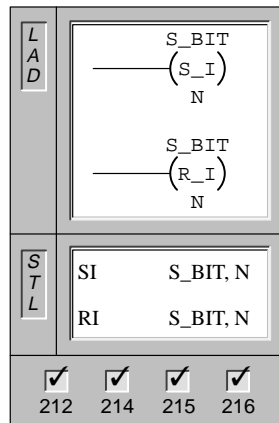


When the **Set** and **Reset** instructions are executed, the specified number of points (N) starting at the S\_BIT are set (turned on) or reset (turned off).

Operands:      S\_BIT:      I, Q, M, SM, T, C, V, S  
                    N:            IB, QB, MB, SMB, VB, AC, Constant, \*VD, \*AC, SB

The range of points that can be set or reset is 1 to 255. When using the Reset instruction, if the S\_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.

### Set, Reset Immediate



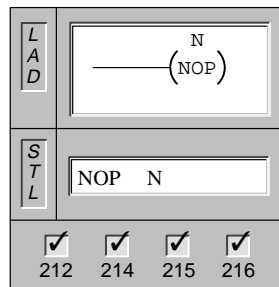
When the **Set Immediate** and **Reset Immediate** instructions are executed, the specified number of physical output points (N) starting at the S\_BIT are immediately set (turned on) or immediately reset (turned off).

Operands:    S\_BIT:    Q  
                   N:        IB, QB, MB, SMB, VB, AC, Constant, \*VD, \*AC, SB

The range of points that can be set or reset is 1 to 64.

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.

### 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:        0 to 255

If you use the NOP instruction, you must place it inside the main program, a subroutine, or an interrupt routine.

Output Examples

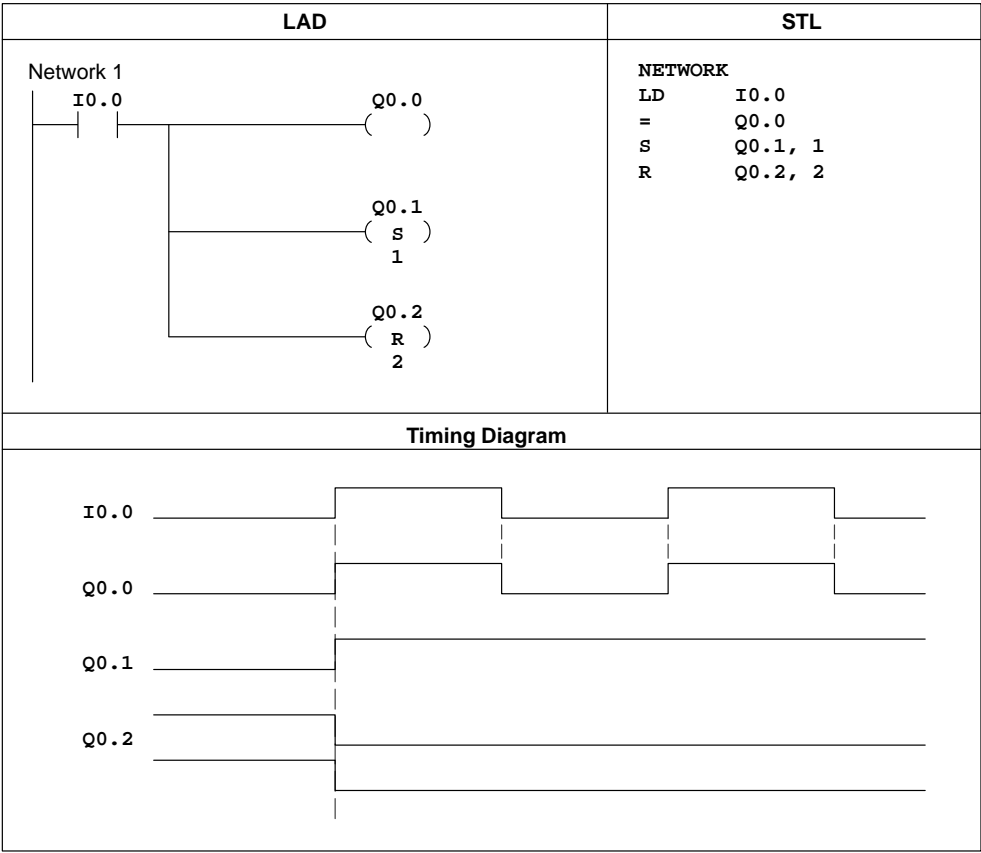
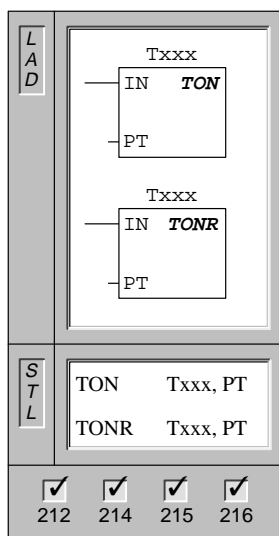


Figure 10-3 Examples of Output Instructions for LAD and STL

## 10.5 Timer, Counter, High-Speed Counter, High-Speed Output, Clock, and Pulse Instructions

### On-Delay Timer, Retentive On-Delay Timer



The **On-Delay Timer** and **Retentive On-Delay Timer** instructions time up to the maximum value when enabled. When the current value (Txxx) is  $\geq$  to the Preset Time (PT), the timer bit turns on.

The On-Delay timer is reset when disabled, while the Retentive On-Delay timer stops timing when disabled. Both timers stop timing when they reach the maximum value.

Operands:	Txxx:	<u>TON</u>	<u>TONR</u>
	1 ms	T32, T96	T0, T64
	10 ms	T33 to T36 T97 to T100	T1 to T4 T65 to T68
	100 ms	T37 to T63 T101 to T255	T5 to T31 T69 to T95
	PT:	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC, SW	

TON and TONR timers are available in three resolutions. The resolution is determined by the timer number and is shown in Table 10-3. Each count of the current value is a multiple of the time base. For example, a count of 50 on a 10-millisecond (ms) timer represents 500 ms.

Table 10-3 Timer Numbers and Resolutions

Timer	Resolution	Maximum Value	CPU 212	CPU 214	CPU 215/216
TON	1 ms	32.767 seconds (s)	T32	T32, T96	T32, T96
	10 ms	327.67 s	T33 to T36	T33 to T36, T97 to T100	T33 to T36, T97 to T100
	100 ms	3276.7 s	T37 to T63	T37 to T63, T101 to T127	T37 to T63, T101 to T255
TONR	1 ms	32.767 s	T0	T0, T64	T0, T64
	10 ms	327.67 s	T1 to T4	T1 to T4, T65 to T68	T1 to T4, T65 to T68
	100 ms	3276.7 s	T5 to T31	T5 to T31, T69 to T95	T5 to T31, T69 to T95

## Understanding the S7-200 Timer Instructions

You can use timers to implement time-based counting functions. The S7-200 provides two different timer instructions: the On-Delay Timer (TON), and the Retentive On-Delay Timer (TONR). The two types of timers (TON and TONR) differ in the ways that they react to the state of the enabling input. Both TON and TONR timers time up while the enabling input is on: the timers do not time up while the enabling input is off, but when the enabling input is off, a TON timer is reset automatically and a TONR timer is not reset and holds its last value. Therefore, the TON timer is best used when you are timing a single interval. The TONR timer is appropriate when you need to accumulate a number of timed intervals.

S7-200 timers have the following characteristics:

- Timers are controlled with a single enabling input, and have a current value that maintains the elapsed time since the timer was enabled. The timers also have a preset time value (PT) that is compared to the current value each time the current value is updated and when the timer instruction is executed.
- A timer bit is set or reset based upon the result of the comparison of current value to the preset time value.
- When the current value is greater than or equal to the preset time value, the timer bit (T-bit), is turned on.

---

### Note

Some timer current values can be made retentive. The timer bits are not retentive, and are set only as a result of the comparison between the current value and the preset value.

---

When you reset a timer, its current value is set to zero and its T-bit is turned off. You can reset any timer by using the Reset instruction, but using a Reset instruction is the only method for resetting a TONR timer. Writing a zero to a timer's current value does not reset its timer bit. In the same way, writing a zero to the timer's T-bit does not reset its current value.

Several 1-ms timers can also be used to generate an interrupt event. See Section 10.14 for information about timed interrupts.

## Updating Timers with 1-ms Resolution

The S7-200 CPU provides timers that are updated once per millisecond (1-ms timers) by the system routine that maintains the system time base. These timers provide precise control of an operation.

Since the current value of an active 1-ms timer is updated in a system routine, the update is automatic. Once a 1-ms timer has been enabled, the execution of the timer's controlling TON/TONR instruction is required only to control the enabled/disabled state of the timer.

Since the current value and T-bit of a 1-ms timer are updated by a system routine (independent from the programmable logic controller scan and the user program), the current value and T-bits of these timers can be updated anywhere in the scan and are updated more than once per scan if the scan time exceeds one millisecond. Therefore, these values are not guaranteed to remain constant throughout a given execution of the main user program.

Resetting an enabled 1-ms timer turns the timer off, resets the timer's current value to zero, and clears the timer T-bit.

---

**Note**

The system routine that maintains the 1-ms system time base is independent of the enabling and disabling of timers. A 1-ms timer is enabled at a point somewhere within the current 1-ms interval. Therefore, the timed interval for a given 1-ms timer can be up to 1 ms short. You should program the preset time value to a value that is 1 greater than the minimum desired timed interval. For example, to guarantee a timed interval of at least 56 ms using a 1-ms timer, you should set the preset time value to 57.

---

### Updating Timers with 10-ms Resolution

The S7-200 CPU provides timers that count the number of 10-ms intervals that have elapsed since the active 10-ms timer was enabled. These timers are updated at the beginning of each scan by adding the accumulated number of 10-ms intervals (since the beginning of the previous scan) to the current value for the timer.

Since the current value of an active 10-ms timer is updated at the beginning of the scan, the update is automatic. Once a 10-ms timer is enabled, execution of the timer's controlling TON/TONR instruction is required only to control the enabled or disabled state of the timer. Unlike the 1-ms timers, a 10-ms timer's current value is updated only once per scan and remains constant throughout a given execution of the main user program.

A reset of an enabled 10-ms timer turns it off, resets its current value to zero, and clears its T-bit.

---

**Note**

The process of accumulating 10-ms intervals is performed independently of the enabling and disabling of timers, so the enabling of 10-ms timers will fall within a given 10-ms interval. This means that a timed interval for a given 10-ms timer can be up to 10 ms short. You should program the preset time value to a value 1 greater than the minimum desired timed interval. For example, to guarantee a timed interval of at least 140 ms using a 10-ms timer, you should set the preset time value to 15.

---

### Updating Timers with 100-ms Resolution

Most of the timers provided by the S7-200 use a 100-ms resolution. These timers count the number of 100-ms intervals that have elapsed since the 100-ms timer was last updated. These timers are updated by adding the accumulated number of 100-ms intervals (since the beginning of the previous scan) to the timer's current value when the timer instruction is executed.

The update of 100-ms timers is not automatic, since 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, 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, the number of 100-ms intervals are added to the timer's current value multiple times, and it gains time. Therefore, 100-ms timers should only be used where the timer instruction is executed exactly once per scan. A reset of a 100-ms timer sets its current value to zero and clears its T-bit.

---

#### Note

The process of accumulating 100-ms intervals is performed independently of the enabling and disabling of timers, so a given 100-ms timer will be enabled at a point somewhere within the current 100-ms interval. This means that a timed interval for a given 100-ms timer can be up to 100 ms short. You should program the preset time value to a value 1 greater than the minimum desired timed 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 10-4.

- 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 (see Figure 10-4). Figure 10-5 and Figure 10-6 show examples of the Timer instructions for ladder logic and statement list.



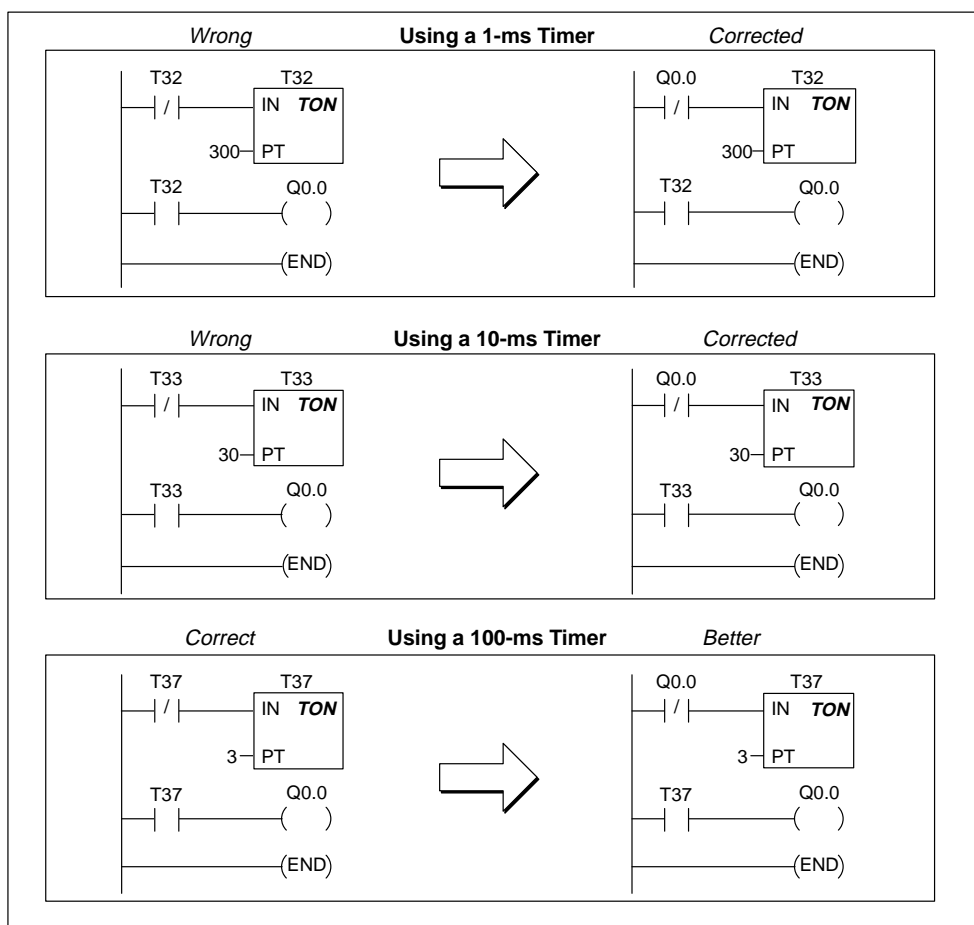


Figure 10-4 Example of Automatically Retriggered One Shot Timer

### On-Delay Timer Example

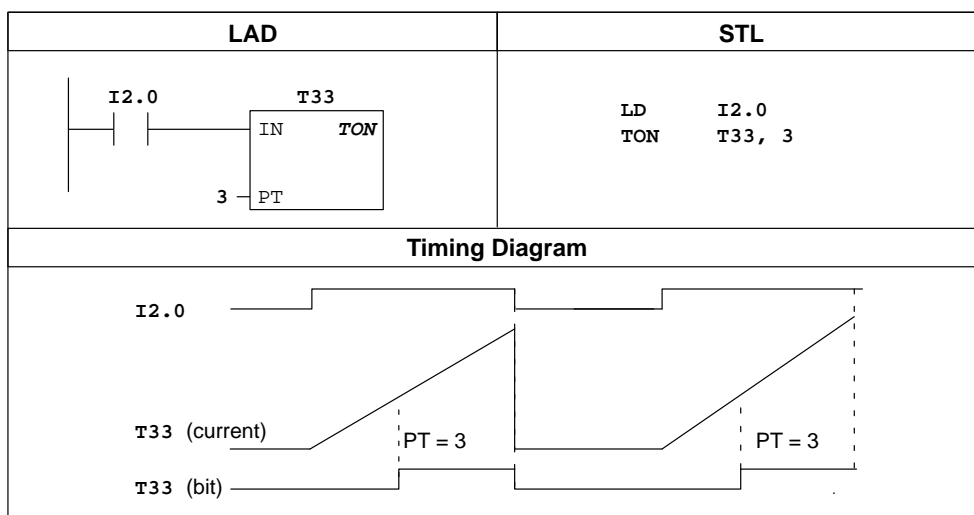


Figure 10-5 Example of On-Delay Timer Instruction for LAD and STL

Retentive On-Delay Timer Example

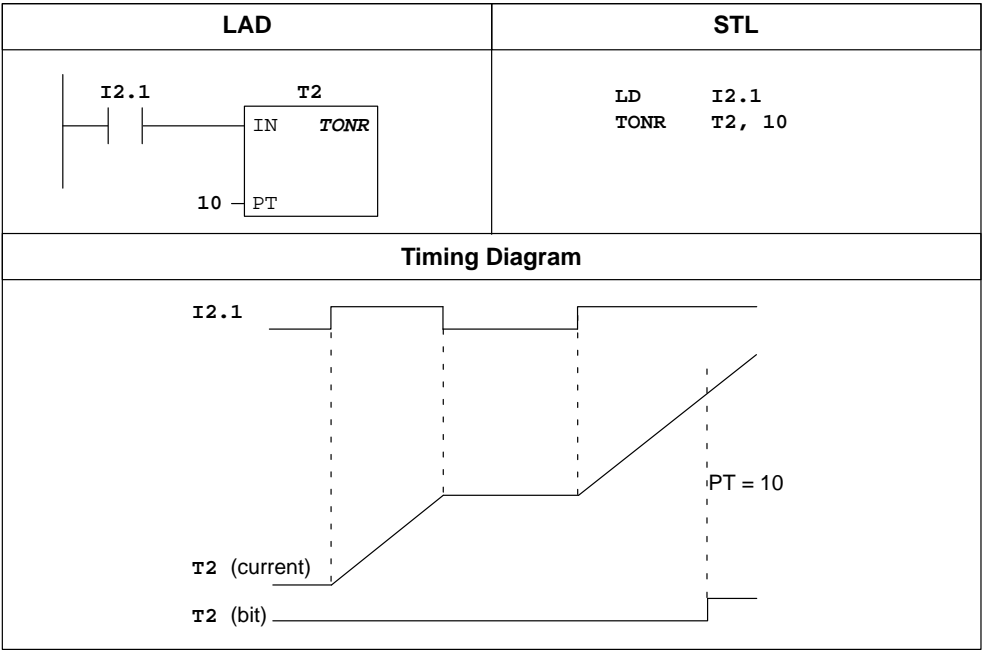
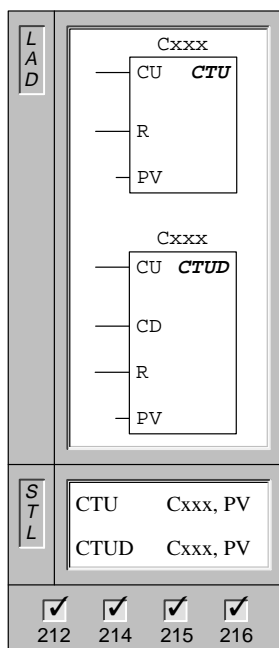


Figure 10-6 Example of Retentive On-Delay Timer Instruction for LAD and STL

## Count Up Counter, Count Up/Down Counter



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) 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.

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

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.

In STL, the 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.

Operands:     Cxxx:     0 to 255  
                   PV:        VW, T, C, IW, QW, MW, SMW, AC,  
                                  AIW, Constant, \*VD, \*AC, SW

## 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).

When you reset a counter using the Reset instruction, both the counter bit and the counter current value are reset.

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.

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 and Up/Down Counters access the same current value.)

Counter Example

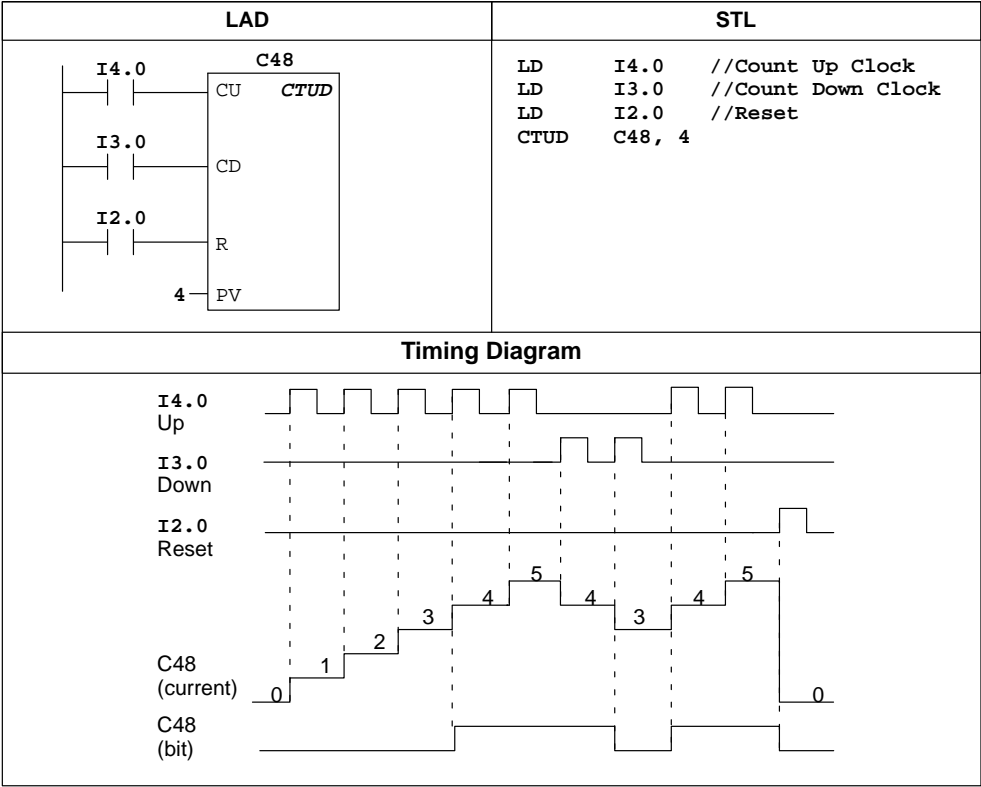
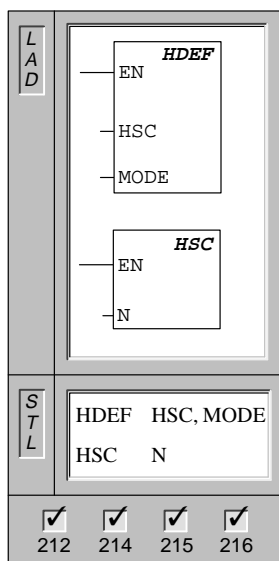


Figure 10-7 Example of Counter Instruction for LAD and STL

## 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 10-5.

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.

Only one HDEF box may be used per counter.

Operands:

HSC:	0 to 2
MODE:	0 (HSC0) 0 to 11 (HSC1 or 2)
N:	0 to 2

## Understanding the High-Speed Counter Instructions

High-speed counters count high-speed events that cannot be controlled at CPU scan rates.

- HSC0 is an up/down software counter that accepts a single clock input. The counting direction (up or down) is controlled by your program, using the direction control bit. The maximum counting frequency of HSC0 is 2 KHz.
- HSC1 and HSC2 are versatile hardware counters that can be configured for one of twelve different modes of operation. The counter modes are listed in Table 10-5. The maximum counting frequency of HSC1 and HSC2 is dependent on your CPU. See Appendix A.

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 time (1x) or four times (4x) the maximum counting rates. HSC1 and HSC2 are completely independent of each other and do not affect other high-speed functions. Both counters run at maximum rates without interfering with one another.

Figure 10-16 shows an example of the initialization of HSC1.

### 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 straight forward and easy to follow. Of course, all interrupt events can be processed in a single interrupt routine. For more information, see the section on Interrupt Instructions.

### Understanding the Detailed Timing for the High-Speed Counters

The following timing diagrams (Figure 10-8, Figure 10-9, Figure 10-10, and Figure 10-11) show how each counter functions according to category. The operation of the reset and start inputs is shown in a separate timing diagram and applies to all categories 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.

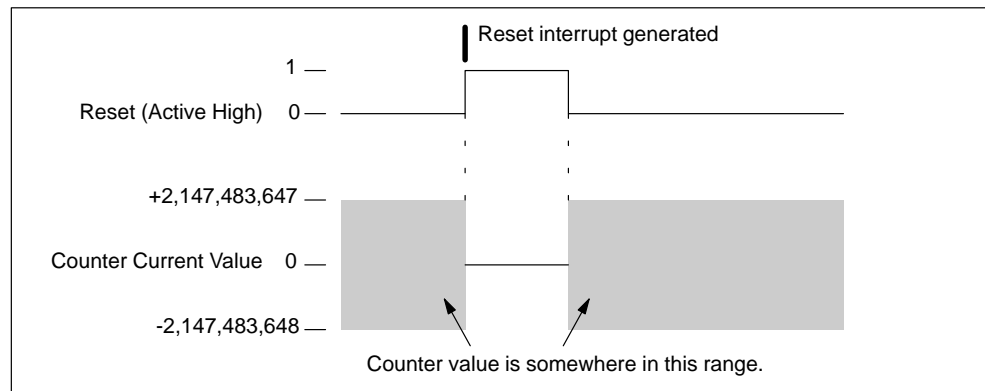


Figure 10-8 Operation Example with Reset and without Start

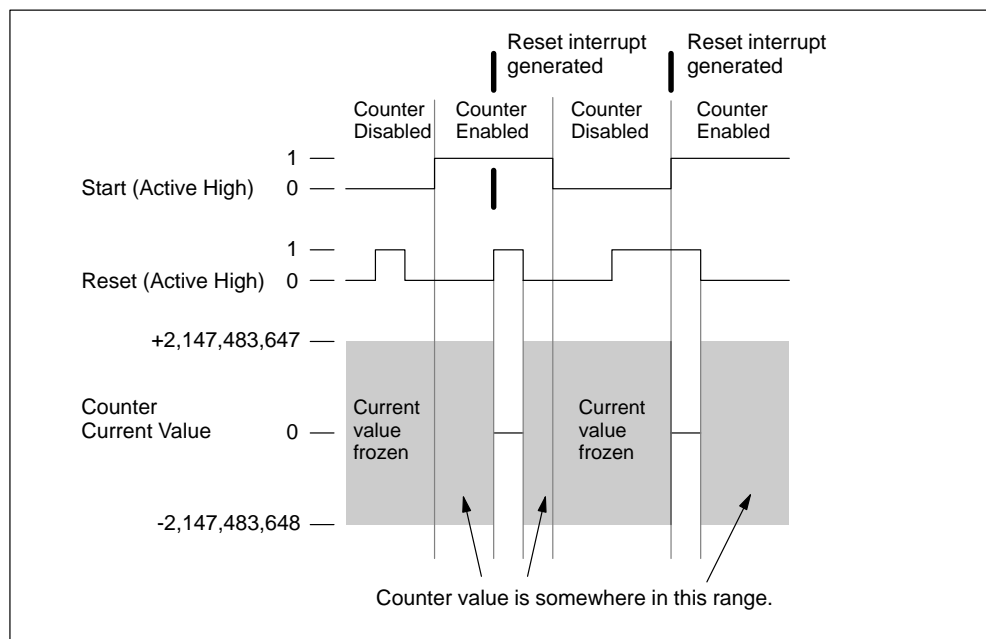


Figure 10-9 Operation Example with Reset and Start

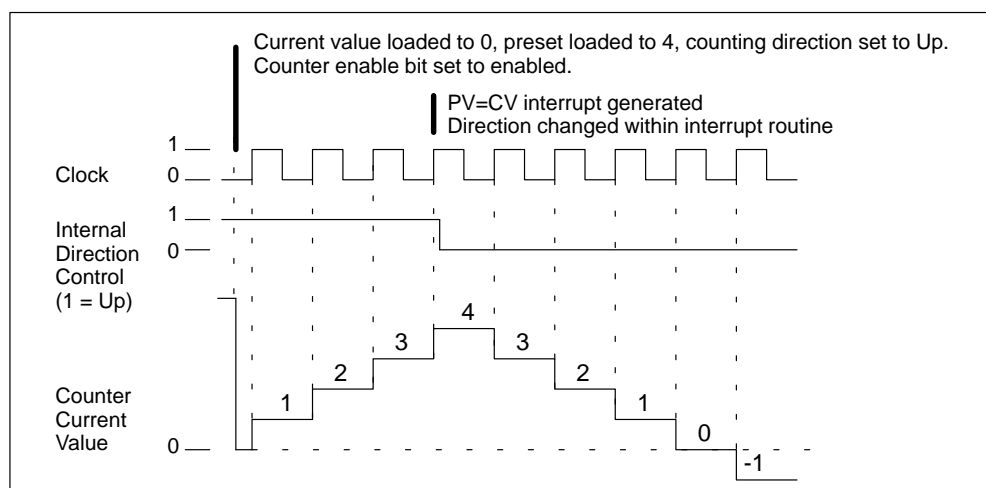


Figure 10-10 Operation Example of HSC0 Mode 0 and HSC1, or HSC2 Modes 0, 1, or 2

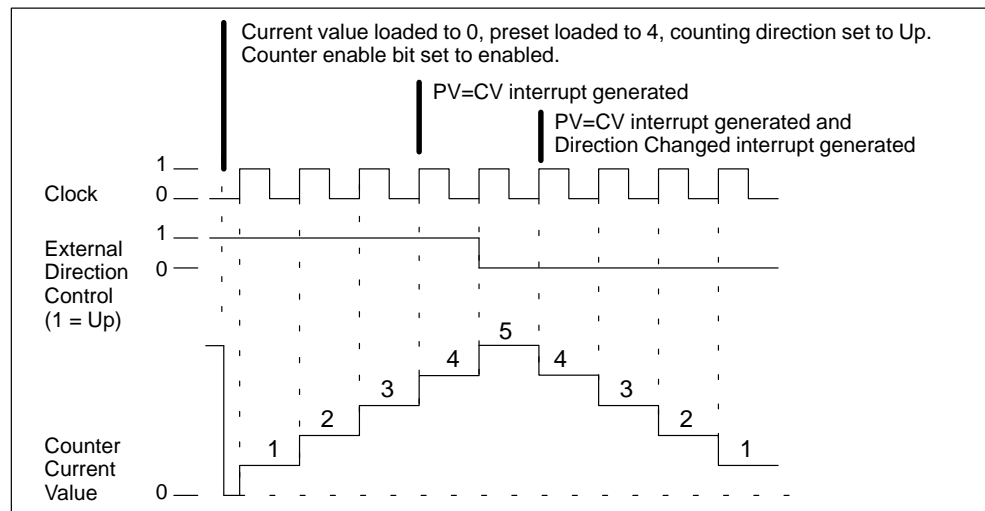


Figure 10-11 Operation Example of HSC1 or HSC2 Modes 3, 4, or 5

When you use counting modes 6, 7, or 8 in HSC1 or HSC2, 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 to each other. 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 10-12, Figure 10-13, and Figure 10-14.

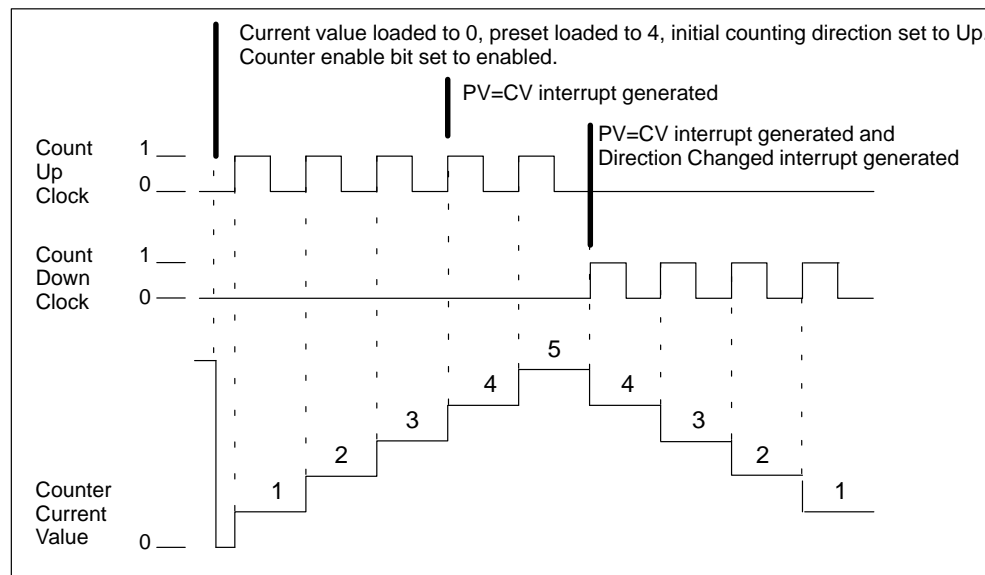


Figure 10-12 Operation Example of HSC1 or HSC2 Modes 6, 7, or 8



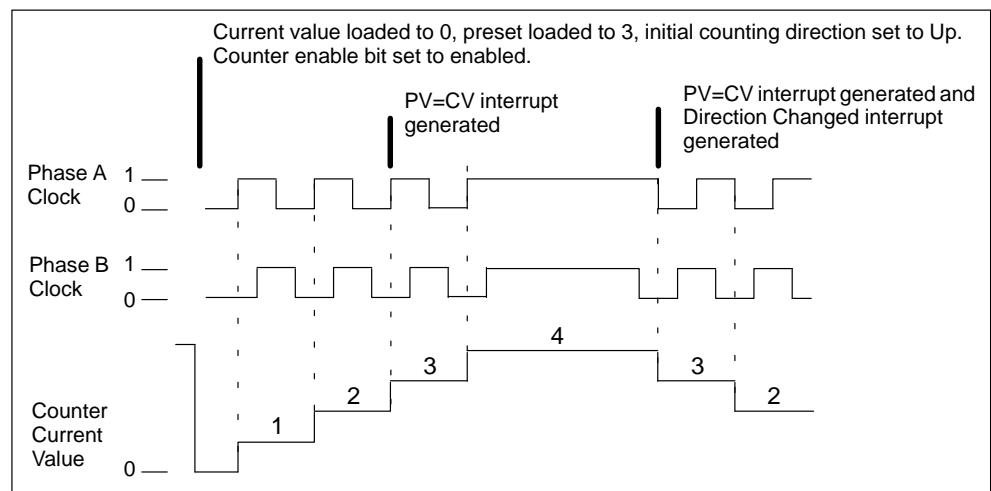


Figure 10-13 Operation Example of HSC1 or HSC2 Modes 9, 10, or 11 (Quadrature 1x Mode)

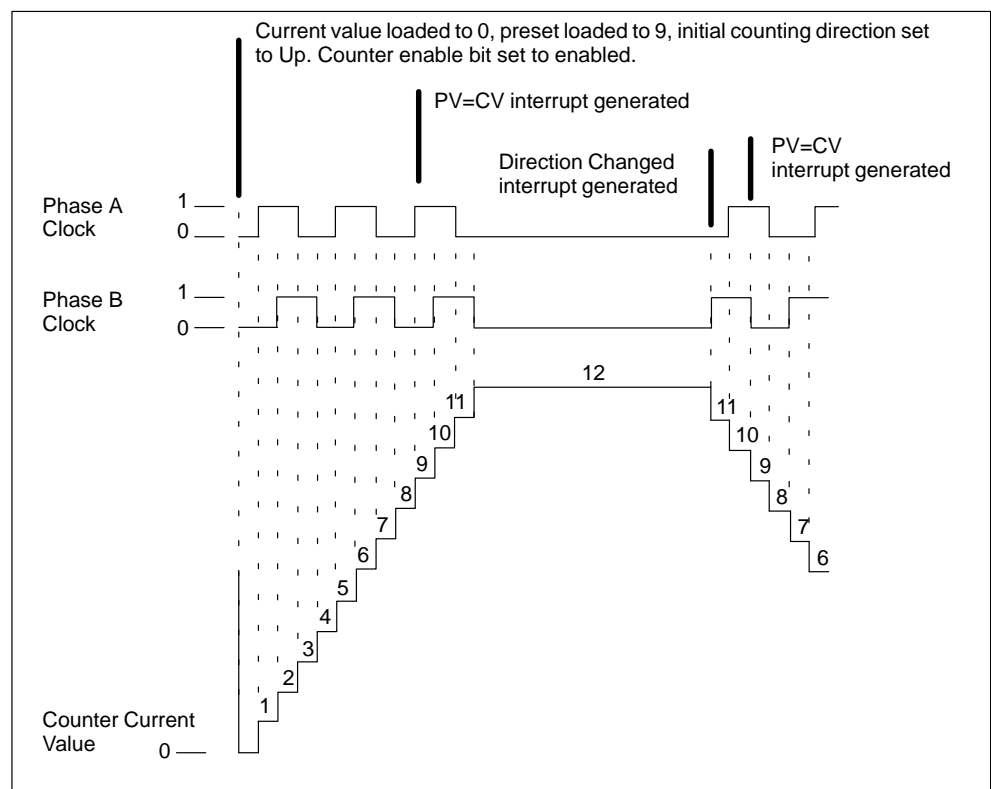


Figure 10-14 Operation Example of HSC1 or HSC2 Modes 9, 10, or 11 (Quadrature 4x Mode)

Connecting the Input Wiring for the High-Speed Counters

Table 10-4 shows the inputs used for the clock, direction control, reset, and start functions associated with the high-speed counters. These input functions are described in Table 10-5.

Table 10-4 Dedicated Inputs for High-Speed Counters

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

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 10-15.

Format: `HC[high-speed counter number] HC1`

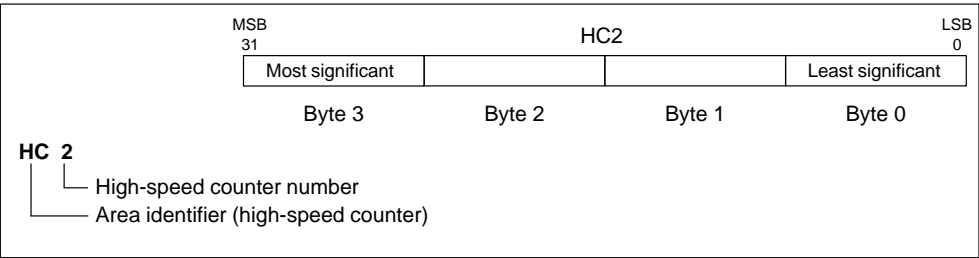


Figure 10-15 Accessing the High-Speed Counter Current Values

Table 10-5 HSC Modes of Operation

HSC0					
Mode	Description	I0.0			
0	Single phase up/down counter with internal direction control SM37.3 = 0, count down SM37.3 = 1, count up	Clock			
HSC1					
Mode	Description	I0.6	I0.7	I1.0	I1.1
0	Single phase up/down counter with internal direction control SM47.3 = 0, count down SM47.3 = 1, count up	Clock			
1				Reset	
2				Start	
3	Single phase up/down counter with external direction control I0.7 = 0, count down I0.7 = 1, count up	Clock	Dir.		
4				Reset	
5				Start	
6	Two-phase counter with count up and count down clock inputs	Clock (Up)	Clock (Dn)		
7				Reset	
8				Start	
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	
11				Start	
HSC2					
Mode	Description	I1.2	I1.3	I1.4	I1.5
0	Single phase up/down counter with internal direction control SM57.3 = 0, count down SM57.3 = 1, count up	Clock			
1				Reset	
2				Start	
3	Single phase up/down counter with external direction control I1.3 = 0, count down I1.3 = 1, count up	Clock	Dir.		
4				Reset	
5				Start	
6	Two phase counter with count up and count down clock inputs	Clock (Up)	Clock (Dn)		
7				Reset	
8				Start	
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	
11				Start	

### Understanding the Different High-Speed Counters (HSC0, HSC1, HSC2)

All counters (HSC0, HSC1, and HSC2) function the same way for the same counter mode of operation. There are four basic types of counter modes for HSC1 and HSC2 as shown in Table 10-5. 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, while the start input remains inactive. If the start input becomes active while reset remains active, 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 (HSC0, HSC1, or HSC2) 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

HSC1 and HSC2 have three control bits 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 10-6.

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 reset input and the start input are active high, and the quadrature counting rate is 4x (or four times the input clock frequency) for HSC1 and HSC2. Once the HDEF instruction has been executed, you cannot change the counter setup unless you first go to the STOP mode.

Table 10-6 Active Level Control for Reset and Start; 1x/4x Select Bits for HSC1 and HSC2

HSC1	HSC2	Description (used only when HDEF is executed)
SM47.0	SM57.0	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
SM47.2	SM57.2	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 10-7 describes each of these control bits.

Table 10-7 Control Bits for HSC0, HSC1, and HSC2

HSC0	HSC1	HSC2	Description
SM37.0	SM47.0	SM57.0	Not used after HDEF has been executed (Never used by HSC0)
SM37.1	SM47.1	SM57.1	Not used after HDEF has been executed (Never used by HSC0)
SM37.2	SM47.2	SM57.2	Not used after HDEF has been executed (Never used by HSC0)
SM37.3	SM47.3	SM57.3	Counting direction control bit: 0 = count down; 1 = count up
SM37.4	SM47.4	SM57.4	Write the counting direction to the HSC: 0 = no update; 1 = update direction
SM37.5	SM47.5	SM57.5	Write the new preset value to the HSC: 0 = no update; 1 = update preset
SM37.6	SM47.6	SM57.6	Write the new current value to the HSC: 0 = no update; 1 = update current value
SM37.7	SM47.7	SM57.7	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 10-8 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, or 2) 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 10-8 Current and Preset Values of HSC0, HSC1, and HSC2

Current Value of HSC0, HSC1, and HSC2			
HSC0	HSC1	HSC2	Description
SM38	SM48	SM58	Most significant byte of the new 32-bit current value
SM39	SM49	SM59	The next-to-most significant byte of the new 32-bit current value
SM40	SM50	SM60	The next-to-least significant byte of the new 32-bit current value
SM41	SM51	SM61	The least significant byte of the new 32-bit current value
Preset Value of HSC0, HSC1, and HSC2			
HSC0	HSC1	HSC2	Description
SM42	SM52	SM62	Most significant byte of the new 32-bit preset value
SM43	SM53	SM63	The next-to-most significant byte of the new 32-bit preset value
SM44	SM54	SM64	The next-to-least significant byte of the new 32-bit preset value
SM45	SM55	SM65	The least significant byte of the new 32-bit preset value

### Status Byte

A status byte is provided for each high-speed counter that provides status memory bits that indicate the current counting direction, if the current value equals preset value, and if the current value is greater than preset. Table 10-9 defines each of these status bits for each high-speed counter.

Table 10-9 Status Bits for HSC0, HSC1, and HSC2

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

---

**Note**

Status bits for HSC0, HSC1, and HSC2 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

HSC0 supports one interrupting condition: interrupt on current value equal to preset value. HSC1 and HSC2 provide three interrupting conditions: interrupt on current value equal to preset value, interrupt on external reset activated, and 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 the Interrupt Instructions.

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 SM47 according to the desired control operation. For example:  

SM47 = 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 SM48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SM52 (double word size value) with the desired preset value.
6. In order to capture the event of current value equal to preset, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See the section on Interrupt Instructions in this chapter 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 HSC1 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 SM47 according to the desired control operation. For example:  

SM47 = 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 SM48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SM52 (double word size value) with the desired preset value.
6. In order to capture the event of current value equal to preset, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Interrupt Instructions 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 HSC1 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 SM47 according to the desired control operation. For example:  
     SM47 = 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 SM48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SM52 (double word size value) with the desired preset value.
6. In order to capture the event of current value equal to preset, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Interrupt Instructions 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 HSC1 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 SM47 according to the desired control operation.

For example (1x counting mode):

SM47 = 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):

SM47 = 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 SM48 (double word size value) with the desired current value (load with 0 to clear it).
5. Load SM52 (double word size value) with the desired preset value.
6. In order to capture the event of current value equal to preset, program an interrupt by attaching the CV = PV interrupt event (event 13) to an interrupt routine. See Interrupt Instructions 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 HSC1 interrupts.
10. Execute the HSC instruction to cause the S7-200 to program HSC1.
11. Exit the subroutine.

**Change Direction 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 SM47 to write the desired direction:  

SM47 = 16#90	Enables the counter
	Sets the direction of the HSC to count down
SM47 = 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 SM47 to write the desired current value:  

SM47 = 16#C0	Enables the counter
	Writes the new current value
2. Load SM48 (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 SM47 to write the desired preset value:  

SM47 = 16#A0	Enables the counter
	Writes the new preset value
2. Load SM52 (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 SM47 to disable the counter:  

SM47 = 16#00	Disables the counter
--------------	----------------------
2. Execute the HSC instruction to disable the counter.

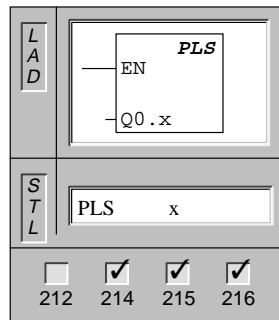
Although the above sequences show how to change direction, current value, and preset value individually, you may change all or any combination of them in the same sequence by setting the value of SM47 appropriately and then executing the HSC instruction.

# High-Speed Counter Example

LAD	STL
<p>Network 1</p> <p>Network 2</p> <p>Network 3</p> <p>Network 4</p> <p>Network 5</p> <p>Network 6</p> <p>Network 7</p> <p>Network 8</p>	<p>Network 1</p> <pre>LD SM0.1 CALL 0</pre> <p>Network 2</p> <pre>END</pre> <p>Network 3</p> <pre>SBR 0</pre> <p>Network 4</p> <pre>LD SM0.0 MOVB 16#F8, SMB47 HDEF 1, 11 MOVD 0, SMD48 MOVD 50, SMD52 ATCH 0, 13 ENI HSC 1</pre> <p>Network 5</p> <pre>RET</pre> <p>Network 6</p> <pre>INT 0</pre> <p>Network 7</p> <pre>LD SM 0.0 MOVD 0, SMD48 MOVB 16#C0, SMB47 HSC 1</pre> <p>Network 8</p> <pre>RETI</pre>

Figure 10-16 Example of Initialization of HSC1

## Pulse



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

Operands:      x:            0 to 1

## Understanding the S7-200 High-Speed Output Instructions

Some CPUs allow Q0.0 and Q0.1 either to generate high-speed pulse train outputs (PTO) or to perform pulse width modulation (PWM) control. The pulse train function provides a square wave (50% duty cycle) output for a specified number of pulses and a specified cycle time. The number of pulses can be specified from 1 to 4,294,967,295 pulses. The cycle time can be specified in either microsecond or millisecond increments either from 250 microseconds to 65,535 microseconds or from 2 milliseconds to 65,535 milliseconds. Specifying any odd number of microseconds or milliseconds causes some duty cycle distortion.

The PWM function provides a fixed cycle time with a variable duty cycle output. The cycle time and the pulse width can be specified in either microsecond or millisecond increments. The cycle time has a range either from 250 microseconds to 65,535 microseconds or from 2 milliseconds to 65,535 milliseconds. The pulse width time has a range either from 0 to 65,535 microseconds or from 0 to 65,535 milliseconds. When the pulse width is equal to the cycle time, the duty cycle is 100 percent and the output is turned on continuously. When the pulse width is zero, the duty cycle is 0 percent 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.

### Note

In the PTO and PWM functions, the switching times of the outputs from off to on and from on to off are not the same. This difference in the switching times manifests itself as duty cycle distortion. Refer to Appendix A for switching time specifications. The PTO/PWM outputs must have a minimum load of at least 10 percent of rated load to provide crisp transitions from off to on and from on to off.

### Changing the Pulse Width

PWM is a continuous function. Changing the pulse width causes the PWM function to be disabled momentarily while the update is made. This is done asynchronously to the PWM cycle, and could cause undesirable jitter in the controlled device. If synchronous updates to the pulse width are required, the pulse output is fed back to one of the interrupt input points (I0.0 to I0.4). By enabling the rising edge interrupt of that input when the pulse width needs to be changed, you can synchronize the PWM cycle. See Figure 10-19 for an example.

The pulse width is actually changed in the interrupt routine and the interrupt event is detached or disabled in the interrupt routine. This prevents interrupts from occurring except when the pulse width is to be changed.

### Invoking the PTO/PWM Operation

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

### PTO Pipeline

In addition to the control information, there are two status bits used with the PTO function that either indicate that the specified number of pulses were generated, or that a pipeline overflow condition has occurred.

The PTO function allows two pulse output specifications to be either chained together or to be piped one after the other. By doing this, continuity between subsequent output pulse trains can be supported. You load the pipeline by setting up the first PTO specification and then executing the PLS instruction. Immediately after executing the PLS instruction, you can set up the second specification, and execute another PLS instruction.

If a third specification is made before the first PTO function is completed (before the number of output pulses of the first specification are generated), the PTO pipeline overflow bit (SM66.6 or SM76.6) is set to one. This bit is set to zero on entry into RUN mode. It must be set to zero by the program after an overflow is detected, if subsequent overflows are to be detected.

The SM locations for pulse outputs 0 and 1 are shown in Table 10-10.

---

#### Note

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

---

Table 10-10 PTO/PWM Locations for Piping Two Pulse Outputs

Q0.0	Q0.1	Status Bit for Pulse Outputs
SM66.6	SM76.6	PTO pipeline overflow 0 = no overflow; 1 = overflow
SM66.7	SM76.7	PTO idle 0 = in progress; 1 = PTO idle
Q0.0	Q0.1	Control Bits for PTO/PWM Outputs
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 $\mu$ s/tick; 1 = 1ms/tick
SM67.4	SM77.4	Not used
SM67.5	SM77.5	Not used
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	Cycle Time Values for PTO/PWM Outputs (Range: 2 to 65,535)
SM68	SM78	Most significant byte of the PTO/PWM cycle time value
SM69	SM79	Least significant byte of the PTO/PWM cycle time value
Q0.0	Q0.1	Pulse Width Values for PWM Outputs (Range: 0 to 65,535)
SM70	SM80	Most significant byte of the PWM pulse width value
SM71	SM81	Least significant byte of the PWM pulse width value
Q0.0	Q0.1	Pulse Count Values for Pulse Outputs (Range: 1 to 4,294,967,295)
SM72	SM82	Most significant byte of the PTO pulse count value
SM73	SM83	Next-to-most significant byte of the PTO pulse count value
SM74	SM84	Next-to-least significant byte of the PTO pulse count value
SM75	SM85	Least significant byte of the PTO pulse count value

You can use Table 10-11 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.

Table 10-11 PTO/PWM Hexadecimal Reference Table

Control Register (Hex Value)	Result of executing the PLS instruction					
	Enable	Select Mode	Time Base	Pulse Count	Pulse Width	Cycle Time
16#81	Yes	PTO	1 $\mu$ s/tick			Load
16#84	Yes	PTO	1 $\mu$ s/tick	Load		
16#85	Yes	PTO	1 $\mu$ s/tick	Load		Load
16#89	Yes	PTO	1 ms/tick			Load
16#8C	Yes	PTO	1 ms/tick	Load		
16#8D	Yes	PTO	1 ms/tick	Load		Load
16#C1	Yes	PWM	1 $\mu$ s/tick			Load
16#C2	Yes	PWM	1 $\mu$ s/tick		Load	
16#C3	Yes	PWM	1 $\mu$ s/tick		Load	Load
16#C9	Yes	PWM	1 ms/tick			Load
16#CA	Yes	PWM	1 ms/tick		Load	
16#CB	Yes	PWM	1 ms/tick		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 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 to set the output to 1, 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 SM67 with a value of 16#C3 for PWM using microsecond increments (or 16#CB 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 SM68 (word size value) with the desired cycle time.
4. Load SM70 (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 SM67 with a value of 16#C2 for microsecond increments (or 16#CA for millisecond increments). This resets the update cycle time value in the control byte and allows the pulse width to change. A new pulse width value is loaded, and the PLS instruction is executed without modifying the control byte.
7. Exit the subroutine.

Optional steps for synchronous updates. If synchronous updates are required, follow these steps:

1. Execute the global interrupt enable instruction (ENI).
2. Using the condition you will use to update pulse width, enable (ATCH) a rising edge event to an interrupt routine. The condition you use to attach the event should be active for only one scan.
3. Add an interrupt routine that updates the pulse width, and then disables the edge interrupt.

---

### Note

The optional steps for synchronous updates require that the PWM output is fed back to one of the interrupt inputs.

---

## Changing the Pulse Width for PWM Outputs

To change the pulse width for PWM outputs in a subroutine, follow these steps:

1. Call a subroutine to load SM70 (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

To initialize the PTO, follow these steps:

1. Use the first scan memory bit to reset the output to 0, and call the subroutine that you need to perform the initialization operations. When you call a subroutine, 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 SM67 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 SM68 (word size value) with the desired cycle time.
4. Load SM72 (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, and execute the global interrupt enable instruction. Refer to Section 10.14 Interrupt Instructions 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

To change the PTO Cycle Time in an interrupt routine or subroutine, follow these steps:

1. Load SM67 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 SM68 (word size value) with the desired cycle time.
3. Execute the PLS instruction to cause the S7-200 to program the PTO/PWM generator.
4. Exit the interrupt routine or the subroutine. (Subroutines cannot be called from interrupt routines.)

### Changing the PTO Count

To change the PTO Count in an interrupt routine or a subroutine, follow these steps:

1. Load SM67 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 SM72 (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.
4. Exit the interrupt routine or the subroutine. (Subroutines cannot be called from interrupt routines.)

### Changing the PTO Cycle Time and the Pulse Count

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

1. Load SM67 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 SM68 (word size value) with the desired cycle time.
3. Load SM72 (double word size value) with the desired pulse count.
4. Execute the PLS instruction so that the S7-200 programs the PTO/PWM generator.
5. Exit the interrupt routine or the subroutine. (Subroutines cannot be called from interrupt routines.)

### Active PTO/PWM

When a PTO or a PWM function is active on Q0.0 or Q0.1, the normal use of Q0.0 or Q0.1, respectively, is inhibited. Neither the values stored in the process-image register nor any forced values for these points are transferred to the output as long as either the PTO or PWM function is active. A PTO is active when enabled and not complete. Immediate output instructions, writing to these points while PTO or PWM outputs are active, do not cause disruptions to either the PTO or PWM wave form.

---

#### Note

If a PTO function is disabled before completion, then the current pulse train is terminated, and the output Q0.0 or Q0.1 reverts to normal image register control. Reenabling the PTO function causes the pulse train to restart from the beginning using the last pulse output specification loaded.

---

Effects on Outputs

The PTO/PWM function and the process-image register share the use of the outputs Q0.0 and Q0.1. The initial and final states of the PTO and PWM waveforms are affected by the value of the corresponding process-image register bit. When a pulse train is output on either Q0.0 or Q0.1, the process-image register determines the initial and final states of the output, and causes the pulse output to start from either a high or a low level.

Since both the PTO and PWM functions are disabled momentarily between PTO pipelined changes and PWM pulse width changes, a small discontinuity in the output waveforms can exist at the change points. To minimize any adverse effects of this discontinuity, always use the PTO function with the process-image register bit set to a 0, and use the PWM function with the process-image register bit set to a 1. The resulting waveforms of both the PTO and PWM functions are shown in Figure 10-17. Notice that in the case of the PTO function at the change point, the last half-cycle is shortened to a pulse width of about 120  $\mu$ s. In the case of the PWM function using the optional sequence for synchronous update, the first high time pulse after the change is increased by about 120  $\mu$ s.

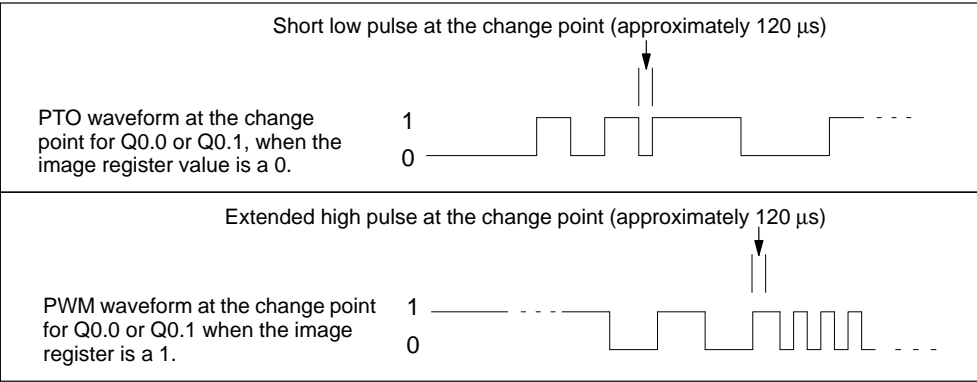


Figure 10-17 Sample Pulse Train Shapes for Q0.0 or Q0.1

## Example of Pulse Train Output

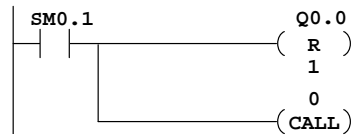


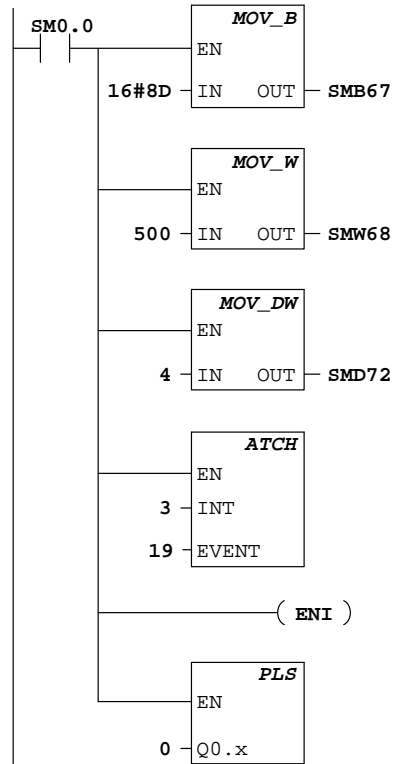

LAD		STL
Network 1	 <p>On the first scan, reset image register bit low, and call subroutine 0.</p>	<b>Network 1</b> LD SM0.1 R Q0.0, 1 CALL 0
Network 2	 <p>End of main ladder.</p>	<b>Network 2</b> MEND
Network 3	 <p>Start of subroutine 0.</p>	<b>Network 3</b> SBR 0
Network 4	 <p>Set up PTO 0 control byte:  - select PTO operation  - select ms increments  - set the pulse count and cycle time values  - enable the PTO function  Set cycle time to 500 ms.</p> <p>Set pulse count to 4 pulses.</p> <p>Define interrupt routine 3 to be the interrupt for processing PTO 0 interrupts.</p> <p>Global interrupt enable.</p> <p>Invoke PTO 0 operation. PLS 0 =&gt; Q0.0</p>	<b>Network 4</b> LD SM0.0 MOVB 16#8D, SMB67 MOVW 500, SMW68 MOVD 4, SMD72 ATCH 3, 19 ENI PLS 0
Network 5	 <p>Terminate subroutine.</p>	<b>Network 5</b> RET

Figure 10-18 Example of a Pulse Train Output

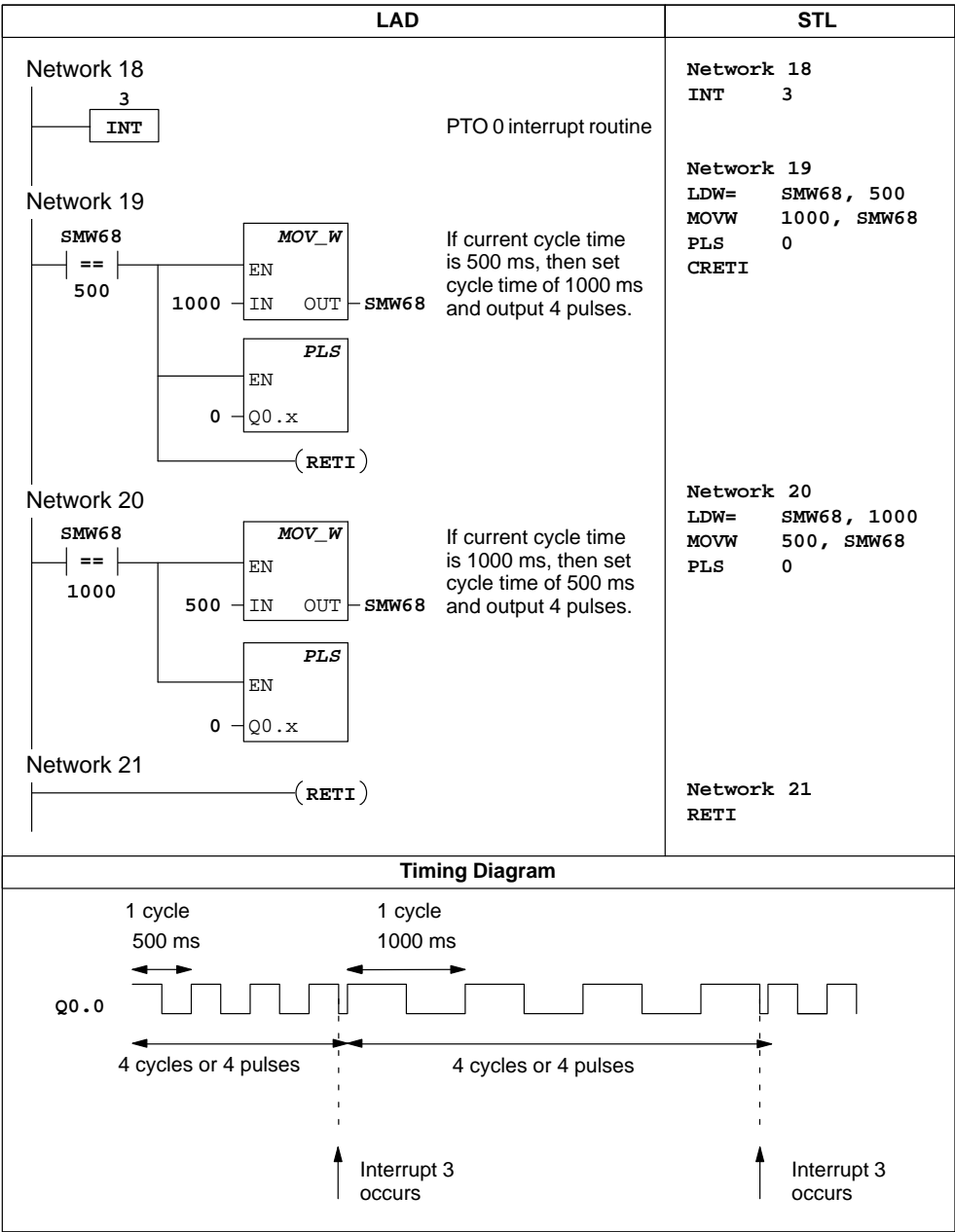


Figure 10-18Example of a Pulse Train Output (continued)

### Example of Pulse Width Modulation

Figure 10-19 shows an example of the Pulse Width Modulation. Changing the pulse width causes the PWM function to be disabled momentarily while the update is made. This is done asynchronously to the PWM cycle, and could cause undesirable jitter in the controlled device. If synchronous updates to the pulse width are required, the pulse output is fed back to the interrupt input point (I0.0). When the pulse width needs to be changed, the input interrupt is enabled, and on the next rising edge of I0.0, the pulse width will be changed synchronously to the PWM cycle.

The pulse width is actually changed in the interrupt routine and the interrupt event is detached or disabled in the interrupt routine. This prevents interrupts from occurring except when the pulse width is to be changed.

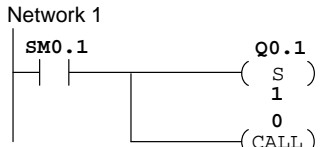
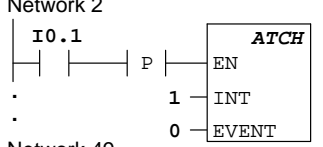

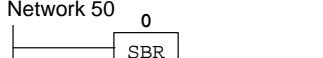
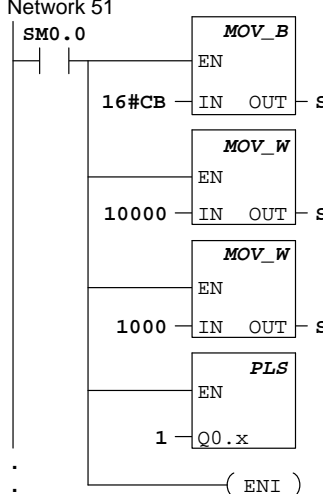

LAD	STL
<p>Network 1</p>  <p>On the first scan, set image register bit high, and call subroutine 0.</p> <p>Network 2</p>  <p>Feedback Q0.1 to I0.0, and attach a rising edge event to INT 1. This updates the pulse width synchronous with the pulse width cycle after I0.1 turns on.</p> <p>Network 49</p>  <p>End of main ladder.</p> <p>Network 50</p>  <p>Start of subroutine 0.</p> <p>Network 51</p>  <p>Set up PWM1 control byte:  - select PWM operation  - select ms increments  - set the pulse width and cycle time values  - enable the PWM function</p> <p>Set cycle time to 10,000 ms.</p> <p>Set pulse width to 1,000 ms.</p> <p>Invoke PWM 1 operation.  PLS 1 =&gt; Q0.1</p> <p>Enable all interrupts.</p> <p>Network 59</p>  <p>(Program continues on next page.)</p>	<p>Network 1</p> <pre>LD SM0.1 S Q0.1, 1 CALL 0</pre> <p>Network 2</p> <pre>LD I0.1 EU ATCH 1, 0 .</pre> <p>Network 49</p> <pre>MEND</pre> <p>Network 50</p> <pre>SBR 0</pre> <p>Network 51</p> <pre>LD SM0.0 MOVB 16#CB, SMB77 MOVW 10000, SMW78 MOVW 1000, SMW80 PLS 1 ENI .</pre> <p>Network 59</p> <pre>RET</pre>

Figure 10-19 Example of High-Speed Output Using Pulse Width Modulation

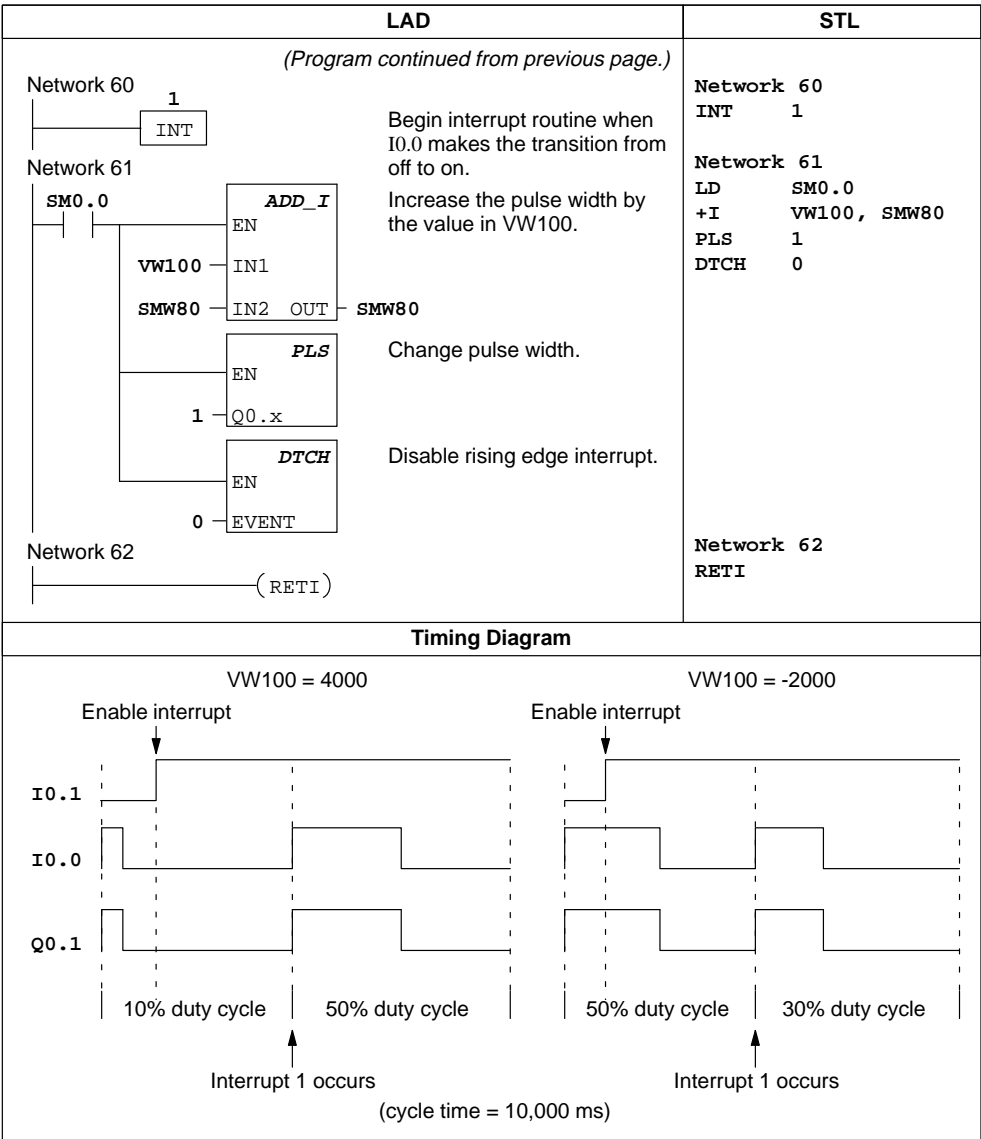
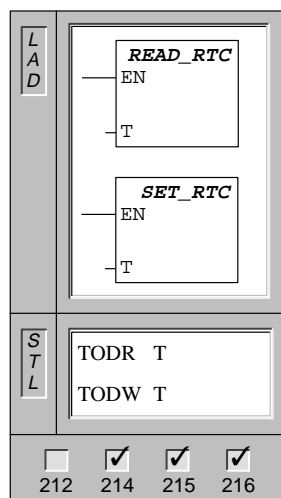


Figure 10-19 Example of High-Speed Output Using Pulse Width Modulation (continued)



## 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 Read\_RTC and Set\_RTC instructions are represented as Time of Day Read (TODR) and Time of Day Write (TODW).

Operands: T: VB, IB, QB, MB, SMB, \*VD, \*AC, SB

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	000d	d - 0 to 7	1 = Sunday
			0 = disables day of week (remains 0)

### Note

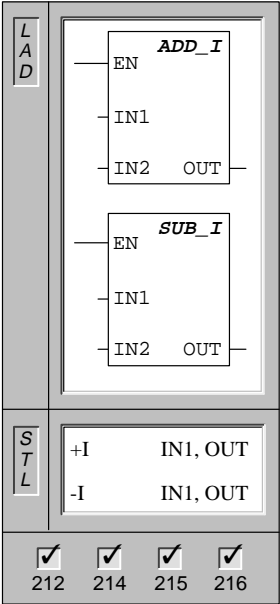
The S7-200 CPU does not perform a check to verify that the day of week is correct based upon the date. Invalid dates, such as February 30, may be accepted. You should ensure that the date you enter is correct.

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.

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.

## 10.6 Math and PID Loop Control Instructions

### Add, Subtract Integer



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

Operands: IN1, IN2: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

OUT: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

In LAD:  $IN1 + IN2 = OUT$   
 $IN1 - IN2 = OUT$

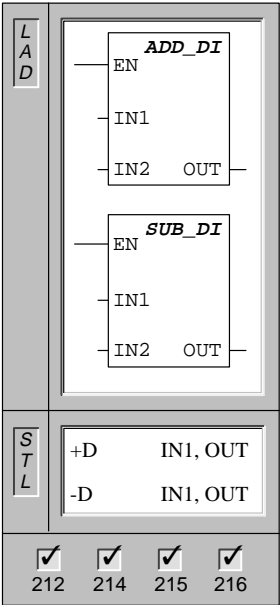
In STL:  $IN1 + OUT = OUT$   
 $OUT - IN1 = OUT$

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

### Add, Subtract Double Integer



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

Operands: IN1, IN2: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD

OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

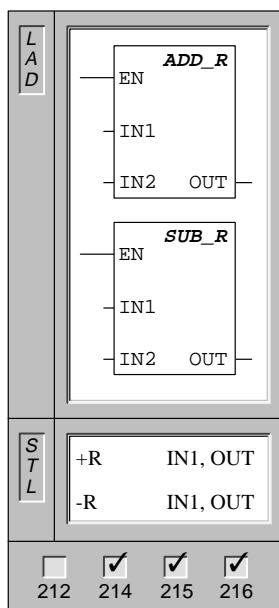
In LAD:  $IN1 + IN2 = OUT$   
 $IN1 - IN2 = OUT$

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

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

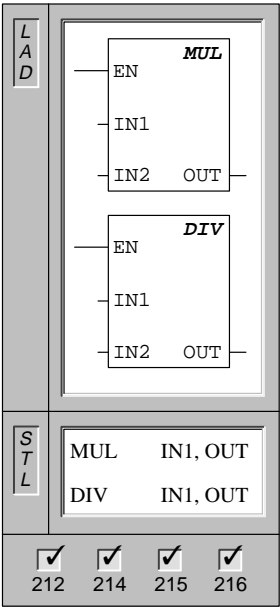


SM1.0 (zero); SM1.1 (overflow/illegal value); SM1.2 (negative)

### 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, Divide Integer



The **Multiply** instruction multiplies two 16-bit integers and produces a 32-bit product (OUT).

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

The **Divide** instruction divides two 16-bit integers and produces a 32-bit result (OUT). The 32-bit result (OUT) is comprised of a 16-bit quotient (least significant) and a 16-bit remainder (most significant).

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

Operands: IN1, IN2: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

In LAD:  $IN1 * IN2 = OUT$   
 $IN1 / IN2 = OUT$

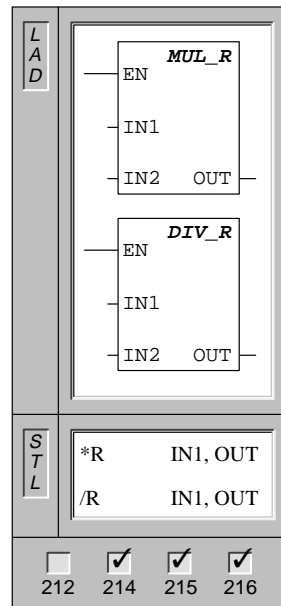
In STL:  $IN1 * OUT = OUT$   
 $OUT / IN1 = OUT$

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative); SM1.3 (divide-by-zero)

## Multiply, 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.

Operands: IN1, IN2: VD, ID, QD, MD, SMD, AC, Constant, \*VD, \*AC, SD

OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

In LAD:  $IN1 * IN2 = OUT$   
 $IN1 / IN2 = OUT$

In STL:  $IN1 * OUT = OUT$   
 $OUT / IN1 = OUT$

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

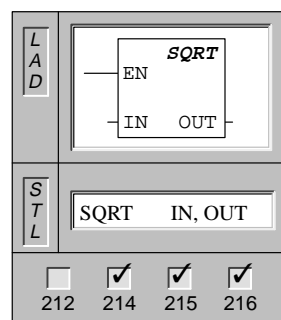
SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative);  
 SM1.3 (divide-by-zero)

If SM1.1 or SM1.3 are set, then the other math status bits are left unchanged and the original input operands are not altered.

### 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.

## Square Root



The **Square Root of Real Numbers** 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:

$$\sqrt{IN} = OUT$$

Operands: IN: VD, ID, QD, MD, SMD, AC, Constant, \*VD, \*AC, SD

OUT: VD, ID, QD, MD, SMD AC, \*VD, \*AC, SD

This instruction affects the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

Math Examples

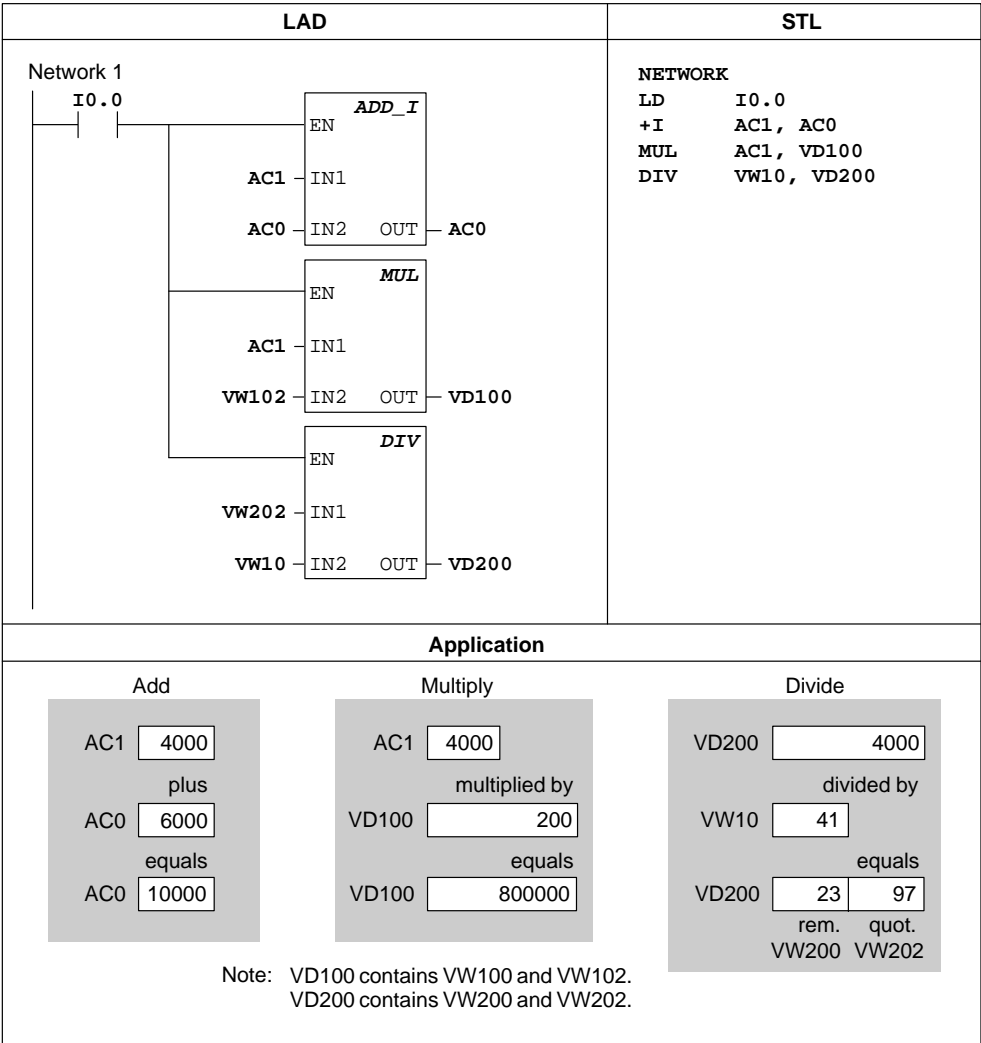
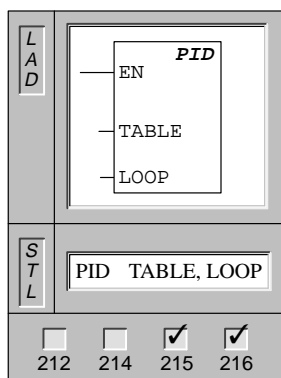


Figure 10-20 Examples of Math Instructions for LAD and STL

## PID Loop Control



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

Operands:     Table:     VB  
                      Loop:     0 to 7

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

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. Only 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.

## 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:

$M(t)$	=	$K_C * e$	+	$K_C \int_0^t e \, dt + M_{\text{initial}}$	+	$K_C * de/dt$
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)  
 $M_{\text{initial}}$    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      0                 // 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:

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

where:

$R_{\text{Norm}}$	is the normalized, real number value representation of the real world value
$R_{\text{Raw}}$	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:

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

where:

$R_{\text{Scal}}$	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:

```

TRUNC   AC0, AC0            // Convert the real number value 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:

$$MX = 1.0 - (MP_n + MD_n) \quad \text{when the calculated output, } M_n > 1.0$$

or

$$MX = - (MP_n + MD_n) \quad \text{when the calculated output, } M_n < 0.0$$

where:

MX	is the value of the adjusted bias
MP <sub>n</sub>	is the value of the proportional term of the loop output at sample time n
MD <sub>n</sub>	is the value of the differential term of the loop output at sample time n
M <sub>n</sub>	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<sub>n</sub>) 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<sub>n</sub>) = process variable (PV<sub>n</sub>)
- Sets old process variable (PV<sub>n-1</sub>) = process variable (PV<sub>n</sub>)
- Sets bias (MX) = output value (M<sub>n</sub>)

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.

## Alarming and Special Operations

The PID instruction is a simple but powerful instruction that performs the PID calculation. If other processing is required such as alarming 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 10-12:

Table 10-12 Format of the Loop Table

Offset	Field	Format	Type	Description
0	Process variable (PV <sub>n</sub> )	Double word - real	in	Contains the process variable, which must be scaled between 0.0 and 1.0.
4	Setpoint (SP <sub>n</sub> )	Double word - real	in	Contains the setpoint, which must be scaled between 0.0 and 1.0.
8	Output (M <sub>n</sub> )	Double word - real	in/out	Contains the calculated output, scaled between 0.0 and 1.0.
12	Gain (K <sub>C</sub> )	Double word - real	in	Contains the gain, which is a proportional constant. Can be a positive or negative number.
16	Sample time (T <sub>S</sub> )	Double word - real	in	Contains the sample time, in seconds. Must be a positive number.
20	Integral time or reset (T <sub>I</sub> )	Double word - real	in	Contains the integral time or reset, in minutes. Must be a positive number.
24	Derivative time or rate (T <sub>D</sub> )	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 <sub>n-1</sub> )	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 10-21 shows the control program for this application.

LAD	STL
<p>Network 1</p> <p>SM0.1 ———— 0     (CALL)</p> <p>Network 2</p> <p>————— (END)</p> <p>Network 3</p> <p>0   [SBR]</p> <p>Network 4</p> <p>SM0.0 ———— EN ———— MOV_R ———— OUT ———— VD104     0.75 IN</p> <p>0.25 ———— EN ———— MOV_R ———— OUT ———— VD112     0.25 IN</p> <p>0.10 ———— EN ———— MOV_R ———— OUT ———— VD116     0.10 IN</p> <p>30.0 ———— EN ———— MOV_R ———— OUT ———— VD120     30.0 IN</p> <p>0.0 ———— EN ———— MOV_R ———— OUT ———— VD124     0.0 IN</p> <p>100 ———— EN ———— MOV_B ———— OUT ———— SMB34     100 IN</p> <p>0 ———— EN ———— ATCH ———— INT ———— 0     10 ———— EVENT ———— 10</p> <p>————— (ENI)</p> <p>Network 5</p> <p>————— (RET)</p> <p>Network 6</p> <p>0   [INT]</p>	<p>Network 1</p> <p>LD SM0.1 //On the first scan call CALL 0 //the initialization //subroutine.</p> <p>Network 2</p> <p>MEND //End of the main program</p> <p>Network 3</p> <p>SBR 0</p> <p>Network 4</p> <p>LD SM0.0 MOVR 0.75, VD104 //Load the loop setpoint. // = 75% full. MOVR 0.25, VD112 //Load the loop gain=0.25. MOVR 0.10, VD116 //Load the loop sample //time = 0.1 seconds. MOVR 30.0, VD120 //Load the integral time //= 30 minutes. // MOVR 0.0, VD124 //Set no derivative action. MOVB 100, SMB34 //Set time interval //(100 ms) for timed //interrupt 0. ATCH 0, 10 //Set up a timed //interrupt to invoke //PID execution. ENI //Enable interrupts.</p> <p>NETWORK 5</p> <p>RET</p> <p>NETWORK 6</p> <p>INT 0 //PID Calculation //Interrupt Routine</p>

(This figure continues on the next page.)

Figure 10-21 Example of PID Loop Control



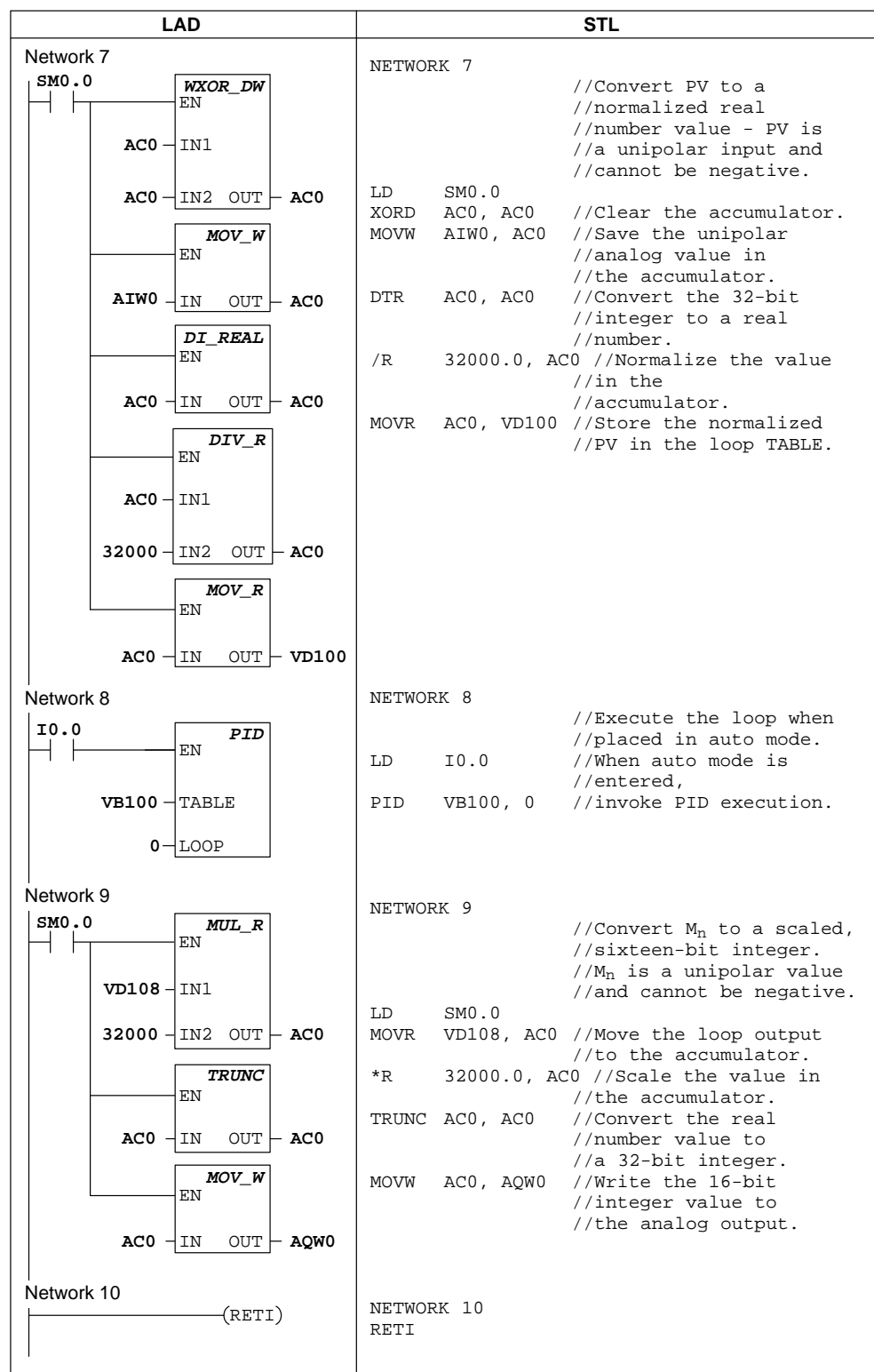
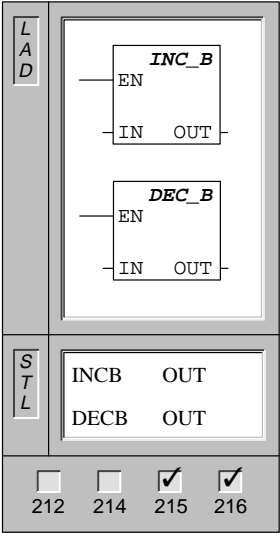


Figure 10-21 Example of PID Loop Control (continued)

10.7 Increment and Decrement Instructions

Increment Byte, Decrement Byte



The **Increment Byte** and **Decrement Byte** instructions add or subtract 1 to or from the input byte.

Operands: IN: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
OUT: VB, IB, QB, MB, SMB, AC, \*VD, \*AC, SB

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

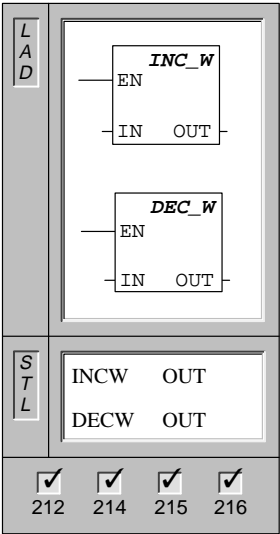
In STL: OUT + 1 = OUT  
OUT - 1 = OUT

Increment and decrement byte operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

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

Increment Word, Decrement Word



The **Increment Word** and **Decrement Word** instructions add or subtract 1 to or from the input word.

Operands: IN: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW  
OUT: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

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

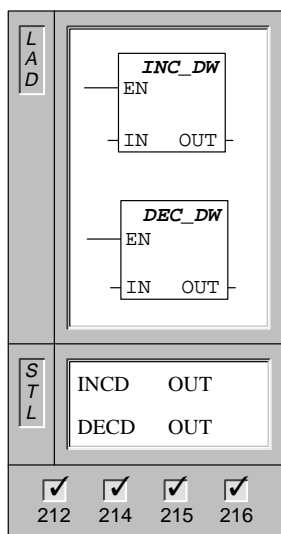
In STL: OUT + 1 = OUT  
OUT - 1 = OUT

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

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

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

# Increment Double Word, Decrement Double Word



The **Increment Double Word** and **Decrement Double Word** instructions add or subtract 1 to or from the input double word.

Operands: IN: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD

OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

In LAD:  $IN + 1 = OUT$   
 $IN - 1 = OUT$

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

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

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow); SM1.2 (negative)

## Increment, Decrement Example

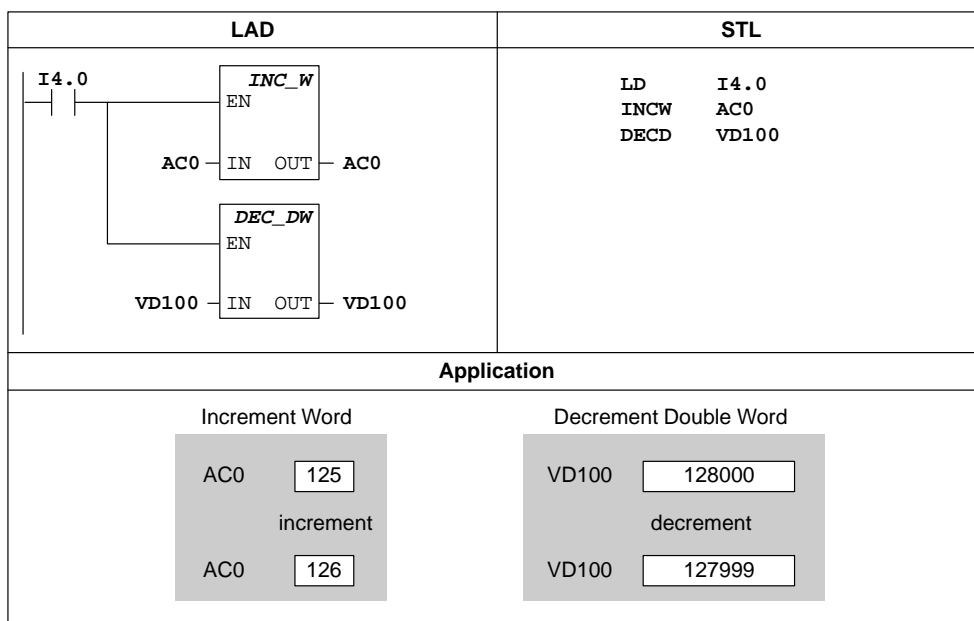
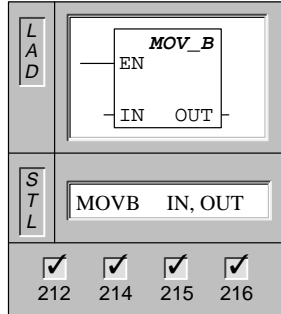


Figure 10-22 Example of Increment/Decrement Instructions for LAD and STL

## 10.8 Move, Fill, and Table Instructions

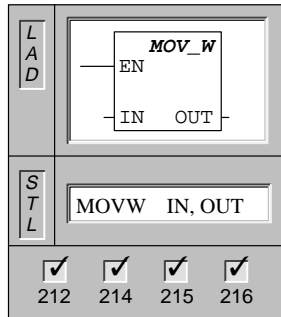
### Move Byte



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

Operands: IN: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
OUT: VB, IB, QB, MB, SMB, AC, \*VD, \*AC, SB

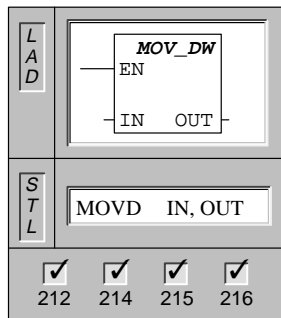
### Move Word



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

Operands: IN: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW  
OUT: VW, T, C, IW, QW, MW, SMW, AC, AQW, \*VD, \*AC, SW

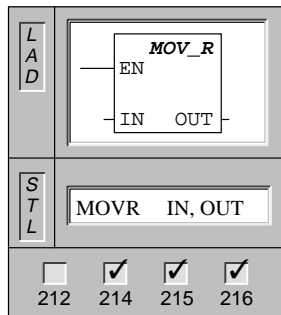
### Move Double Word



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.

Operands: IN: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, &VB, &IB, &QB, &MB, &T, &C, &SB, SD  
OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

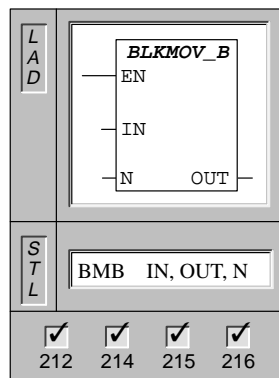
### Move Real



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.

Operands: IN: VD, ID, QD, MD, SMD, AC, Constant, \*VD, \*AC, SD  
OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

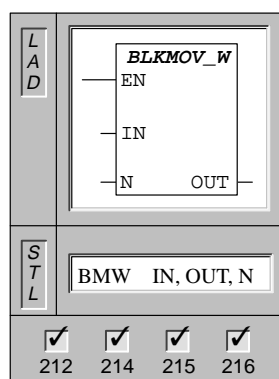
### Block Move Byte



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

Operands: IN, OUT: VB, IB, QB, MB, SMB, \*VD, \*AC, SB  
N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

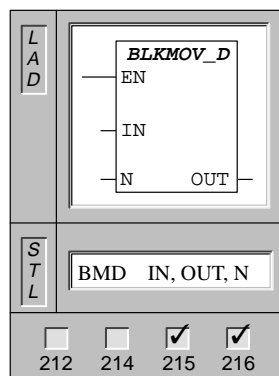
### Block Move Word



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

Operands: IN: VW, T, C, IW, QW, MW, SMW, AIW, \*VD, \*AC, SW  
OUT: VW, T, C, IW, QW, MW, SMW, AQW, \*VD, \*AC, SW  
N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

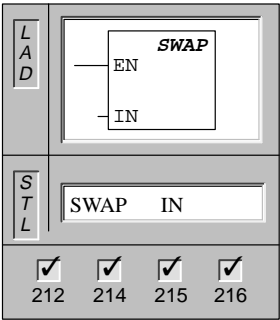
### Block Move Double Word



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

Operands: IN, OUT: VD, ID, QD, MD, SMD, \*VD, \*AC, SD  
N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

Swap Bytes



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

Operands: IN: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

Move and Swap Examples

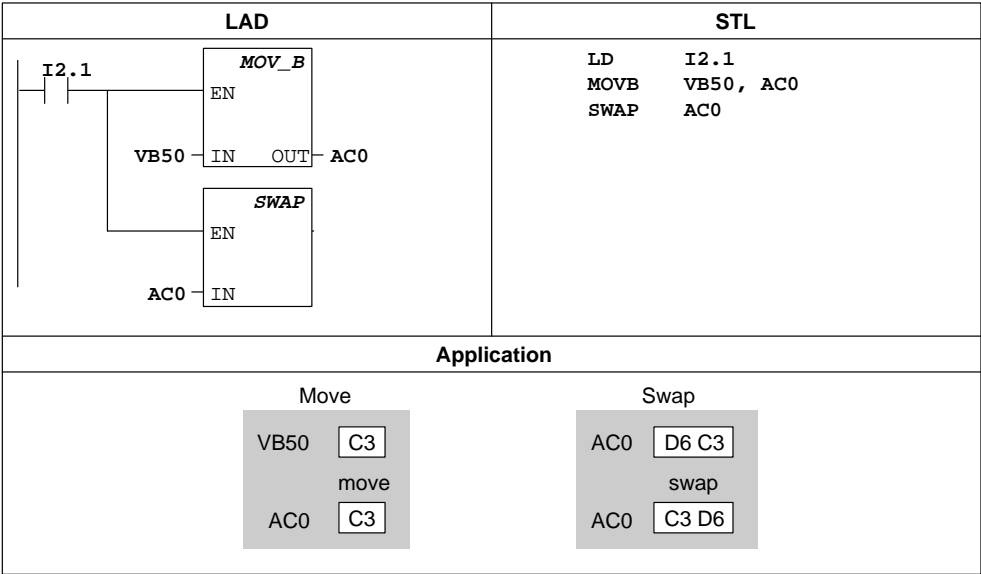


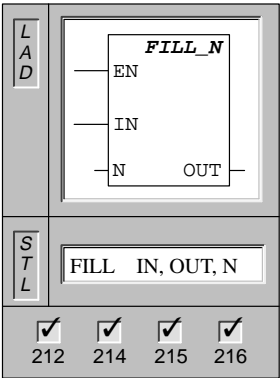
Figure 10-23 Example of Move and Swap Instructions for LAD and STL

### Block Move Example

LAD	STL
<div><div><div><div><div>I2.1</div><div></div></div><div></div></div><div><div><div>BLKMOV_B</div><div>EN</div><div>IN</div><div>N</div><div>OUT</div></div><div>VB20</div><div>4</div><div>VB100</div></div></div><div>Move Array 1 (VB20 to VB23) to Array 2 (VB100 to VB103)</div></div>	<div><div>LD</div><div>I2.1</div><div>BMB</div><div>VB20, VB100, 4</div></div>
Application	
<div><div><div>Array 1</div><div><div>VB20</div><div>30</div></div><div><div>VB21</div><div>31</div></div><div><div>VB22</div><div>32</div></div><div><div>VB23</div><div>33</div></div></div><div>block move</div><div><div>Array 2</div><div><div>VB100</div><div>30</div></div><div><div>VB101</div><div>31</div></div><div><div>VB102</div><div>32</div></div><div><div>VB103</div><div>33</div></div></div></div>	

Figure 10-24 Example of Block Move Instructions for LAD and STL

Memory Fill



The **Memory Fill** instruction fills the 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.

- Operands:
- IN:** VW, T, C, IW, QW, MW, SMW, AIW, Constant, \*VD, \*AC, SW
  - OUT:** VW, T, C, IW, QW, MW, SMW, AQW, \*VD, \*AC, SW
  - N:** VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

Fill Example

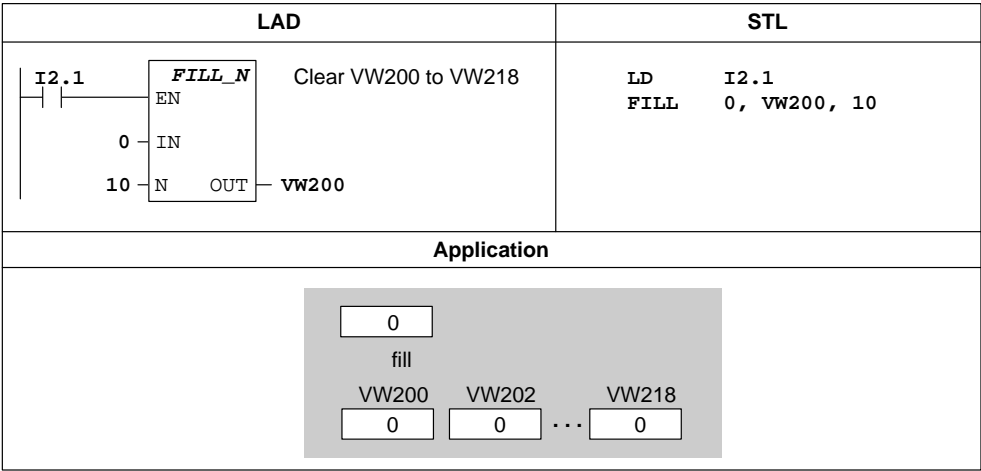
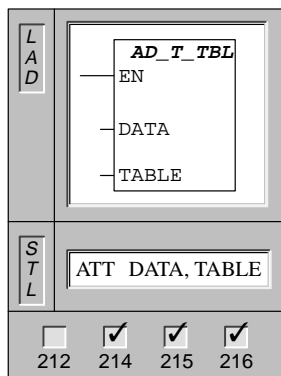


Figure 10-25 Example of Fill Instructions for LAD and STL



## Add to Table



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

Operands: DATA: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW  
TABLE: VW, T, C, IW, QW, MW, SMW, \*VD, \*AC, SW

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 10-26.) 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 entries, excluding both parameters specifying the maximum number of entries and the actual number of entries.

This instruction affects the following Special Memory bits:

SM1.4 is set to 1 if you try to overfill the table.

## Add to Table Example

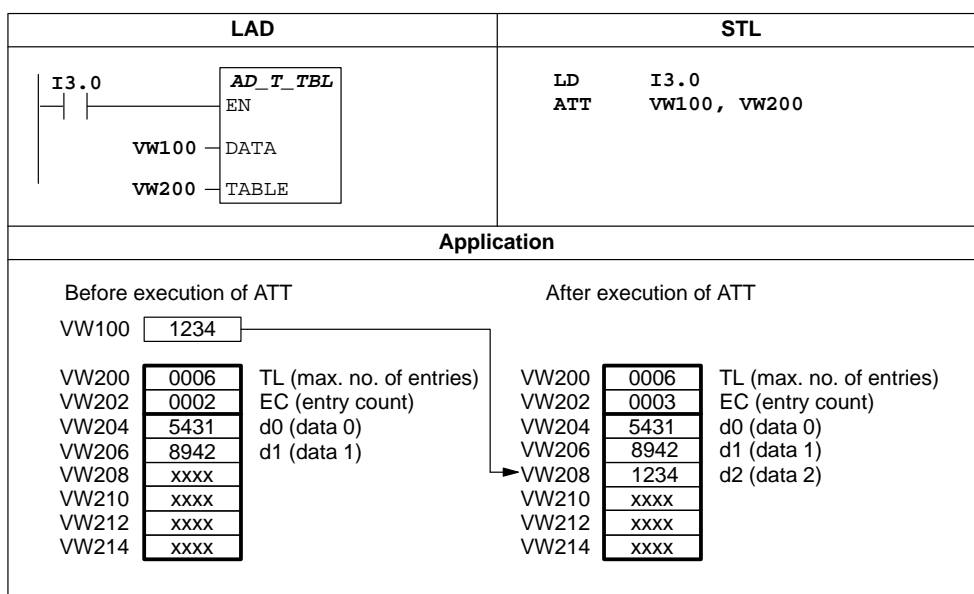
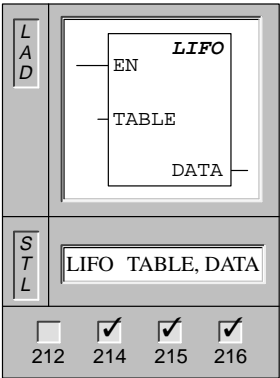


Figure 10-26 Example of Add To Table Instruction

Last-In-First-Out



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

Operands:     TABLE:    VW, T, C, IW, QW, MW, SMW, \*VD, \*AC, SW  
                 DATA:    VW, T, C, IW, QW, MW, SMW, AC, AQW, \*VD, \*AC, SW

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.

Last-In-First-Out Example

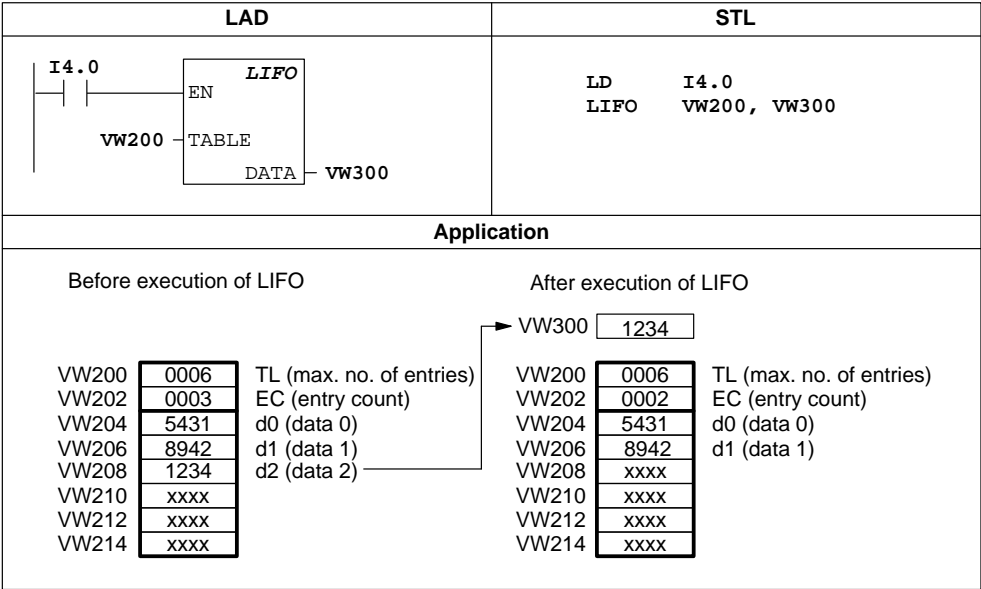
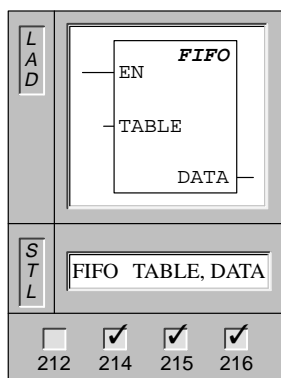


Figure 10-27    Example of Last-In-First-Out Instruction

## First-In-First-Out



The **First-In-First-Out** instruction removes the first entry in the table (TABLE), 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.

Operands:      TABLE:    VW, T, C, IW, QW, MW, SMW, \*VD, \*AC, SW  
                          DATA:    VW, T, C, IW, QW, MW, SMW, AC, AQW, \*VD, \*AC, SW

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.

## First-In-First-Out Example

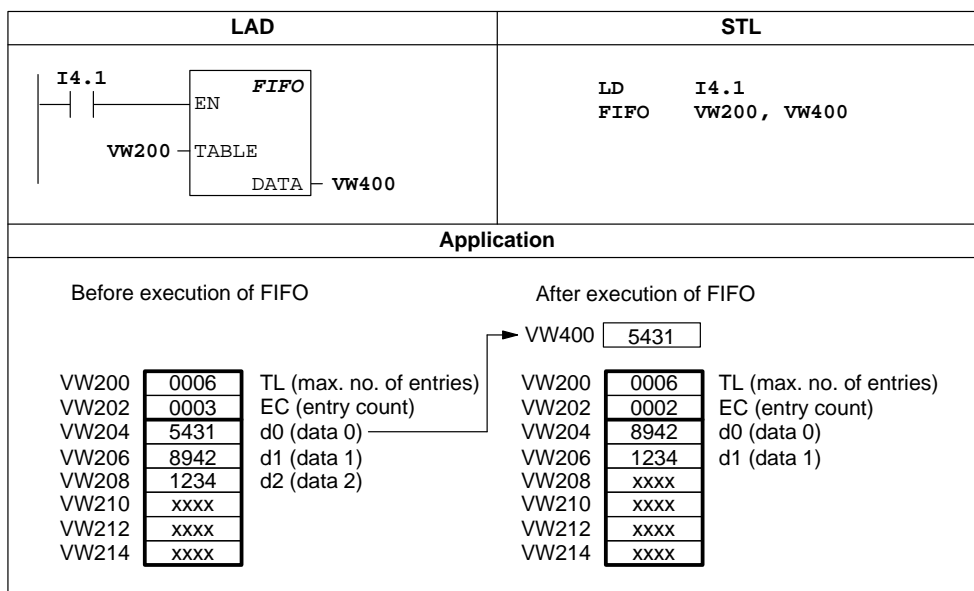
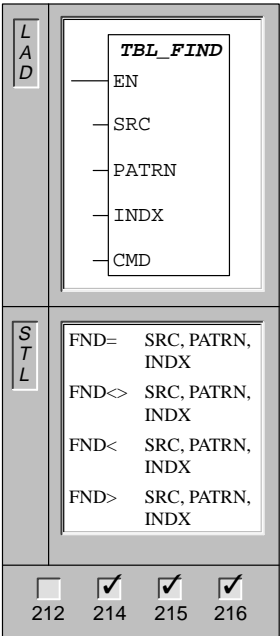


Figure 10-28 Example of First-In-First-Out Instruction

Table Find



The **Table Find** instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the search criteria of =, <>, <, or >.

In LAD, the command parameter (CMD) is given a numeric value of 1 to 4 that corresponds to =, <>, <, and >, respectively.

- Operands:
- SRC: VW, T, C, IW, QW, MW, SMW, \*VD, \*AC, SW
  - PATRN: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW
  - INDX: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW
  - CMD: 1 (=) 2 (<>) 3 (<) 4 (>)

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.

The data entries (area to be searched) are numbered from 0 to a maximum value of 99. A table may have up to 100 entries, excluding both the parameters specifying the allowed number of entries and the actual number of entries.

**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 TABLE operand of a corresponding ATT, LIFO, or FIFO instruction, as shown in Figure 10-29.

Table format for ATT, LIFO, and FIFO				Table format for TBL_FIND			
VW200	0006	TL (max. no. of entries)		VW202	0006	EC (entry count)	
VW202	0006	EC (entry count)		VW204	xxxx	d0 (data 0)	
VW204	xxxx	d0 (data 0)		VW206	xxxx	d1 (data 1)	
VW206	xxxx	d1 (data 1)		VW208	xxxx	d2 (data 2)	
VW208	xxxx	d2 (data 2)		VW210	xxxx	d3 (data 3)	
VW210	xxxx	d3 (data 3)		VW212	xxxx	d4 (data 4)	
VW212	xxxx	d4 (data 4)		VW214	xxxx	d5 (data 5)	
VW214	xxxx	d5 (data 5)					

Figure 10-29 Difference in Table Format between Find Instructions and ATT, LIFO, and FIFO

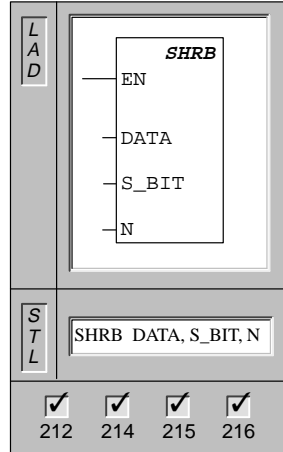
# Table Find Example

LAD		STL														
<div><div><div>I2.1</div><div>EN</div><div>VW202</div><div>16#3130</div><div>AC1</div><div>1</div></div><div>TBL_FIND</div><div>SRC</div><div>PATRN</div><div>INDX</div><div>CMD</div></div> <div>When I2.1 is on, search the table for a value equal to 3130 HEX.</div>	<div><div>LD</div><div>I2.1</div><div>FND=</div><div>VW202, 16#3130, AC1</div></div>															
Application																
<p>This is the table you are searching. If the table was created using ATT, LIFO, and FIFO instructions, VW200 contains the maximum number of allowed entries and is not required by the Find instructions.</p>																
	<table><tr><td>VW202</td><td>0006</td></tr><tr><td>VW204</td><td>3133</td></tr><tr><td>VW206</td><td>4142</td></tr><tr><td>VW208</td><td>3130</td></tr><tr><td>VW210</td><td>3030</td></tr><tr><td>VW212</td><td>3130</td></tr><tr><td>VW214</td><td>4541</td></tr></table>	VW202	0006	VW204	3133	VW206	4142	VW208	3130	VW210	3030	VW212	3130	VW214	4541	<div>EC (entry count)</div> <div>d0 (data 0)</div> <div>d1 (data 1)</div> <div>d2 (data 2)</div> <div>d3 (data 3)</div> <div>d4 (data 4)</div> <div>d5 (data 5)</div>
VW202	0006															
VW204	3133															
VW206	4142															
VW208	3130															
VW210	3030															
VW212	3130															
VW214	4541															
AC1	<div>0</div>	AC1 must be set to 0 to search from the top of table.														
Execute table search																
AC1	<div>2</div>	AC1 contains the data entry number corresponding to the first match found in the table (d2).														
AC1	<div>3</div>	Increment the INDX by one, before searching the remaining entries of the table.														
Execute table search																
AC1	<div>4</div>	AC1 contains the data entry number corresponding to the second match found in the table (d4).														
AC1	<div>5</div>	Increment the INDX by one, before searching the remaining entries of the table.														
Execute table search																
AC1	<div>6</div>	AC1 contains a value equal to the entry count. The entire table has been searched without finding another match.														
AC1	<div>0</div>	Before the table can be searched again, the INDX value must be reset to 0.														

Figure 10-30 Example of Find Instructions for LAD and STL

## 10.9 Shift and Rotate Instructions

### Shift Register Bit



The **Shift Register Bit** 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).

Operands: DATA, S\_BIT: I, Q, M, SM, T, C, V, S  
N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

### Understanding the Shift Register Bit Instruction

The Shift Register Bit instruction provides an easy method for the sequencing and controlling of 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 10-32 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:

$$\text{MSB.b} = [(\text{Byte of S\_BIT}) + ((N) - 1 + (\text{bit of S\_BIT})) / 8] . [\text{remainder of the division by 8}]$$

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:

$$\begin{aligned} \text{MSB.b} &= \text{V33} + ((14) - 1 + 4) / 8 \\ &= \text{V33} + 17 / 8 \\ &= \text{V33} + 2 \text{ with a remainder of } 1 \\ &= \text{V35.1} \end{aligned}$$

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 10-31 shows bit shifting for negative and positive values of N.

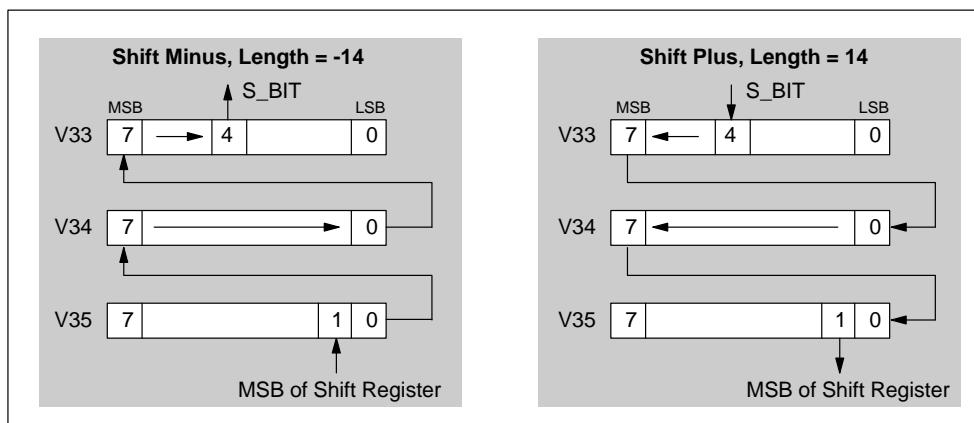


Figure 10-31 Shift Register Entry and Exit for Plus and Minus Shifts

### Shift Register Bit Example

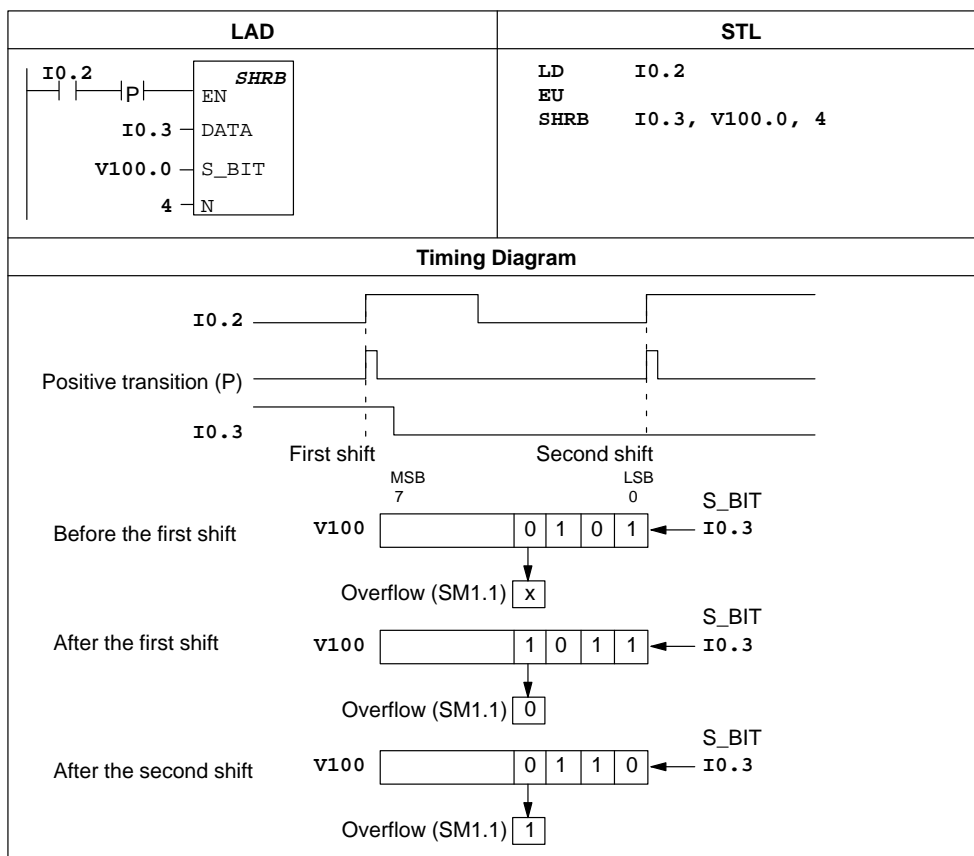
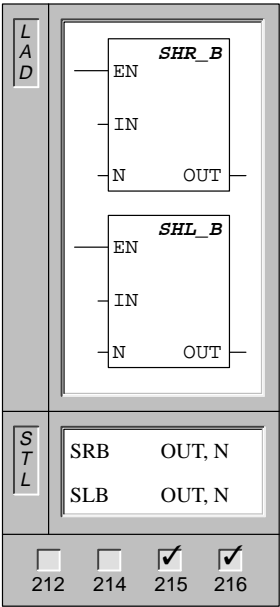


Figure 10-32 Example of Bit Shift Register Instruction for LAD and STL

Shift Right Byte, Shift Left Byte



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

Operands:    IN:        VB, IB, QB, MB, SMB, SB, AC, \*VD, \*AC  
                 N:        VB, IB, QB, MB, SMB, SB, AC, Constant, \*VD, \*AC  
                 OUT:      VB, IB, QB, MB, SMB, SB, AC, \*VD, \*AC

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 takes on the value of the last bit shifted out.

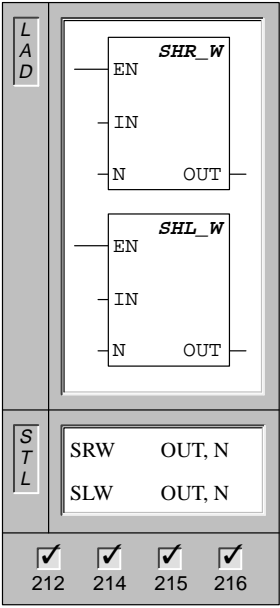
Shift right and shift left byte operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)

Shift Right Word, Shift Left Word



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

Operands:    IN:        VW, T, C, IW, MW, SMW, AC, QW, AIW, Constant, \*VD, \*AC, SW  
                 N:        VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
                 OUT:      VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

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 zero, the overflow memory bit takes on the value of the last bit shifted out.

Shift right and shift left word operations are unsigned.

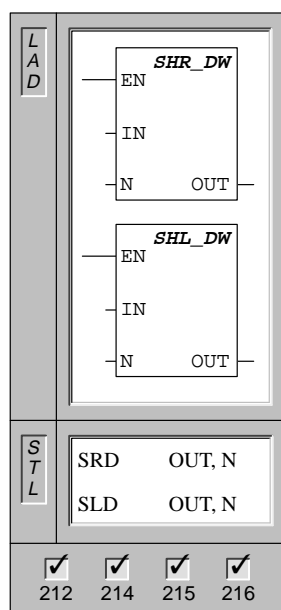
Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)



## 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 right or left by the shift count (N), and load the result in the output double word (OUT).

Operands:    IN:        VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD  
                   N:        VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
                   OUT:    VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

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 takes on the value of the last bit shifted out.

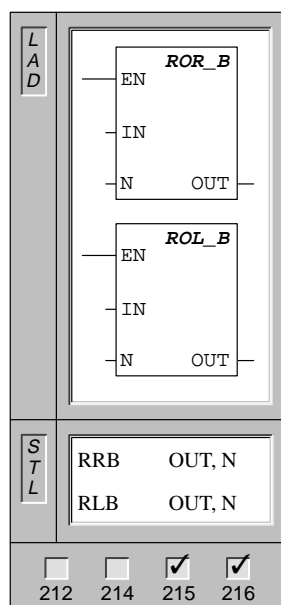
Shift right and shift left double word operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)

## Rotate Right Byte, Rotate Left Byte



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

Operands:    IN:        VB, IB, QB, MB, SMB, SB, AC, \*VD, \*AC, SB  
                   N:        VB, IB, QB, MB, SMB, SB, AC, Constant, \*VD, \*AC, SB  
                   OUT:    VB, IB, QB, MB, SMB, SB, AC, \*VD, \*AC, SB

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 rotate 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.

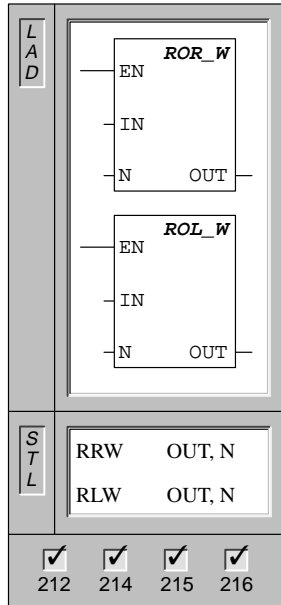
Rotate right and rotate left byte operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)

### Rotate Right Word, Rotate Left Word



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

Operands: IN: VW, T, C, IW, MW, SMW, AC, QW, AIW, Constant, \*VD, \*AC, SW  
 N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
 OUT: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

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.

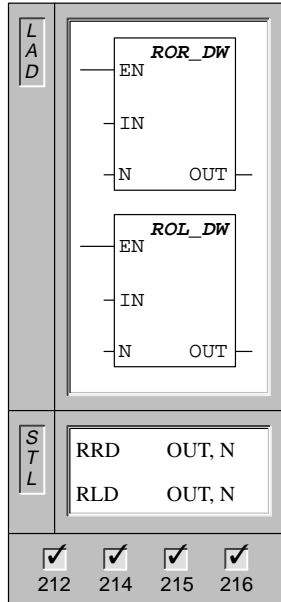
Rotate Right and Rotate Left Word operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)

### 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 right or left by the shift count (N), and load the result in the output double word (OUT).

Operands: IN: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD  
 N: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
 OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

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.

Rotate Right and Rotate Left Double-Word operations are unsigned.

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero); SM1.1 (overflow)

# Shift and Rotate Examples

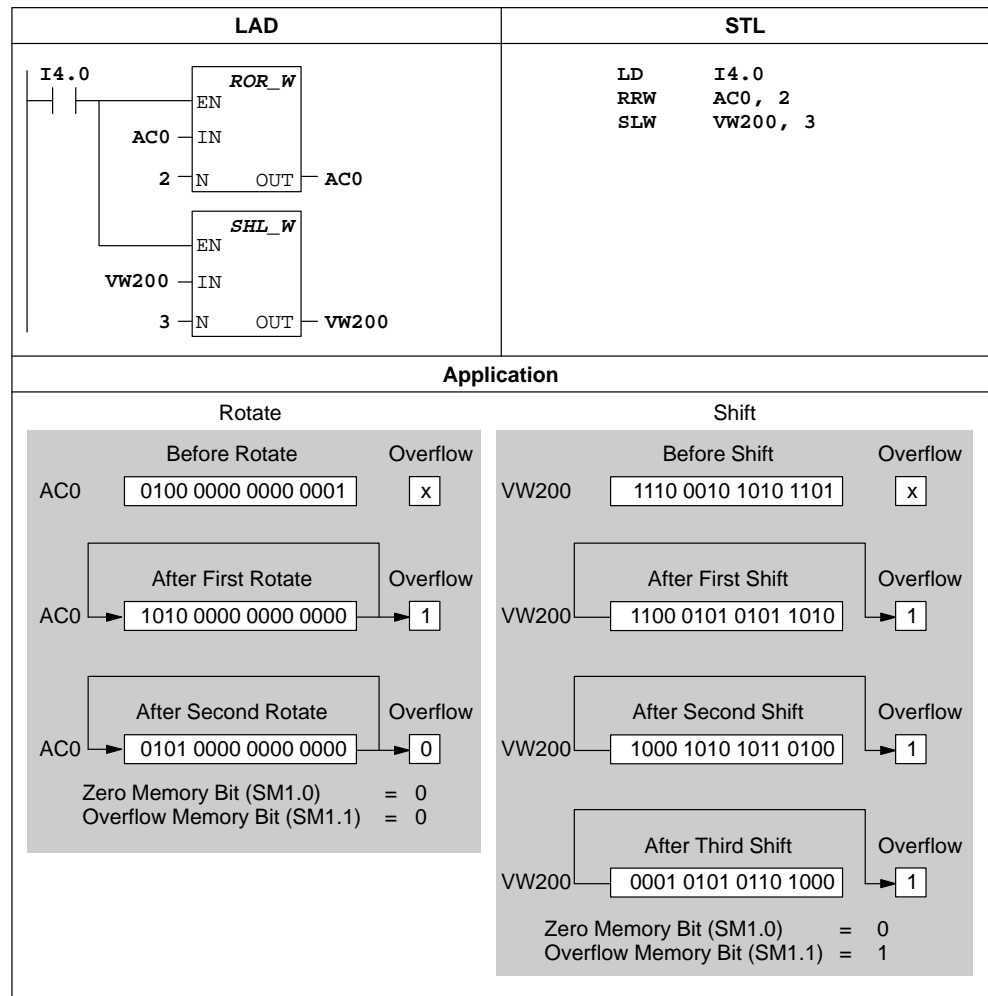
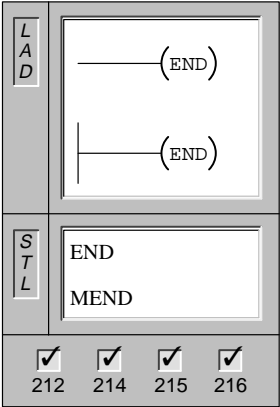


Figure 10-33 Example of Shift and Rotate Instructions for LAD and STL

## 10.10 Program Control Instructions

### End



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

The **Unconditional END** coil must be used to terminate the main user program.

In STL, the unconditional END operation is represented by the **MEND** instruction.

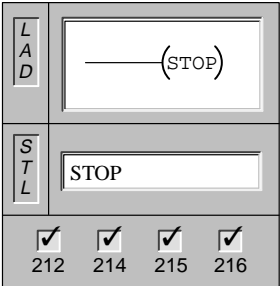
Operands:     None

All user programs must terminate the main program with an unconditional END instruction. The conditional END instruction is used to terminate execution before encountering the unconditional END instruction.

#### Note

You can use the Conditional END and Unconditional END instructions in the main program, but you cannot use them in either subroutines or interrupt routines.

### Stop

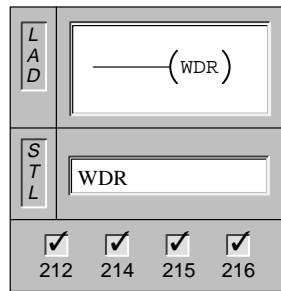


The **STOP** instruction terminates the execution of your program immediately by causing a transition of the CPU from RUN to STOP mode.

Operands:     None

If the STOP instruction is executed in an interrupt routine, the interrupt routine is terminated immediately, and all pending interrupts are ignored. The rest of the program is scanned 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 terminated:

- 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 assume the STOP mode within 1.4 seconds.

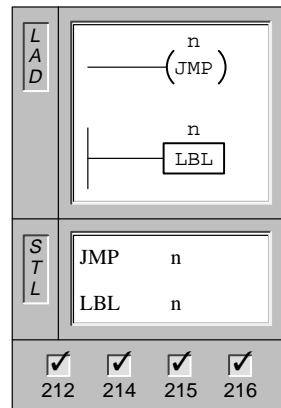
---

Stop, End, and WDR Example

LAD	STL
<p>Network 1</p> <p>When an I/O error is detected, force the transition to STOP mode.</p> <p>...</p> <p>Network 15</p> <p>When M5.6 is on, retrigger the Watchdog Reset (WDR) to allow the scan time to be extended.</p> <p>...</p> <p>Network 78</p> <p>Terminate the main program.</p>	<p>Network</p> <p>LD SM5.0</p> <p>STOP</p> <p>.</p> <p>.</p> <p>Network</p> <p>LD M5.6</p> <p>WDR</p> <p>.</p> <p>.</p> <p>Network</p> <p>MEND</p>

Figure 10-34 Example of Stop, End, and WDR Instructions for LAD and STL

## Jump to Label, and 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

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.

Figure 10-35 shows an example of the Jump to Label and Label instructions.

## Jump to Label Example

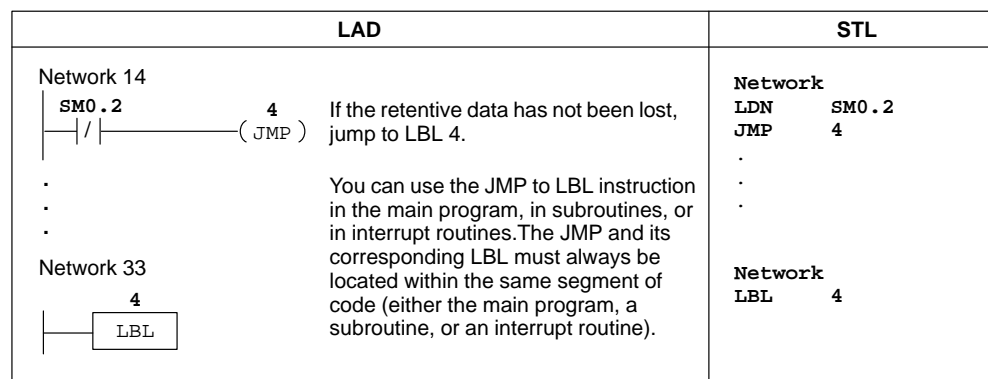
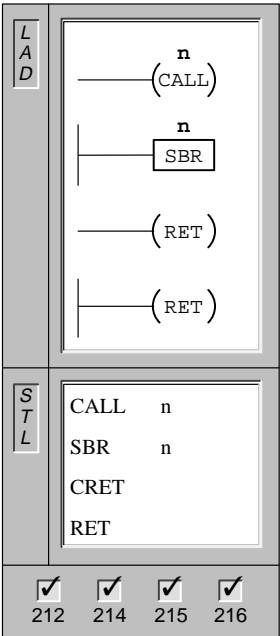


Figure 10-35 Example of Jump to Label and Label Instructions for LAD and STL

Call, Subroutine, and Return from Subroutine



The **Call** instruction transfers control to the subroutine (n).

The **Subroutine** instruction marks the beginning of the subroutine (n).

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

The **Unconditional Return from Subroutine** instruction must be used to terminate each subroutine.

Operands: n: 0 to 63

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

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.

Also when a subroutine is called, the top of stack value is always a logical 1. Therefore, you can connect outputs or boxes directly to the left power rail for the network following the SBR instruction. In STL, the Load instruction can be omitted following the SBR instruction.

Accumulators are passed freely among the main program and subroutines. No save or restore operation is performed on accumulators due to subroutine use.

Figure 10-36 shows an example of the Call, Subroutine, and Return from Subroutine instructions.

Restrictions

Restrictions for using subroutines follow:

- Put all subroutines after the end of the main ladder program.
- You cannot use the LSCR, SCRE, SCRT, and END instructions in a subroutine.
- You must terminate each interrupt routine by an unconditional return from subroutine instruction (RET).



# Call to Subroutine Example



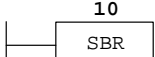


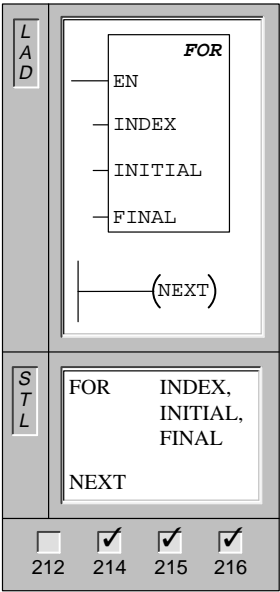
LAD		STL
Network 1	 <p>On the first scan: Call SBR 10 for initialization.</p>	<b>Network</b> LD SM0.1 CALL 10 .
Network 39	 <p>You must locate all subroutines after the END instruction.</p>	<b>Network</b> MEND .
Network 50	 <p>Start of Subroutine 10</p>	<b>Network</b> SBR 10 .
Network 65	 <p>A conditional return (RET) from Subroutine 10 may be used.</p>	<b>Network</b> LD M14.3 CRET .
Network 68	 <p>Each subroutine must be terminated by an unconditional return (RET). This terminates Subroutine 10.</p>	<b>Network</b> RET

Figure 10-36 Example of Subroutine Instructions for LAD and STL

For, Next



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

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

Operands: INDEX: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

INITIAL: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

FINAL: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

For example, given an INITIAL value of 1 and a FINAL value of 10, the instructions between the FOR and the NEXT are executed 10 times with the INDEX 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 INDEX value is incremented and the result is compared to the final value. If the INDEX is greater than the final value, the loop is terminated.

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.

Figure 10-37 shows an example of the FOR/NEXT instructions.

# For/Next Example

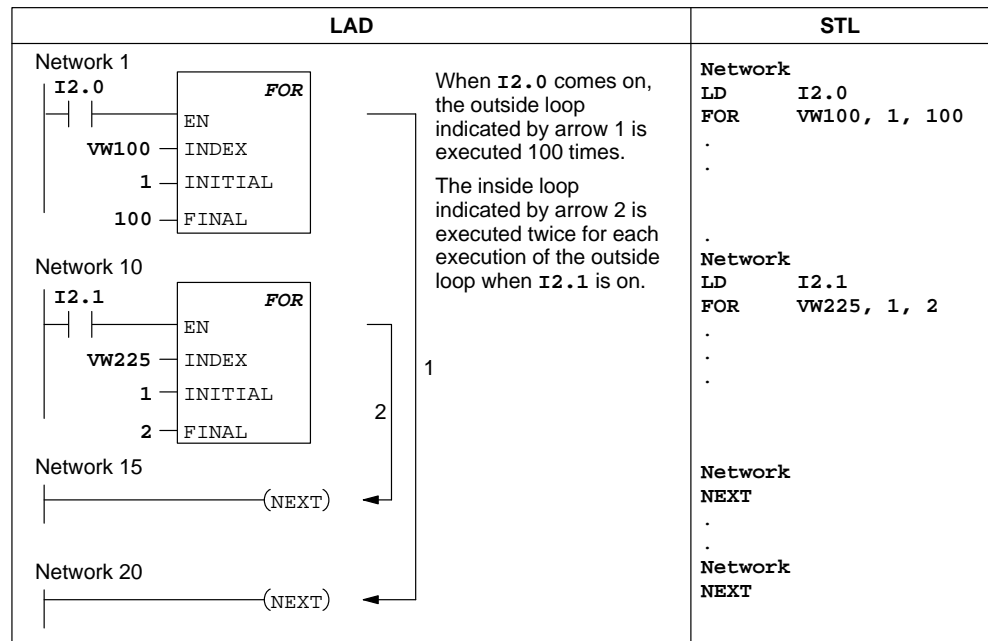
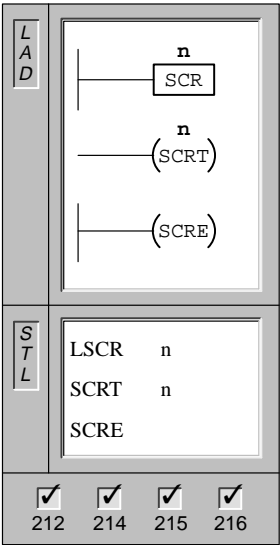


Figure 10-37 Example of For/Next Instructions for LAD and STL

Sequence Control Relay Instructions



The **Load Sequence Control Relay** instruction marks the beginning of an SCR segment. When n = 1, 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, 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.

Operands:      n:            S

Understanding SCR Instructions

In ladder logic and statement list, 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 10-38 shows the S stack and the logic stack and the effect of executing the LSCR instruction.

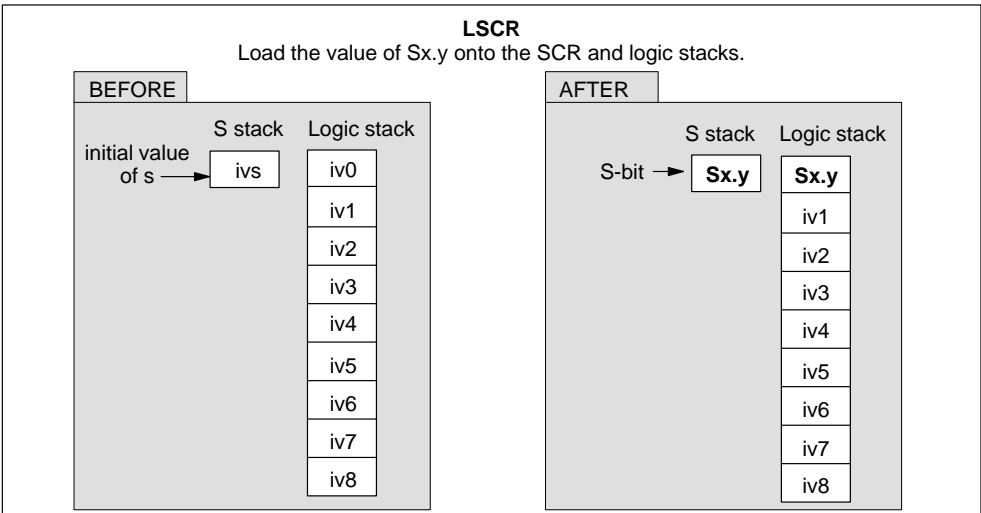


Figure 10-38 Effect of LSCR on the Logic Stack

The following is true of Segmentation 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 can use SCRs in the main program, but you cannot use them in subroutines and interrupt routines.
- 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 10-39 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.

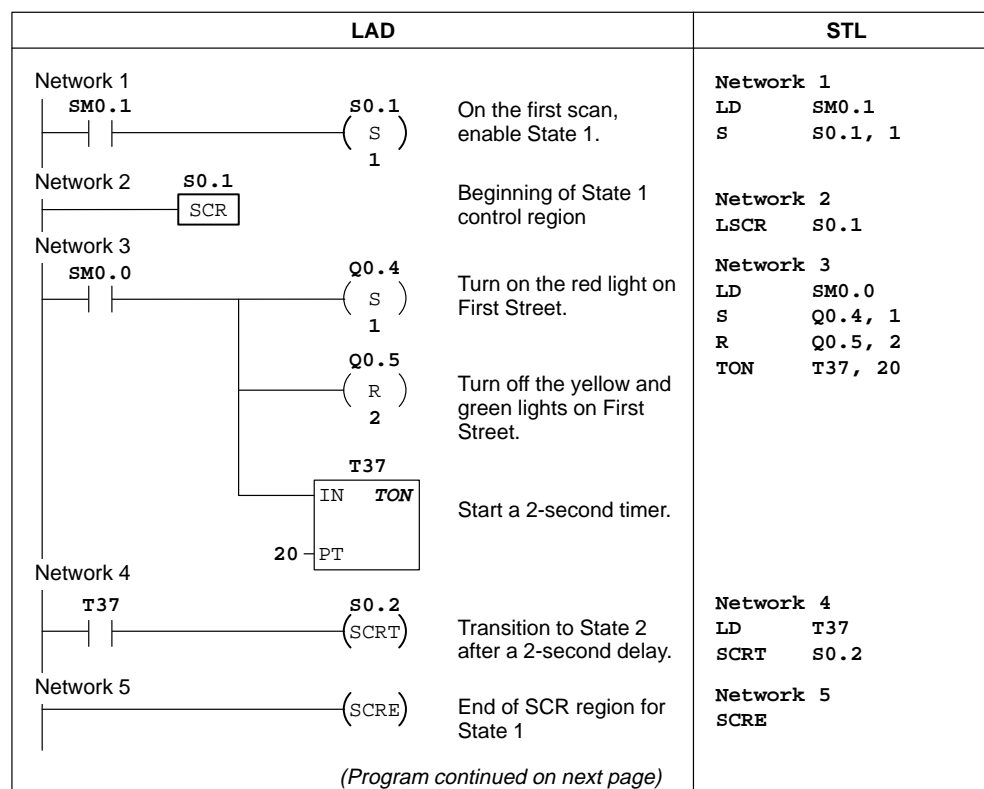


Figure 10-39 Example of Sequence Control Relays (SCRs)

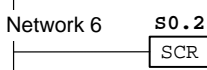
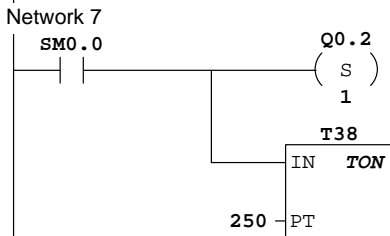
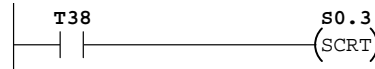
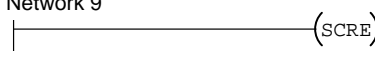
LAD	STL
(Program continued from previous page)	
Network 6 	Network 6 LSCR S0.2
Network 7 	Network 7 LD SM0.0 S Q0.2, 1 TON T38, 250
Network 8 	Network 8 LD T38 SCRT S0.3
Network 9 	Network 9 SCRE
...	...

Figure 10-39 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 10-40.

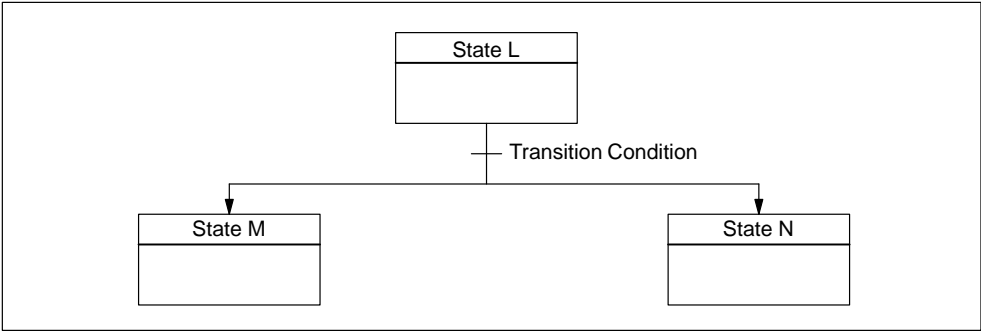


Figure 10-40 Divergence of Control Stream

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 10-41.

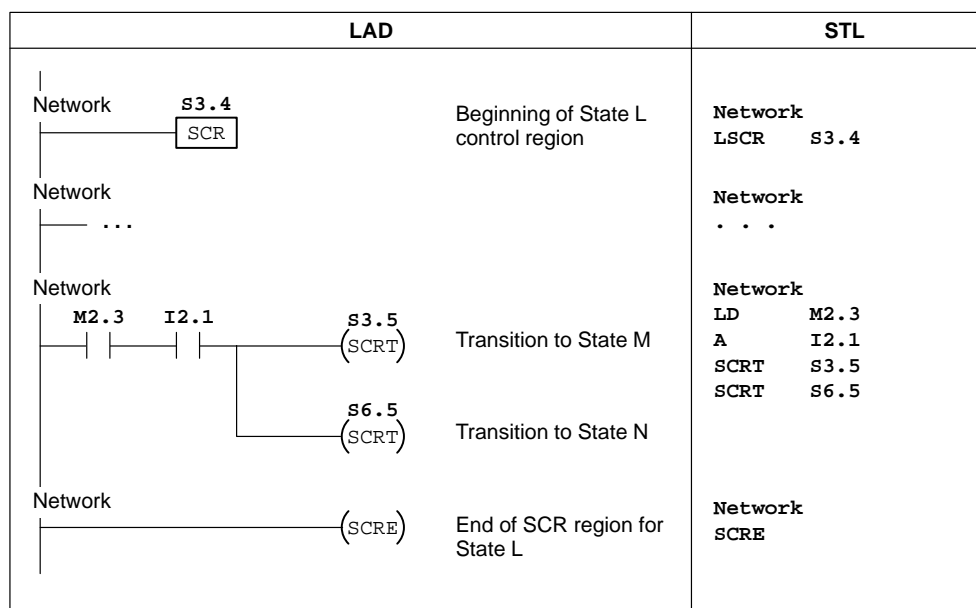


Figure 10-41 Example of Divergence of Control Streams

### 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 10-42 depicts the convergence of two control streams.

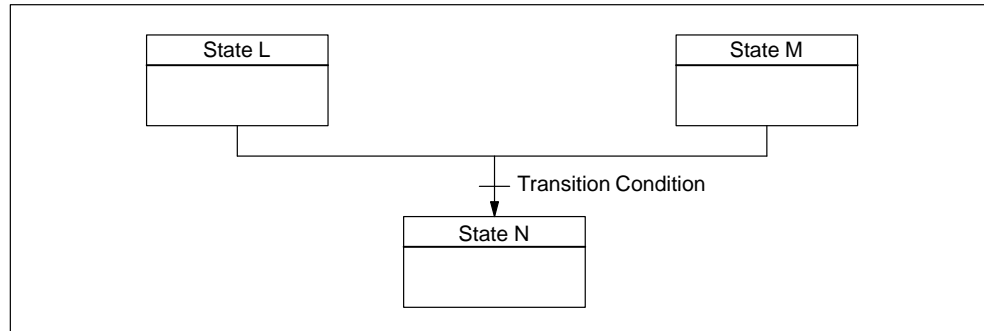


Figure 10-42 Convergence of Control Streams



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 be enabled as shown below.

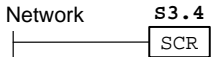


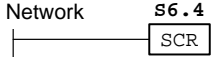
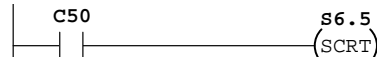

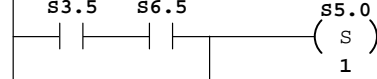
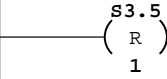
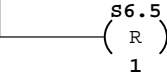
LAD		STL
Network		Beginning of State L control region
Network	...	
Network		Transition to State L'
Network		End of SCR region for State L
Network		Beginning of State M control region
Network	...	
Network		Transition to State M'
Network		End of SCR region for State M
Network		Enable State N
		Reset State L'
		Reset State M'

Figure 10-43 Example of Convergence of Control Streams

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 10-44.

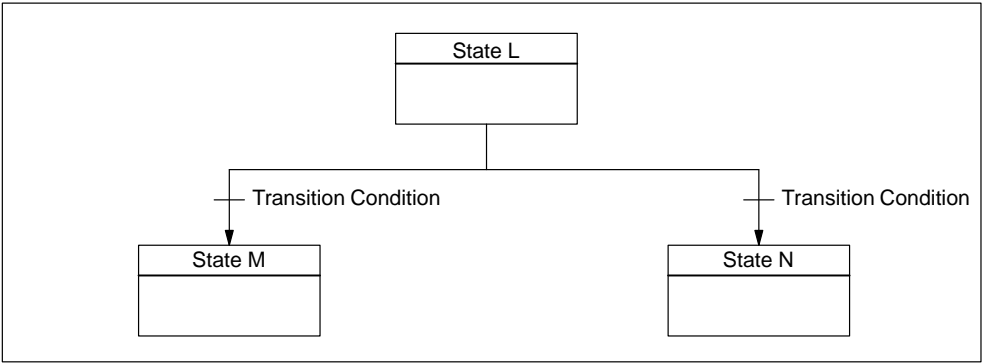


Figure 10-44 Divergence of Control Stream, Depending on Transition Condition

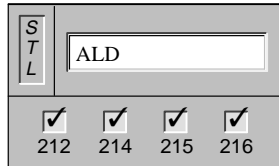
An equivalent SCR program is shown in Figure 10-45.

LAD		STL
Network	<div>s3.4</div> <div>SCR</div>	Network LSCR s3.4
Network	...	Network ...
Network	<div>M2.3</div> <div>(s3.5)</div> <div>(SCRT)</div>	Network LD M2.3 SCRT s3.5
Network	<div>I3.3</div> <div>(s6.5)</div> <div>(SCRT)</div>	Network LD I3.3 SCRT s6.5
Network	<div>(scre)</div>	Network SCRE

Figure 10-45 Example of Conditional Transitions

## 10.11 Logic Stack Instructions

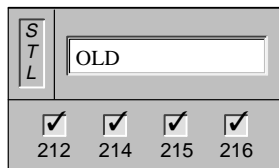
### And Load



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

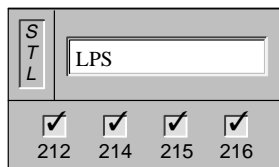
### Or Load



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

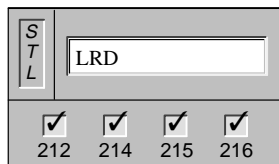
### Logic Push



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

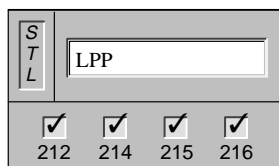
### Logic Read



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

Logic Stack Operations

Figure 10-46 illustrates the operation of the And Load and Or Load instructions.

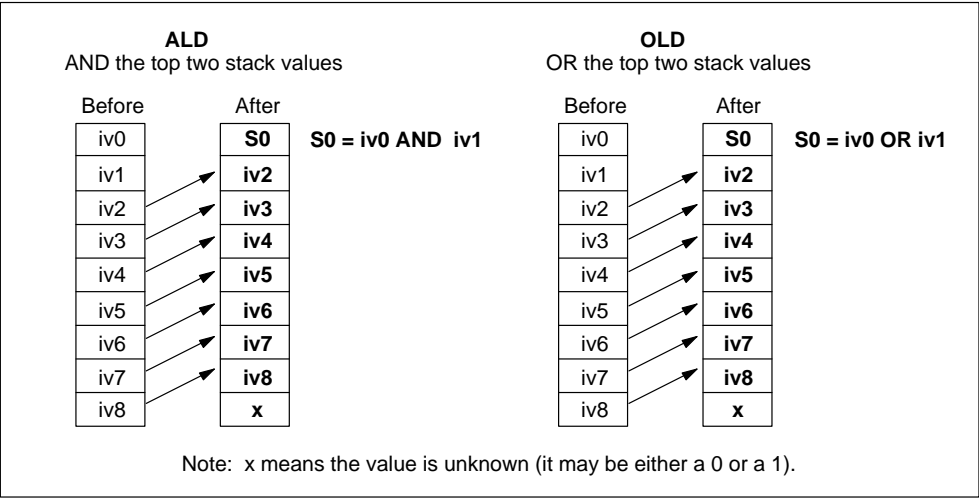


Figure 10-46 And Load and Or Load Instructions

Figure 10-47 illustrates the operation of the Logic Push, Logic Read, and Logic Pop instructions.

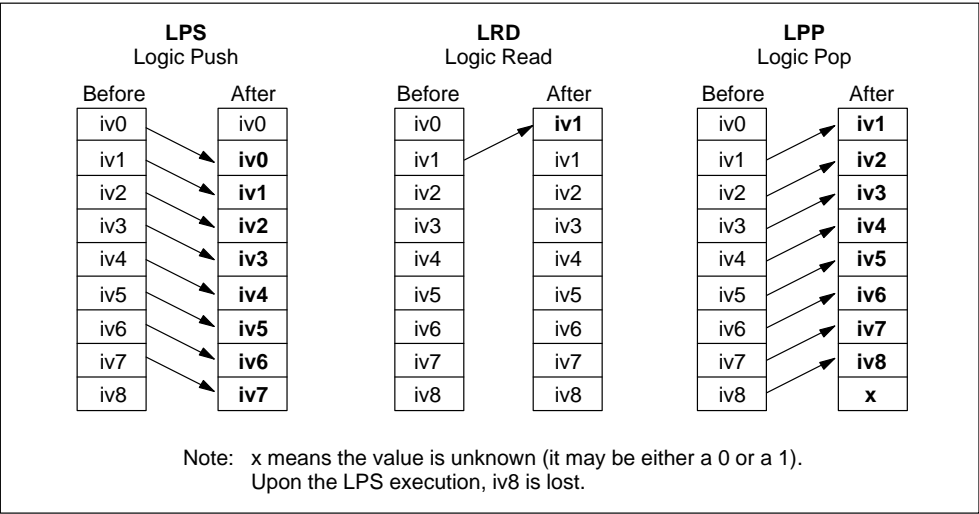


Figure 10-47 Logic Push, Logic Read, and Logic Pop Instructions

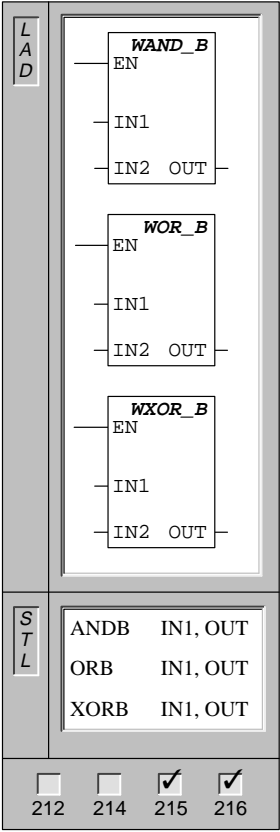
# Logic Stack Example

LAD	STL
<p>Network 1</p> <pre> graph LR     I0_0[I0.0] -- NO --- J1(( ))     I0_1[I0.1] -- NC --- J1     J1 --- J2(( ))     I2_0[I2.0] -- NO --- J2     I2_1[I2.1] -- NC --- J2     J2 --- Q5_0[Q5.0]                     </pre> <p>Network 2</p> <pre> graph LR     I0_0[I0.0] -- NO --- J1(( ))     I0_5[I0.5] -- NC --- J1     J1 --- J2(( ))     I0_6[I0.6] -- NO --- J2     I2_1[I2.1] -- NO --- J2     I1_3[I1.3] -- NO --- J2     I1_0[I1.0] -- NO --- J2     J2 --- Q7_0[Q7.0]     J2 --- Q6_0[Q6.0]     J2 --- Q3_0[Q3.0]                     </pre>	<pre> NETWORK LD  I0.0 LD  I0.1 LD  I2.0 A   I2.1 OLD ALD =   Q5.0  NETWORK LD  I0.0 LPS LD  I0.5 O   I0.6 ALD =   Q7.0 LRD LD  I2.1 O   I1.3 ALD =   Q6.0 LPP A   I1.0 =   Q3.0                     </pre>

Figure 10-48 Example of Logic Stack Instructions for LAD and STL

10.12 Logic Operations

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.

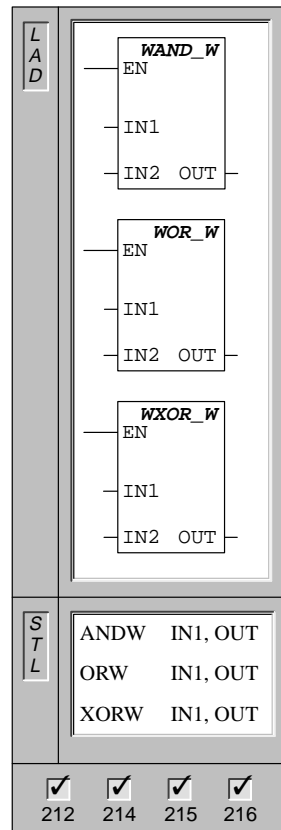
Operands: IN1, IN2: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
OUT: VB, IB, QB, MB, SMB, AC, \*VD, \*AC, SB

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero)

# 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.

Operands: IN1, IN2: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW

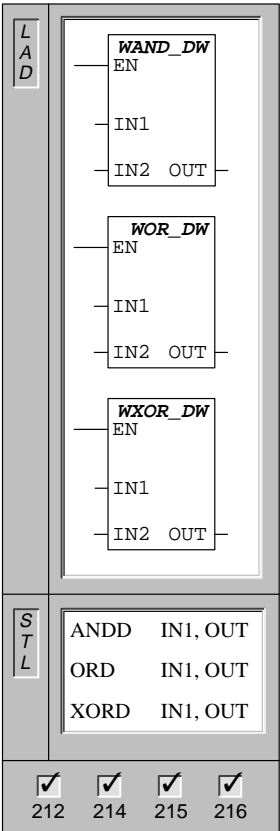
OUT: VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

These instructions affect the following Special Memory bits:

SM1.0 (zero)

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



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

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

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

Operands: IN1, IN2: VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD  
OUT: VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

Note: When programming in LAD, if you specify IN1 to be the same as OUT, you can reduce the amount of memory required.

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



# And, Or, and Exclusive Or Instructions Example

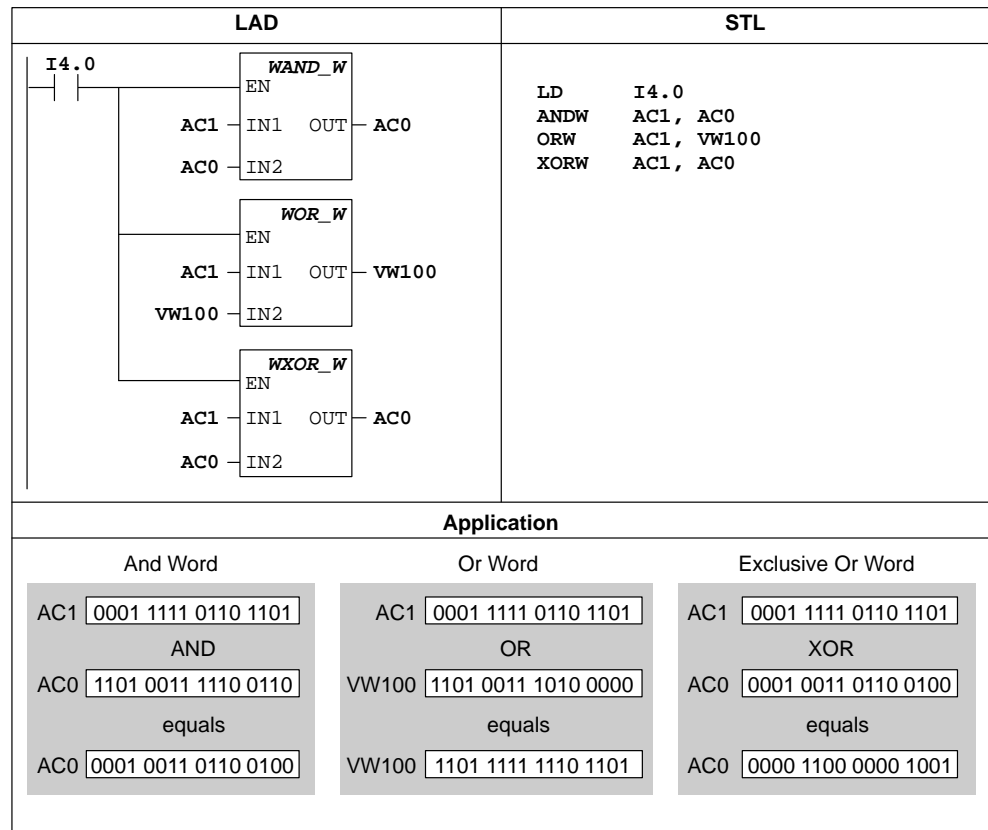
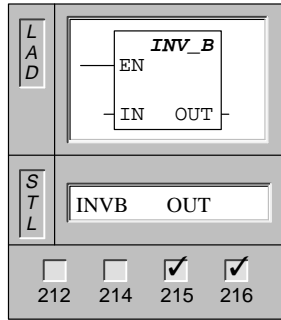


Figure 10-49 Example of the Logic Operation Instructions

### Invert Byte



The **Invert Byte** instruction forms the ones complement of the input byte value, and loads the result in a byte value (OUT).

Operands:    IN:        VB, IB, QB, MB, SMB, AC,  
                              \*VD, \*AC, SB

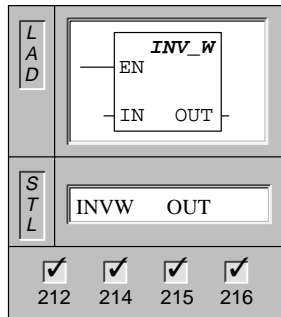
                 OUT:      VB, IB, QB, MB, SMB, AC,  
                              \*VD, \*AC, SB

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

This instruction affects the following Special Memory bits:

SM1.0 (zero)

### Invert Word



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

Operands:    IN:        VW, T, C, IW, QW, MW, SMW, AC,  
                              AIW, Constant, \*VD, \*AC, SW

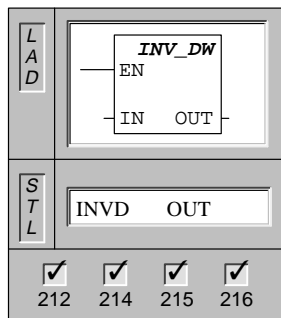
                 OUT:      VW, T, C, IW, QW, MW, SMW, AC,  
                              \*VD, \*AC, SW

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

This instruction affects the following Special Memory bits:

SM1.0 (zero)

### Invert Double Word



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

Operands:    IN:        VD, ID, QD, MD, SMD, AC, HC,  
                              Constant, \*VD, \*AC, SD

                 OUT:      VD, ID, QD, MD, SMD, AC, \*VD, \*AC,  
                              SD

Note: When programming in LAD, if you specify IN to be the same as OUT, you can reduce the amount of memory required.

This instruction affects the following Special Memory bits:

SM1.0 (zero)

# Invert Example

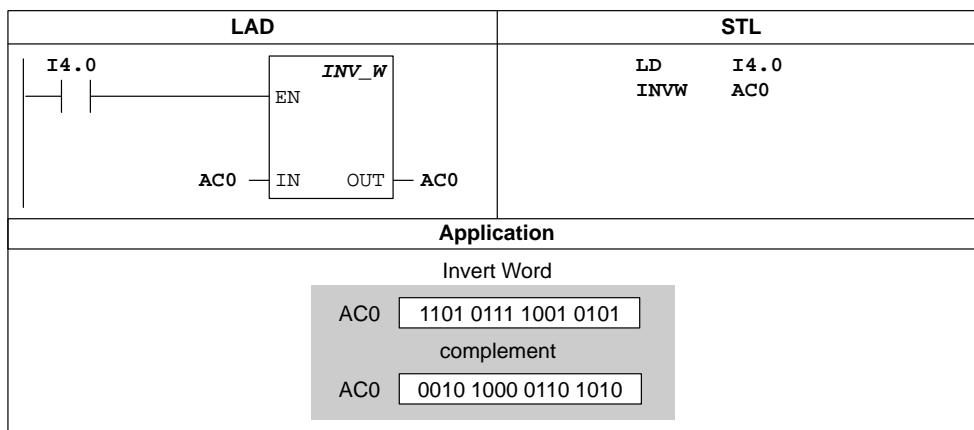
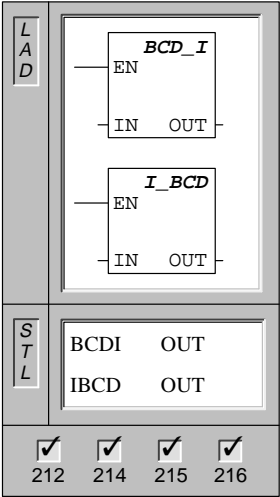


Figure 10-50 Example of Invert Instruction for LAD and STL

10.13 Conversion Instructions

BCD to Integer, Integer to BCD Conversion



The **BCD to Integer** instruction converts the input Binary-Coded Decimal value and loads the result in OUT.

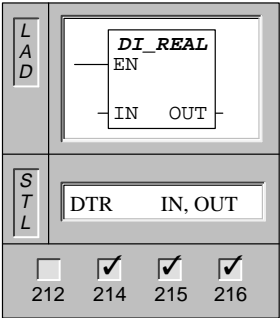
The **Integer to BCD** instruction converts the input integer value to a Binary-Coded Decimal value and loads the result in OUT.

Operands:    IN:        VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW  
                 OUT:      VW, T, C, IW, QW, MW, SMW, AC, \*VD, \*AC, SW

Note: When programming in ladder logic, if you specify IN to be the same as OUT, you can reduce the amount of memory used.

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

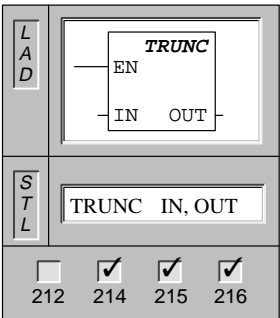
Double Word Integer to Real



The **Double Word Integer to Real** instruction converts a 32-bit, signed integer (IN) into a 32-bit real number (OUT).

Operands:    IN:        VD, ID, QD, MD, SMD, AC, HC, Constant, \*VD, \*AC, SD  
                 OUT:      VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

Truncate



The **Truncate** instruction converts a 32-bit, real number (IN) into a 32-bit signed integer (OUT). Only the whole number portion of the real number is converted (round-to-zero).

Operands:    IN:        VD, ID, QD, MD, SMD, AC, Constant, \*VD, \*AC, SD  
                 OUT:      VD, ID, QD, MD, SMD, AC, \*VD, \*AC, SD

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

# Convert and Truncate Example

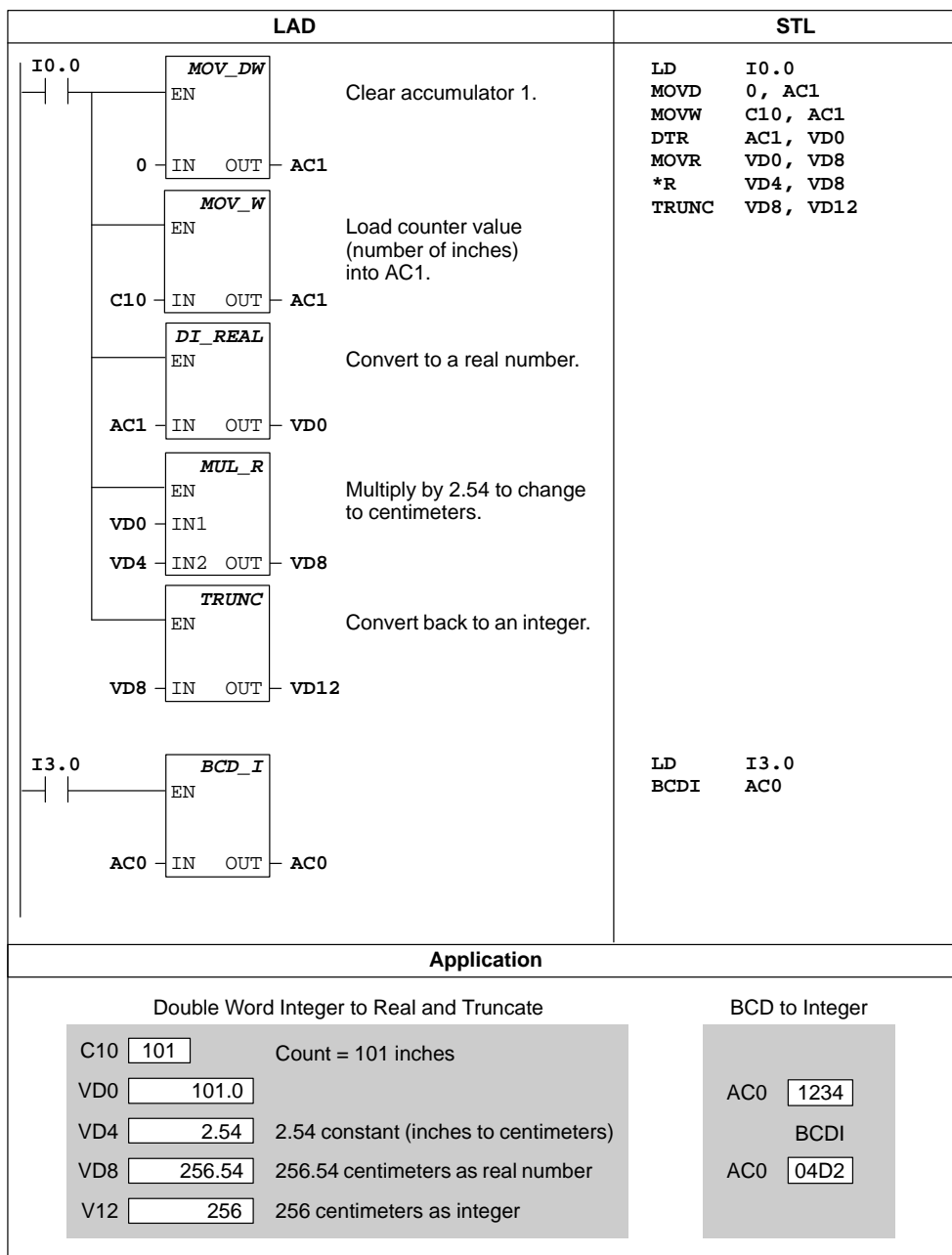
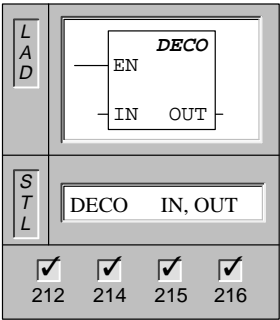


Figure 10-51 Example of Real Number Conversion Instruction

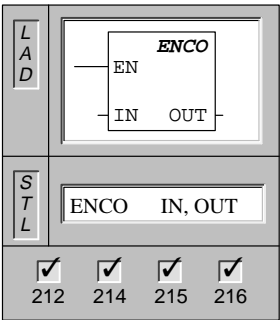
Decode



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

Operands: IN: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
OUT: VW, T, C, IW, QW, MW, SMW, AC, AQW, \*VD, \*AC, SW

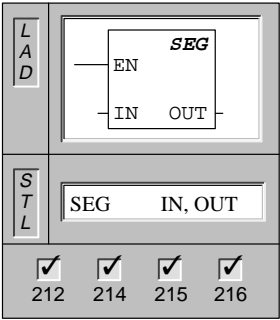
Encode



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

Operands: IN: VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, \*VD, \*AC, SW  
OUT: VB, IB, QB, MB, SMB, AC, \*VD, \*AC, SB

Segment



The **Segment** instruction generates 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).

Operands: IN: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB  
OUT: VB, IB, QB, MB, SMB, AC, \*VD, \*AC, SB

Figure 10-52 shows the seven segment display coding used by the Segment instruction.

(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

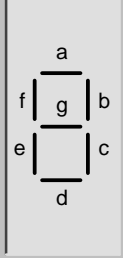
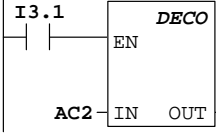


Figure 10-52 Seven Segment Display Coding

# Decode, Encode Examples

LAD	STL
 <p>Set the bit that corresponds to the error code in AC2.</p>	<pre>LD    I3.1 DECO  AC2, VW40</pre>
Application	
<p>AC2 contains the error code 3. The DECO instruction sets the bit in VW40 that corresponds to this error code.</p>	

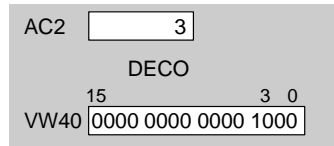
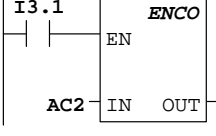


Figure 10-53 Example of Setting an Error Bit Using Decode

LAD	STL
 <p>Convert the error bit in AC2 to the error code in VB40.</p>	<pre>LD    I3.1 ENCO  AC2, VB40</pre>
Application	
<p>AC2 contains the error bit. The ENCO instruction converts the least significant bit set to an error code that is stored in VB40.</p>	

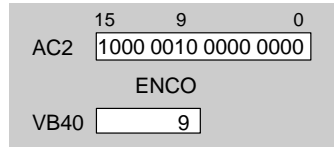


Figure 10-54 Example of Converting the Error Bit into an Error Code Using Encode

# Segment Example

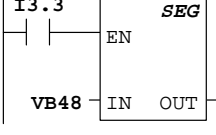
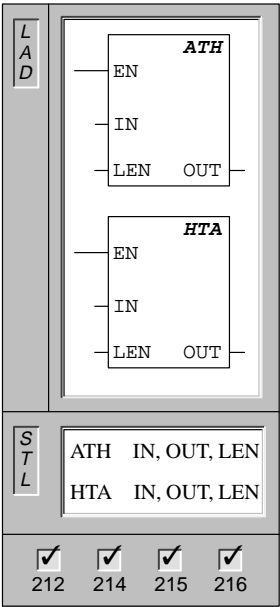
LAD	STL
	<pre>LD    I3.3 SEG   VB48, AC1</pre>
Application	
<p>VB48 05 SEG AC1 6D (display character)</p>	

Figure 10-55 Example of Segment Instruction

ASCII to HEX, HEX to ASCII



The **ASCII to HEX** instruction converts the ASCII string of length (LEN), starting with the character (IN), to hexadecimal digits starting at a specified location (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 a specified location (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.

Operands: IN, OUT: VB, IB, QB, MB, SMB, \*VD, \*AC, SB  
LEN: VB, IB, QB, MB, SMB, AC, Constant, \*VD, \*AC, SB

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

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



ASCII to HEX Example

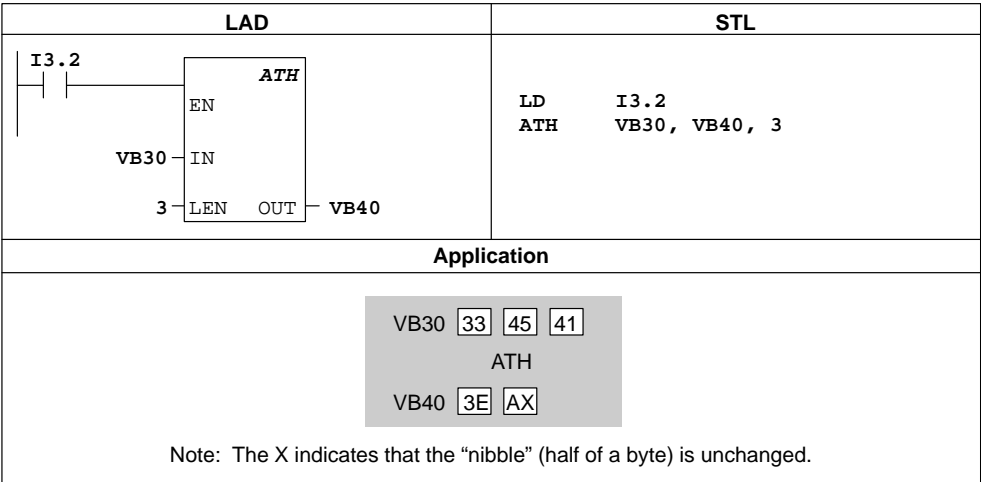
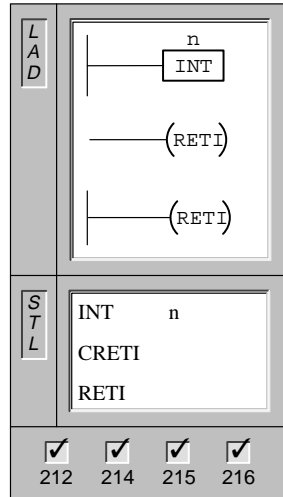


Figure 10-56 Example of ASCII to HEX Instruction

## 10.14 Interrupt and Communications Instructions

### Interrupt Routine, Return from Interrupt Routine



The **Interrupt Routine** instruction marks the beginning of the interrupt routine (n).

The **Conditional Return from Interrupt** instruction may be used to return from an interrupt, based upon the condition of the preceding logic.

The **Unconditional Return from Interrupt** coil must be used to terminate each interrupt routine.

Operands:      n:            0 to 127

### Interrupt Routines

You can identify each interrupt routine by an interrupt routine label that marks the entry point into the routine. The routine consists of all your instructions between the interrupt label and the unconditional return from interrupt instruction. The interrupt routine is executed in response to an associated internal or external event. You can exit the routine (thereby returning control to the main program) by executing the unconditional return from interrupt instruction (RETI), or by executing a conditional return from an interrupt instruction. The unconditional return instruction is always required.

### 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

Restrictions for using the interrupt routine follow:

- Put all interrupt routines after the end of the main ladder program.
- You cannot use the DISI, ENI, CALL, HDEF, FOR/NEXT, LSCR, SCRE, SCRT, and END instructions in an interrupt routine.
- You must terminate each interrupt routine by an unconditional return from interrupt instruction (RETI).

### 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.

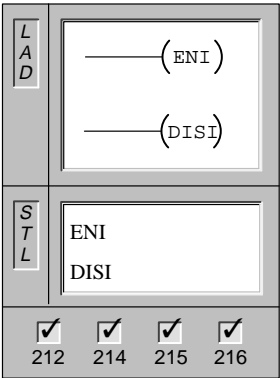
### Sharing Data Between the Main Program and Interrupt Routines

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 ladder logic, then correct shared access can be ensured by establishing the convention that access to shared memory locations be made using only Move instructions (MOV\_B, MOV\_W, MOV\_DW, MOV\_R). 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



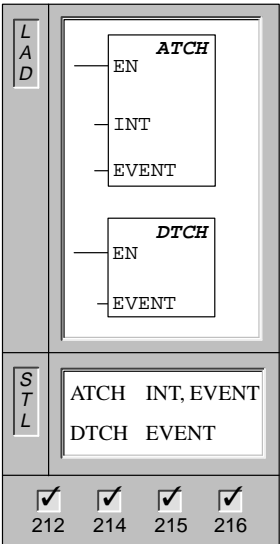
The **Enable Interrupt** instruction globally enables processing of all attached interrupt events.

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

Operands:      None

When you make the transition to the RUN mode, you disable the interrupts. 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.

Attach Interrupt, Detach Interrupt



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

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

Operands:      INT :      0 to 127  
                  EVENT:    0 to 26

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 10-13 lists the different types of interrupt events.

Table 10-13 Descriptions of Interrupt Events

Event Number	Interrupt Description	212	214	215	216
0	Rising edge, I0.0*	Y	Y	Y	Y
1	Falling edge, I0.0*	Y	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	Y
9	Port 0: Transmit complete	Y	Y	Y	Y
10	Timed interrupt 0, SMB34	Y	Y	Y	Y
11	Timed interrupt 1, SMB35		Y	Y	Y
12	HSC0 CV=PV (current value = preset value)*	Y	Y	Y	Y
13	HSC1 CV=PV (current value = preset value)		Y	Y	Y
14	HSC1 direction input changed		Y	Y	Y
15	HSC1 external reset		Y	Y	Y
16	HSC2 CV=PV (current value = preset value)		Y	Y	Y
17	HSC2 direction input changed		Y	Y	Y
18	HSC2 external reset		Y	Y	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
22	Timer T96 CT=PT interrupt			Y	Y
23	Port 0: Receive message complete			Y	Y
24	Port 1: Receive message complete				Y
25	Port 1: Receive character				Y
26	Port 1: Transmit complete				Y
* If event 12 (HSC0, PV = CV) is attached to an interrupt, then neither event 0 nor 1 can be attached to interrupts. Likewise, if either event 0 or 1 is attached to an interrupt, then event 12 cannot be attached to an interrupt.					

### Communication Port Interrupts

The serial communications port of the programmable logic controller can be controlled by the ladder logic or statement list 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 10-14 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 an error condition that must receive immediate attention when the event happens.

Table 10-14 Rising/Falling Edge Interrupts Supported

I/O Interrupts	CPU 212	CPU 214	CPU 215	CPU 216
I/O Points	I0.0	I0.0 to I0.3	I0.0 to I0.3	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 one or more timed interrupts (see Table 10-15). 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 5 ms to 255 ms. You must write the cycle time in SMB34 for timed interrupt 0, and in SMB35 for timed interrupt 1.

Table 10-15 Timed Interrupts Supported

Timed Interrupts	CPU 212	CPU 214	CPU 215	CPU 216
Number of timed interrupts supported	1	2	2	2

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.

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 10-58 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 timers (TON) 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 (refer to section 10.5). 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 10-16.

Table 10-16 Interrupt Queues and Maximum Number of Entries per Queue

Queue	CPU 212	CPU 214	CPU 215	CPU 216
Communications queue	4	4	4	8
I/O Interrupt queue	4	16	16	16
Timed Interrupt queue	2	4	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 10-17. 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 10-17 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 10-18 shows the interrupt event, priority, and assigned event number.

Table 10-18 Descriptions of Interrupt Events

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*
0	Rising edge, I0.0**	I/O (middle)	0
2	Rising edge, I0.1		1
4	Rising edge, I0.2		2
6	Rising edge, I0.3		3
1	Falling edge, I0.0**		4
3	Falling edge, I0.1		5
5	Falling edge, I0.2		6
7	Falling edge, I0.3		7
12	HSC0 CV=PV (current value = preset value)**		0
13	HSC1 CV=PV (current value = preset value)		8
14	HSC1 direction input changed		9
15	HSC1 external reset		10
16	HSC2 CV=PV (current value = preset value)		11
17	HSC2 direction input changed		12
18	HSC2 external reset		13
19	PLS0 pulse count complete interrupt		14
20	PLS1 pulse count complete interrupt		15
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
<div>* Since communication is inherently half-duplex, both transmit and receive are the same priority.</div> <div>** If event 12 (HSC0, PV = CV) is attached to an interrupt, then neither event 0 nor 1 can be attached to interrupts. Likewise, if either event 0 or 1 is attached to an interrupt, then event 12 cannot be attached to an interrupt.</div>			

Interrupt Examples

Figure 10-57 shows an example of the Interrupt Routine instructions.

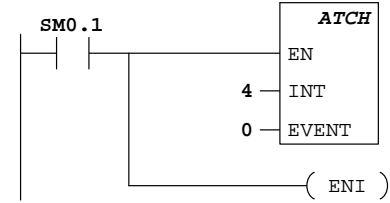
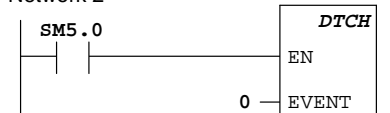
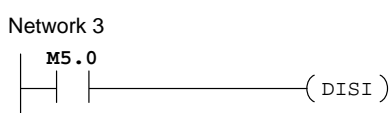

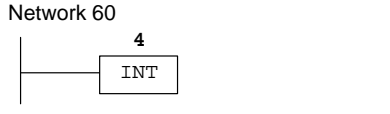


LAD	STL
<p>Network 1</p>  <p>On the first scan: Define interrupt routine 4 to be a rising edge interrupt routine for I0.0.</p> <p>Globally enable interrupts.</p>	<p>Network 1</p> <pre>LD SM0.1 ATCH 4, 0 ENI</pre>
<p>Network 2</p>  <p>If an I/O error is detected, disable the rising edge interrupt for I0.0. (This rung is optional.)</p>	<p>Network 2</p> <pre>LD SM5.0 DTCH 0</pre>
<p>Network 3</p>  <p>Disable all interrupts when M5.0 is on.</p>	<p>Network 3</p> <pre>LD M5.0 DISI</pre> <p>.</p> <p>.</p> <p>.</p>
<p>Network 50</p>  <p>End of main ladder.</p>	<p>Network 50</p> <pre>MEND</pre> <p>.</p> <p>.</p> <p>.</p>
<p>Network 60</p>  <p>I/O rising edge interrupt subroutine.</p>	<p>Network 60</p> <pre>INT 4</pre> <p>.</p> <p>.</p> <p>.</p>
<p>Network 65</p>  <p>Conditional return based on I/O error</p>	<p>Network 65</p> <pre>LD SM5.0 CRETI</pre>
<p>Network 66</p>  <p>End of I0.0 rising edge interrupt routine.</p>	<p>Network 66</p> <pre>RETI</pre>

Figure 10-57 Example of Interrupt Instructions

Figure 10-58 shows how to set up a timed interrupt to read the value of an analog input.

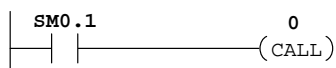

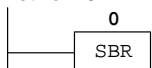
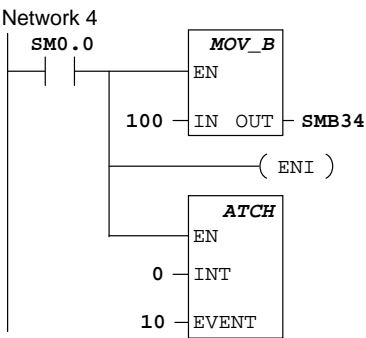
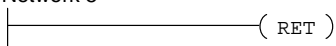
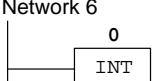
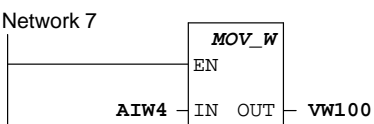

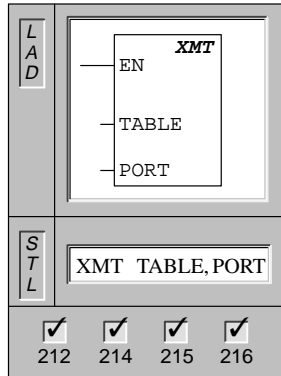
LAD		STL
<b>Main Program</b>		
<p>Network 1</p>  <p>Network 2</p> 	<p>First scan memory bit: Call Subroutine 0.</p>	<p>Network 1</p> <pre>LD    SM0.1 CALL  0</pre> <p>Network 2</p> <pre>MEND</pre>
<b>Subroutines</b>		
<p>Network 3</p>  <p>Network 4</p>  <p>Network 5</p> 	<p>Begin Subroutine 0.</p> <p>Always on memory bit: Set timed interrupt 0 interval to 100 ms.</p> <p>Global Interrupt Enable</p> <p>Attach timed interrupt 0 to Interrupt routine 0.</p> <p>Terminate Subroutine</p>	<p>Network 3</p> <pre>SBR    0</pre> <p>Network 4</p> <pre>LD      SM0.0 MOVB    100, SMB34  ENI  ATCH    0, 10</pre> <p>Network 5</p> <pre>RET</pre>
<b>Interrupt Routines</b>		
<p>Network 6</p>  <p>Network 7</p>  <p>Network 8</p> 	<p>Begin Interrupt routine 0.</p> <p>Sample AIW4.</p> <p>Terminate Interrupt routine.</p>	<p>Network 6</p> <pre>INT     0</pre> <p>Network 7</p> <pre>MOVW    AIW4, VW100</pre> <p>Network 8</p> <pre>RETI</pre>

Figure 10-58 Example of How to Set Up a Timed Interrupt to Read the Value of an Analog Input

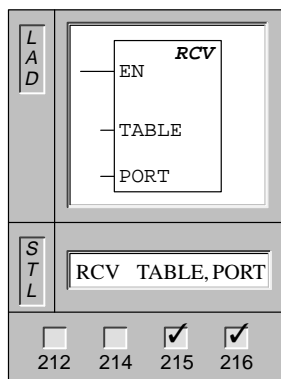
## Transmit, Receive



The **Transmit** instruction invokes the transmission of the data buffer (TABLE). 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.

Operands: TABLE: VB, IB, QB, MB, SMB, \*VD, \*AC, SB  
PORT: 0 to 1

The XMT instruction is used in Freeport mode to transmit data by means of the communication port(s).



The **Receive** instruction invokes setup changes that initiate or terminate 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 (TABLE). The first entry in the data buffer specifies the number of bytes received.

Operands: TABLE: VB, IB, QB, MB, SMB, \*VD, \*AC, SB  
PORT: 0 to 1

The RCV instruction is used in Freeport mode to receive data by means of the communication port(s).

## 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 ladder logic 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 mode switch. When SM0.7 is equal to 0, the switch is in TERM position; when SM0.7 = 1, the 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 10-19.

Table 10-19 Special Memory Bytes SMB30 and SMB130

Port 0	Port 1	Description
Format of SMB30	Format of SMB130	<div style="display: flex; justify-content: space-between; align-items: center;"> <div> <div>MSB</div> <div>7</div> </div> <div> <div>LSB</div> <div>0</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">p</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">p</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">d</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">b</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">b</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">b</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">m</div> <div style="border: 1px solid black; padding: 2px 5px; margin: 0 2px;">m</div> </div> <div style="margin-left: 20px;">Freeport mode control byte</div>
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 (for CPU 212: = 19,200 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: For Port 0 operation, one stop bit is generated for all configurations except for the 7 bits per character, no parity case, where two stop bits are generated. For Port 1 operation, one stop bit is generated for all configurations.		

**Using the XMT Instruction to Transmit Data**

You can use the XMT instruction to facilitate transmission. 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.

**Using the RCV Instruction to Receive Data**

You can use the RCV instruction to facilitate receiving messages. 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 SM86.

SMB86 (or SMB186) will be non-zero when the RCV box is inactive. 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 10-20 (SM86 through SM94 for port 0, and SM186 through SM194 for port 1) for descriptions of the start and end message conditions.

---

**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.

---

Table 10-20 Special Memory Bytes SMB86 to SMB94, and SMB186 to SMB194

Port 0	Port 1	Description
SMB86	SMB186	<div> <div> <div>MSB</div> <div>7</div> <div>n</div> <div>r</div> <div>e</div> <div>0</div> <div>0</div> <div>t</div> <div>c</div> <div>p</div> <div>LSB</div> <div>0</div> </div> <div>Receive message status byte</div> </div> <p> 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 </p>
SMB87	SMB187	<div> <div> <div>MSB</div> <div>7</div> <div>n</div> <div>x</div> <div>y</div> <div>z</div> <div>m</div> <div>t</div> <div>0</div> <div>0</div> <div>LSB</div> <div>0</div> </div> <div>Receive message control byte</div> </div> <p> 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.  These bits define the criteria for identifying the message (including both the start-of-message and end-of-message criteria). To determine the start of a message, the enabled start of message criteria are logically ANDed, and must occur in sequence (idle line followed by a start character). To determine the end of a message, the enabled end of message criteria are logically ORed.  Equations for start and stop criteria:  Start of Message = z * x  End of Message = y + t + maximum character count reached  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. </p>
SMB88	SMB188	Start of message character
SMB89	SMB189	End of message character
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.



Table 10-20 Special Memory Bytes SMB86 to SMB94, and SMB186 to SMB194

Port 0	Port 1	Description
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.

LAD		STL
<div>Network 1</div> <div><div><div><div>SM0.1</div><div></div></div><div><div>MOV_B</div><div>EN</div></div><div><div>16#9</div><div>IN</div><div>OUT</div><div>SMB30</div></div><div><div>MOV_B</div><div>EN</div></div><div><div>16#B0</div><div>IN</div><div>OUT</div><div>SMB87</div></div><div><div>MOV_B</div><div>EN</div></div><div><div>16#A</div><div>IN</div><div>OUT</div><div>SMB89</div></div><div><div>MOV_W</div><div>EN</div></div><div><div>+5</div><div>IN</div><div>OUT</div><div>SMW90</div></div><div><div>MOV_B</div><div>EN</div></div><div><div>100</div><div>IN</div><div>OUT</div><div>SMB94</div></div><div><div>ATCH</div><div>EN</div></div><div><div>0</div><div>INT</div></div><div><div>23</div><div>EVENT</div></div><div><div>ATCH</div><div>EN</div></div><div><div>1</div><div>INT</div></div><div><div>9</div><div>EVENT</div></div><div><div>( ENI )</div></div><div><div>RCV</div><div>EN</div></div><div><div>VB100</div><div>TABLE</div></div><div><div>0</div><div>PORT</div></div></div></div> <div><div>On the first scan:</div><div><div>- Initialize freeport</div><div>- Select 9600 baud</div><div>- Select 8 data bits</div><div>- Select no parity</div></div><div><div>Initialize RCV message control byte</div><div><div>- RCV enabled</div><div>- Detect end of message character</div><div>- Detect idle line condition as message start condition</div></div><div><div>Set end of message character to hex 0A (line feed)</div></div><div><div>Set idle line timeout to 5 ms</div></div><div><div>Set maximum number of characters to 100</div></div><div><div>Attach interrupt to receive complete event</div></div><div><div>Attach interrupt to transmit complete event</div></div><div><div>Enable user interrupts</div></div><div><div>Enable receive box with buffer at VB100 for port 0</div></div></div></div> <td><div>Network 1</div><div><div>LD</div><div>SM0.1</div><div>MOVB</div><div>16#9, SMB30</div><div>MOVB</div><div>16#B0, SMB87</div><div>MOVB</div><div>16#0A, SMB89</div><div>MOVW</div><div>+5, SMW90</div><div>MOVB</div><div>100, SMB94</div><div>ATCH</div><div>0, 23</div><div>ATCH</div><div>1, 9</div><div>ENI</div><div>RCV</div><div>VB100, 0</div></div></td>		<div>Network 1</div> <div><div>LD</div><div>SM0.1</div><div>MOVB</div><div>16#9, SMB30</div><div>MOVB</div><div>16#B0, SMB87</div><div>MOVB</div><div>16#0A, SMB89</div><div>MOVW</div><div>+5, SMW90</div><div>MOVB</div><div>100, SMB94</div><div>ATCH</div><div>0, 23</div><div>ATCH</div><div>1, 9</div><div>ENI</div><div>RCV</div><div>VB100, 0</div></div>

Figure 10-59Example of Transmit Instruction

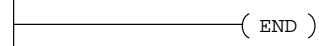
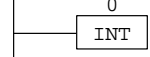
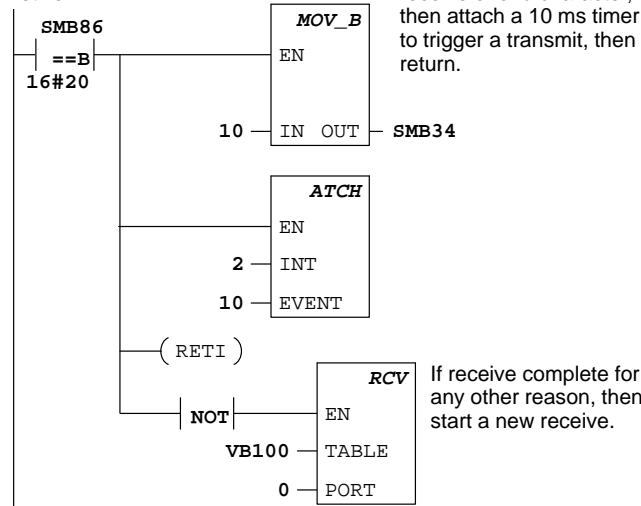


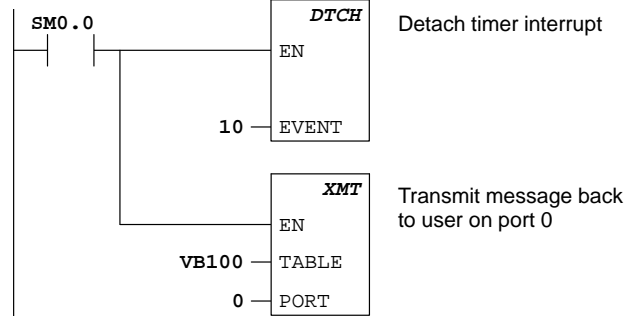
LAD	STL
<p>Network 2</p>  <p>Network 3</p>  <p>Network 4</p>  <p>Network 5</p>  <p>Network 6</p>  <p>Network 7</p> 	<p>Network 2</p> <pre>MEND</pre> <p>Network 3</p> <pre>INT 0</pre> <p>Network 4</p> <pre>LDB=   SMB86, 16#20 MOVB   10, SMB34 ATCH   2, 10 CRETI NOT RCV    VB100, 0</pre> <p>Network 5</p> <pre>RETI</pre> <p>Network 6</p> <pre>INT 2</pre> <p>Network 7</p> <pre>LD     SM0.0 DTCH   10 XMT    VB100, 0</pre>

Figure 10-60 Example of Transmit Instruction (continued)

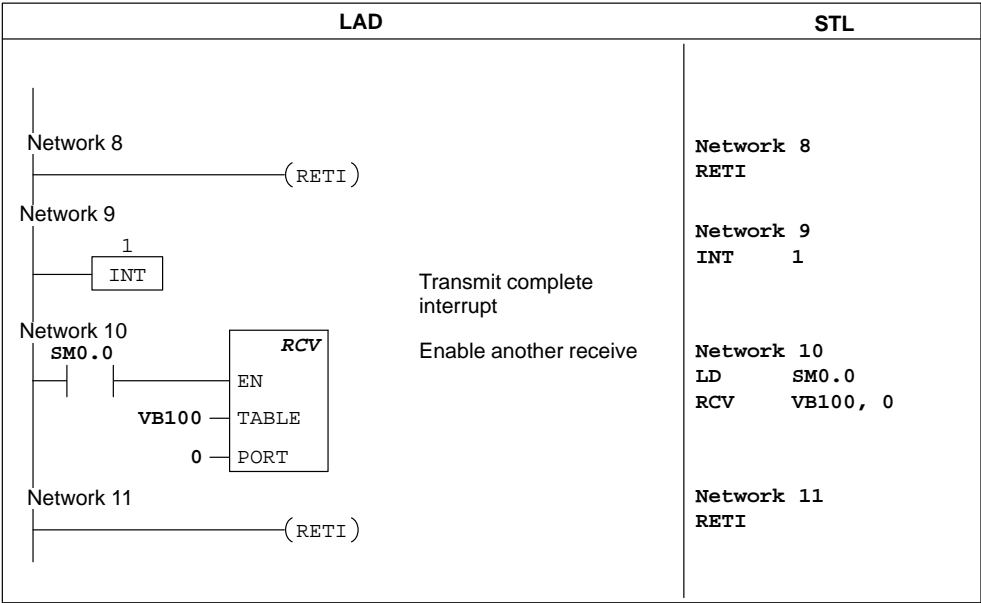
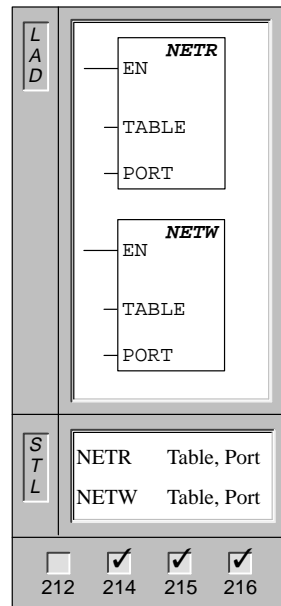


Figure 10-60 Example of Transmit Instruction (continued)

# 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 (TABLE).

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 (TABLE).

Operands:     Table:     VB, MB, \*VD, \*AC  
                  Port:       0 to 1

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. 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 in a given S7-200 programmable logic controller.

Figure 10-60 defines the table that is referenced by the TABLE parameter in the NETR and NETW instructions.

Byte Offset	7	0
0	D	A
1	Remote station address	
2	Pointer to the data	
3	area in the	
4	remote station	
5	(I, Q, M, S, or V)	
6	Data length	
7	Data byte 0	
8	Data byte 1	
	:	
22	Data byte 15	

D Done (function has been completed):
0 = not done
1 = done

A Active (function has been queued):
0 = not active
1 = active

E Error (function returned an error):
0 = no error
1 = error

**Remote station address:** the address of the PLC whose data is to be accessed.

**Pointer to the data area in the remote station:** an indirect pointer to the data that is to be accessed.

**Data length:** the number of bytes of data that is to be accessed in the remote station (1 to 16 bytes).

**Receive or transmit data area:** 1 to 16 bytes reserved for the data, as described below:

For NETR, this data area is where the values that are read from the remote station are stored after execution of the NETR.

For NETW, this data area is where the values to be sent to the remote station are stored before execution of the NETW.

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 10-60 Definition of TABLE for NETR and NETW

Example of Network Read and Network Write

Figure 10-61 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 212 modules are used to control the case packers and a CPU 214 module equipped with a TD 200 operator interface is used to control the diverter. Figure 10-61 shows the network setup.

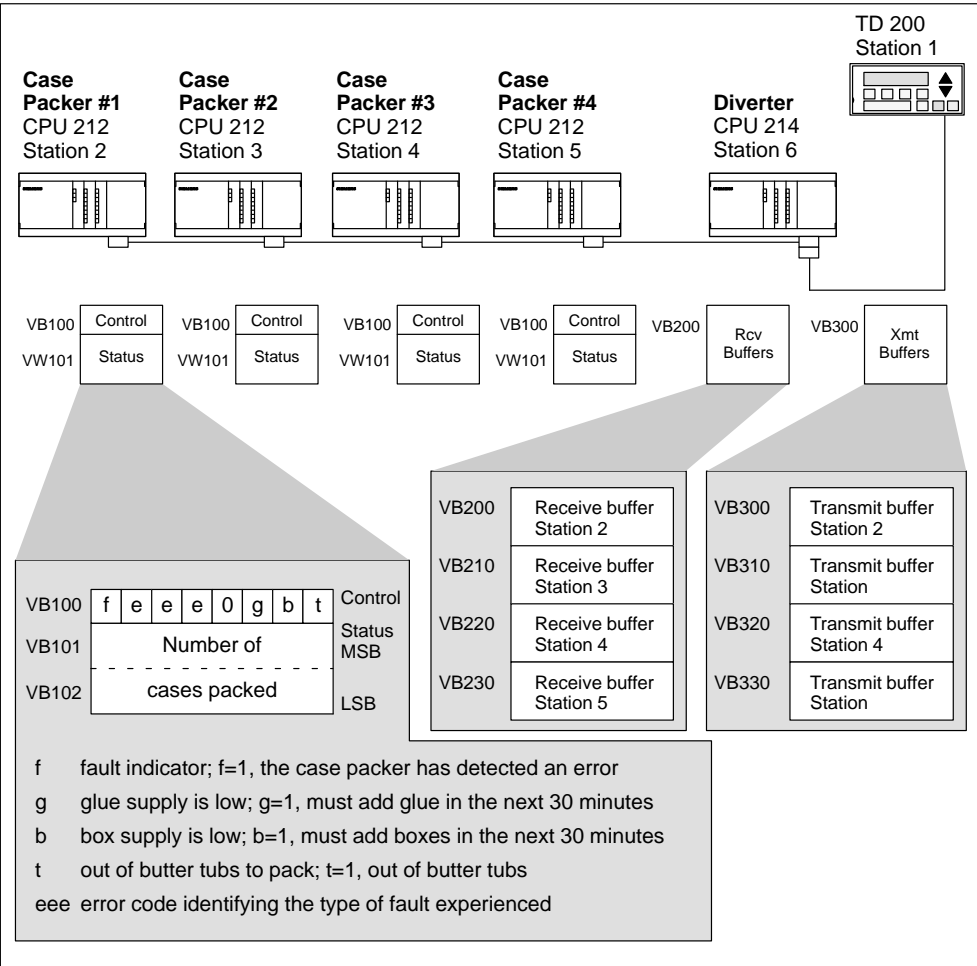


Figure 10-61 Example of NETR and NETW Instructions

The receive and transmit buffers for accessing the data in station 2 (located at VB200 and VB300, respectively) are shown in detail in Figure 10-62.

The CPU 214 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 10-63.

Diverter's Receive Buffer for reading from Case Packer #1						Diverter's Transmit Buffer for clearing the count of Case Packer #1					
	7				0		7				0
VB200	D	A	E	0	Error code	VB300	D	A	E	0	ErrorCode
VB201	Remote station address					VB301	Remote station address				
VB202	Pointer to the					VB302	Pointer to the				
VB203	data area					VB303	data area				
VB204	in the					VB304	in the				
VB205	Remote station = (&VB100)					VB305	Remote station = (&VB101)				
VB206	Data length = 3 bytes					VB306	Data length = 2 bytes				
VB207	Control					VB307	0				
VB208	Status (MSB)					VB308	0				
VB209	Status (LSB)										

Figure 10-62 Sample TABLE Data for NETR and NETW Example

LAD	STL
<p>Network 1</p> <p>SM0.1</p> <p>2 IN OUT SMB30</p> <p>0 IN OUT VW200</p> <p>68 N</p> <p>Network 2</p> <p>V200.7 VW208</p> <p>==I 100</p> <p>2 IN OUT VB301</p> <p>MOV_D</p> <p>EN</p> <p>&amp;VB101 IN OUT VD302</p> <p>2 IN OUT VB306</p> <p>MOV_W</p> <p>EN</p> <p>0 IN OUT VW307</p> <p>NETW</p> <p>EN</p> <p>VB300 TABLE</p> <p>0 PORT</p> <p>Network 3</p> <p>V200.7</p> <p>VB207 IN OUT VB400</p> <p>Network 4</p> <p>SM0.1 V200.6 V200.5</p> <p>2 IN OUT VB201</p> <p>MOV_D</p> <p>EN</p> <p>&amp;VB100 IN OUT VD202</p> <p>3 IN OUT VB206</p> <p>MOV_B</p> <p>EN</p> <p>VB200 TABLE</p> <p>0 PORT</p>	<p>Network 1</p> <p>LD SM0.1</p> <p>MOVB 2, SMB30</p> <p>FILL 0, VW200, 68</p> <p>Network 2</p> <p>LD V200.7</p> <p>AW= VW208, 100</p> <p>MOVB 2, VB301</p> <p>MOVD &amp;VB101, VD302</p> <p>MOVB 2, VB306</p> <p>MOVW 0, VW307</p> <p>NETW VB300, 0</p> <p>Network 3</p> <p>LD V200.7</p> <p>MOVB VB207, VB400</p> <p>Network 4</p> <p>LDN SM0.1</p> <p>AN V200.6</p> <p>AN V200.5</p> <p>MOVB 2, VB201</p> <p>MOVD &amp;VB100, VD202</p> <p>MOVB 3, VB206</p> <p>NETR VB200, 0</p>

Figure 10-63 Example of NETR and NETW Instructions