1988

NEAR EAST UNIVERSITY

GRADUATION PROJECT
EE-400

FACULTY OF ELECTRICAL & ELECTRONIC ENGINEERING

"Step Motor Control with PLC"

INDEX

Introduction To Programmable Logic Controllers (PLCs)

HISTORY OF PLC's

In 1970's discovering microprocessors many thing was begun to change in the world. They can make many jobs in very short times and they can be modulates in desired wishes. In industrial control complex control systems can be realise easily with microprocessors and the PLC was born. By one account, PLCs were born in a Request for Proposals issued by GM to industrial control vendors. GM was tired of replacing and re-wiring complex relay panels every time tailfins got a little bigger or smaller.

ADVANTAGES

-ACCURACY:

In relay control systems logical knowledge's carry's in electro mechanical contactors, they can be lose their knowledge's because of mechanical errors. But PLC's are microprocessor based system so logical knowledge's carries inside the processor, so that PLC's are more accurate than relay type of controllers.

-FLEXIBLITY:

When need of any changing of control logic, relay type of controllers modification are so hard, in PLC's this changing can be made with PLC programmer equipment.

-COMMUNICATION:

PLC's are computer based systems. So that they can transfers their position in working to another PC or they can take external inputs to another PC, with this specification we can control the system were they are and we can effect the system with our PC.(Help of extra equipment's.) With relays it can not be possible.

Logic Control in Industrial Automation

Everyday examples of these systems are machines like dishwashers, clothes washers and dryers, and elevators. In these systems, the outputs tend to be 220vac power signals to motors, solenoids, and indicator lights, and the inputs are DC or AC signals from user interface switches, motion limit switches, binary liquid level sensors, etc. Another major function in these types of controlers is timing.

Relays and Ladder Logic

In the "old days" (i.e. before the 1980's) these types of controllers were implemented with relays.

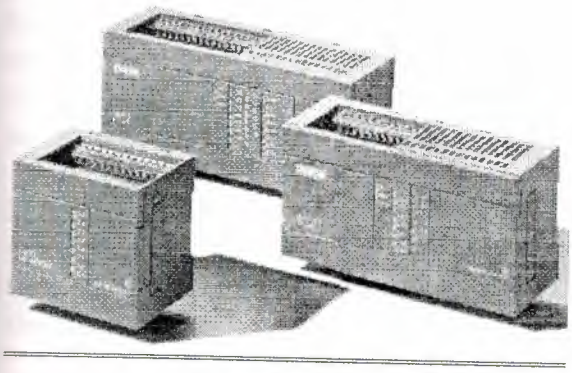
Relays are a technology from the early days of electricity in which an electromagnet activates an electrical switch. When current flows in the coil, electrical contacts are pulled together or apart making (or breaking) a circuit. Relays are electrically, thermally, and mechanically rugged, easy to design with, cheap, and capable of controlling very large currents in their output contacts.

Relays can be thought of as logic gates. For example, if two normally open relays are wired in series, and one end of the resulting output circuit is attached to a voltage source, then the two coils form the inputs of a AND gate: only if current is flowing in BOTH input coils will current flow in the output circuit. A typical application in a washing machine might be to implement the rule that

if (state = wash) **AND** (door = close) Take water inside

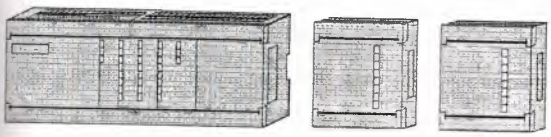
A collection of these boolean rules can be represented by a diagram in which each output circuit is drawn horizontally between vertical "power rails".

The shape of these diagrams invariably led to the name "Ladder Diagrams" and "Ladder Logic" to describe them. The term "Relay Ladder Logic" (RLL) describes this logic notation. By including interconnections between the horizontal rungs, it is possible to create latches ("flip-flops") and implement state transitions. Although LL "state machines" get quite complex and are typically not designed with the convenience of finite state machine theory, they have become widely used and supported by technical workers. Because the logic was implemented in physical wiring, it was difficult to change as new functions were required.
To learning and describing as well as possible I choice SIMATIC® S7-200 PLC and MICROWIN / STEP7 software.



System Overview

A typical S7-200 system will include an S7-200 base unit which includes the central processing unit, power supply, and discrete input and output points. Expansion module contain additional input or output points and are connected to the base unit using bus connectors. The central processing unit has a built-in communications port for programming or talking with intelligent ASCII devices.



CPU Overview

The S7-200 series is a line of small, compact, micro-programmable logic controllers and expansion modules that can be used for a variety of programming applications. There are two types of base units in the S7-200 product line, CPU 212 and CPU 214. Each base unit comes in different models to accommodate the type of power supply, inputs and outputs you require.

Architecture

This section relates to how the S7-200 CPU arranges data and how it executes your program during it's scan cycle.

Memory map

The memory space of the S7-200 is divided into five data areas and six data objects. To reference a memory location for use, you must address that location. The addressing conventions allow memory to be accessed as bits, bytes, words and double words. All addresses are zero-based.

Data space is highly flexible, and it allows read and write access to all memory areas as bits, bytes, words and double words. Data objects are the memory locations that are associated with devices (such as the current value of a counter or the temperature value of an oven). Access to data objects is more restrictive because the data object can be addressed only according to the intended use of that object.

Data Areas. Data memory contains variable memory, and input image register, and output image register, internal memory bits, and special memory bits. This memory is accessed by a byte.bit convention. For example to access bit 3 of Variable Memory byte 25 you would use the address V25.3.

The following table shows the identifiers and ranges for each of the data area memory types:

Area Identifier	Data Area	CPU 212	CPU 214
I	Input	I0.0 to I7.7	I0.0 to I7.7
Q	Output	Q0.0 to Q7.7	Q0.0 to Q7.7
M	Internal Memory	M0.0 to M15.7	M0.0 to M31.7
SM	Special Memory	SM0.0 to SM 45.7	SM0.0 to SM 85.7
V	Variable Memory	V0.0 to V1023.7	V0.0 to V4095.7

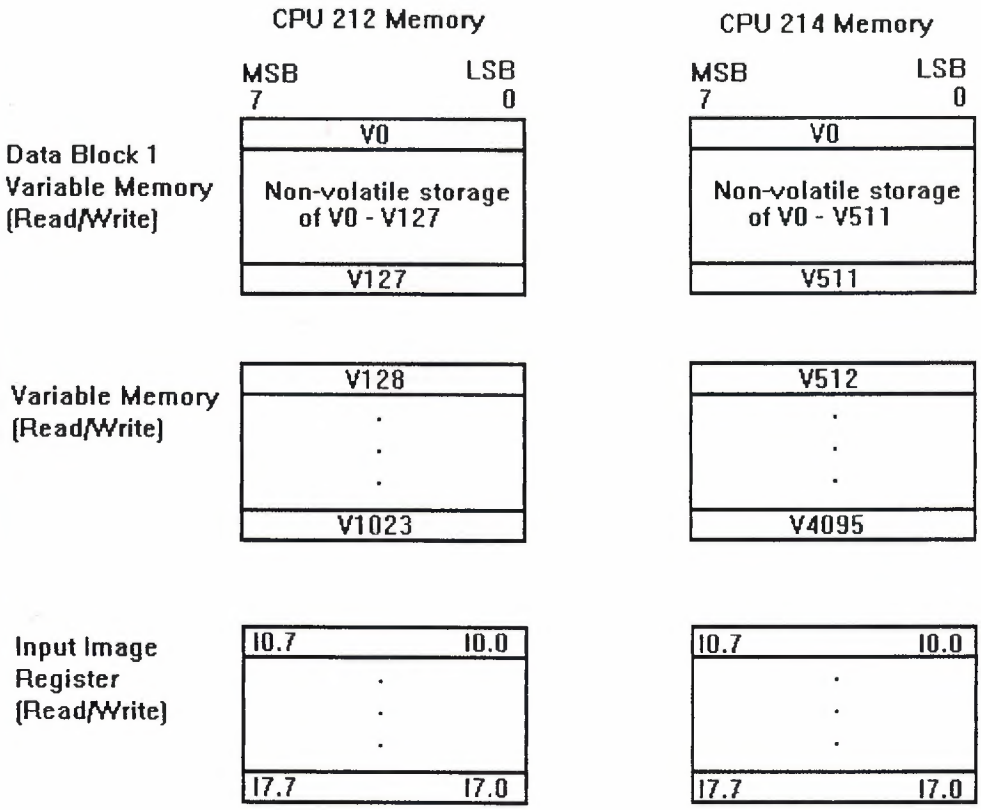
Data Objects. The S7-200 has six kinds of devices with associated data: timers, counters, analog inputs, analog outputs, accumulators and high-speed counters. Each device has associated data (data objects). For example, the S7-200 has counter devices. Counters have a data value that maintains the current count value. There is also a bit value which is set when the current value is greater than or equal to the preset value. Since there are multiple devices of each kind, devices are numbered from 0 to *n*. The corresponding data objects and object bits are also numbered.

The following table shows the identifiers and ranges for each of the data object memory types:

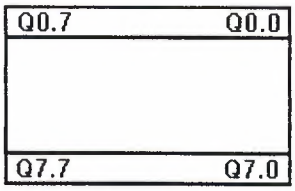
Object Identifier	Object	CPU 212	CPU 214
T	Timers	T0 to T63	T0 to T127
C	Counters	C0 to C63	C0 to C127
AI	Analog Input	AIW0 to AIW30	AIW0 to AIW30
AQ	Analog Output	AQW0 to AQW30	AQW0 to AQW30
AC	Accumulator Registers	AC0 to AC3	AC0 to AC3
HC	High-speed Counter Current	HC0	HC0 to HC2

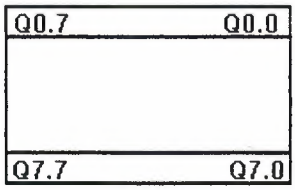
The programmable logic controller can also divide the memory space of the S7-200 into data areas identified by a symbolic name or data area name.

The table below shows memory space and data object spaces:

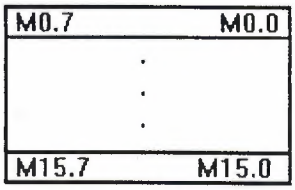


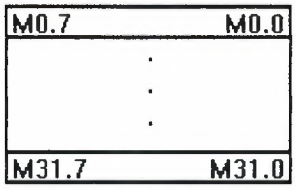
Output Image Register (Read/Write)



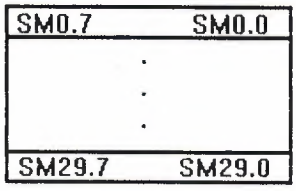


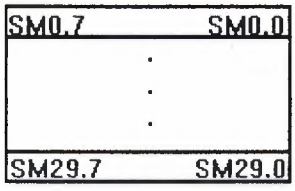
Internal Memory Bits (Read/Write)



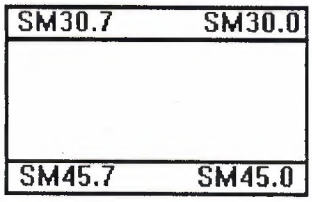


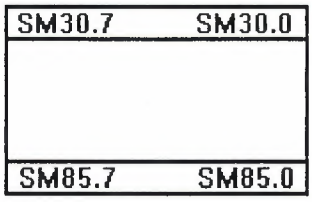
Special Memory Bits (Read Only)

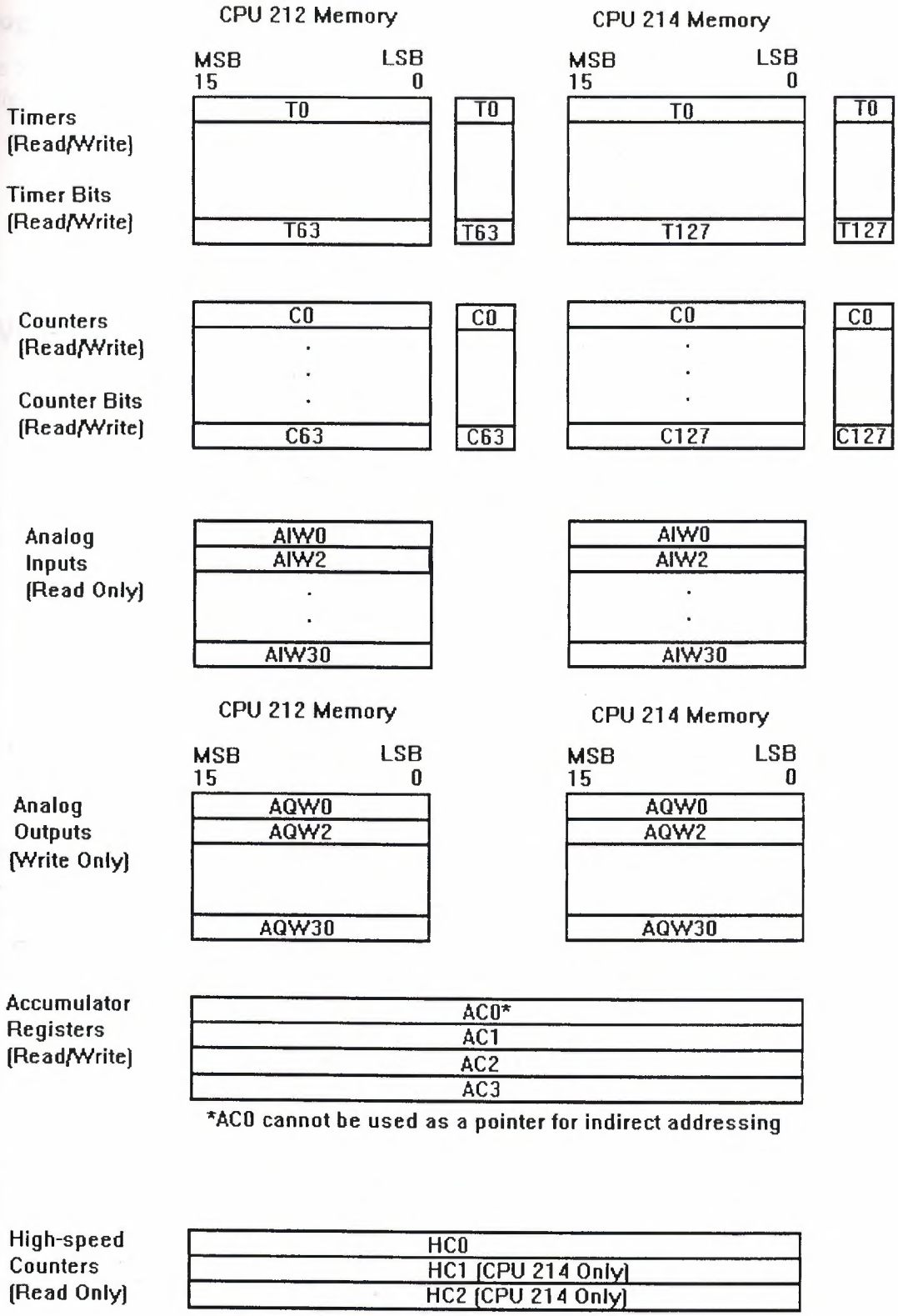




Special Memory Bits (Read/Write)

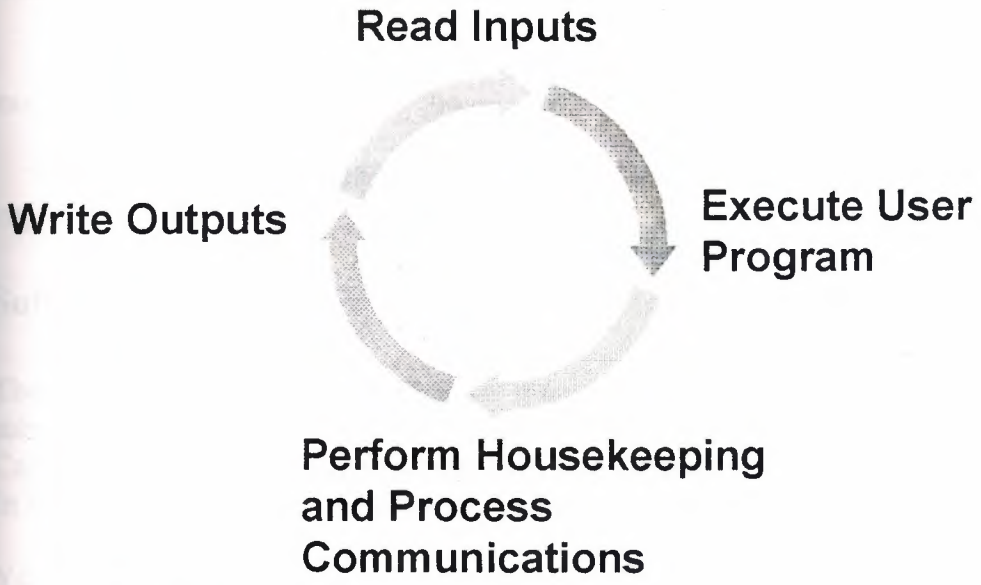





CPU 212 Memory
CPU 214 Memory
MSB
15
LSB
0
Timers
(Read/Write)
Timer Bits
(Read/Write)
T0
T63
T127
Counters
(Read/Write)
Counter Bits
(Read/Write)
C0
C63
C127
Analog
Inputs
(Read Only)
AIW0
AIW2
AIW30
Analog
Outputs
(Write Only)
AQW0
AQW2
AQW30
Accumulator
Registers
(Read/Write)
AC0*
AC1
AC2
AC3
*AC0 cannot be used as a pointer for indirect addressing
High-speed
Counters
(Read Only)
HC0
HC1 (CPU 214 Only)
HC2 (CPU 214 Only)

Program Execution Modes

The S7-200 normally executes your program in a cyclic fashion called a "scan". The basic scan cycle is as follows:



- Read Inputs and store in Input Image Register
- Execute the User's Program (updating the Output Image Register)
- Process Communication Requests
- Perform internal housekeeping (memory check, self-diagnostics, etc.)
- Write outputs from the Output Image Register

These actions are performed regularly and in sequential order. The CPU manages the scan cycle and also activates each task in the order that it must be performed. For information on "special" processing activities click on one of the following:

The S7-200 CPU can also perform "special handling" of interrupts and other high speed events. For details on these activities, just click on the desired topic:

Interrupt Processing

The SIMATIC S7-200 can respond to several types of interrupt events, including: Hardware Interrupts, Timed Interrupts, and Communication Interrupts.

An interrupt subroutine can be "attached" to selected discrete input points to create a "hardware interrupt routine". The PLC will interrupt it's normal scan cycle and execute this interrupt routine whenever it detects a change of state on that input point. When used in conjunction with the "immediate I/O instructions", hardware interrupts permit very high-speed reaction to emergency events. After the CPU completes the Interrupt routine it returns to the user program to resume normal processing.

Another unique feature of the S7-200 interrupt processing is the ability to dynamically attach an interrupt to more than one interrupt routine. This gives you more flexibility to process

interrupts where you may want to perform different actions on the same interrupt depending on where and when the interrupt occurs in your program.

Hardware Interrupts are executed when an input signal is received causing an interrupt routine to execute.

Timed Interrupts can be executed either on a specific date-time, or on a regular time interval (such as every 20 milliseconds).

Communication Interrupts are executed in conjunction with Freeport Mode for simple implementation of ASCII I/O.

Subroutines

The Subroutine Call instruction transfers program execution control to a subroutine. Once the subroutine completes its execution, control returns to the instruction that followed the original Call statement. Each subroutine must have a corresponding unconditional return instruction. In addition, you can have one or more conditional return instructions for added flexibility.

You can nest Subroutines to a depth of eight. Recursion (where a subroutine calls itself) is not prohibited, but use caution when using recursion with subroutines.

Jump Instruction

Jump instructions allow you to transfer control from one point of the program to another. Each jump instruction has a corresponding label. Both the jump and the label must be in the main program, or a subroutine or an interrupt routine. The Jump allows you to skip over a section of logic depending on the logic preceding the jump.

You cannot jump from the main program to a label in either a subroutine or interrupt routine. Likewise you cannot jump from a subroutine or interrupt routine to a label outside that subroutine or interrupt routine.

Another related instruction to the Jump is the FOR-NEXT loop. This instruction allows you to execute looping on a particular portion of code. This instruction is only supported in the CPU 214 but is extremely useful. FOR-NEXT instructions can be nested to a depth of 8 with a maximum loop count of 32,766!

Error Handling

The S7-200 programmable logic controller classifies errors as either fatal errors or non-fatal errors.

Fatal errors render the programmable logic controller incapable of executing the user program. Depending on the severity of the fatal error, it can render the PLC incapable of performing any or all functions. The objective of Fatal Error handling is to put the PLC into a safe state from which the PLC can respond to inquiries about the existing error conditions. Therefore, all fatal error conditions cause the PLC to transition to the STOP mode. The Fault

LED will turn on and the outputs are cleared. The PLC will remain in this condition until the fatal error condition is corrected. Some examples of Fatal Errors are:

Internal EEPROM failure
Internal EEPROM checksum failure
Internal Software Error
Memory cartridge failure (CPU 214 only)
User Program checksum failure
Scan watchdog timeout error

Non-Fatal errors can degrade some aspect of the PLC performance, but they do not render the PLC incapable of executing the user program and updating the I/O. All non-fatal errors detected in RUN mode are reflected in special memory bits where they are accessible by the user program. If you do not want to continue operation in the RUN mode with certain non-fatal error conditions, then your program can force a transition to STOP mode when this condition occurs. The decision to force a transition to STOP mode is left up to your discretion. Some examples of Non-Fatal Errors are:

Divide by Zero Error
Communication Parity Error
I/O Error
Timed Interrupt Queue overflow
Too many analog points
Run-time Programming Problem

Troubleshooting

To aid in debugging your program, other information associated with error conditions is stored in special areas of system-data memory. This information can then be accessed to determine what the problem was. The S7-200 also supports the following test functions to aid in detecting problems and in capturing important pieces of your data:

- ***Taking snapshots***. You can use the snapshot to capture the values from 1 to 8 user-data locations just after the PLC has executed a specified instruction (the CPU 214 supports 8

snapshots while the CPU 212 supports 1 snapshot). You can then use the snapshots to capture byte, word, or double word values of the data areas to determine whether your program is executing the way you think it is.

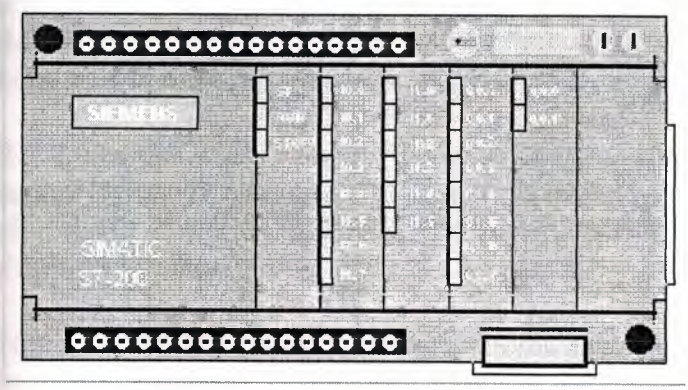
- ***Tracing***. The trace function captures the values from 1 to 8 user-defined locations at the end of each scan for up to 124 scans (the CPU 214 supports 8 traces while the CPU 212 supports 1 trace). You can use the trace function to capture byte, word or double word values of data areas to determine how a particular data location is being updated.

- ***Single and multiple scan execution***. The S7-200 supports the execution of 1 or more scans of the entire user program. You must put the PLC in the STOP mode to execute this function. Upon receiving the single/multiple scan command, the PLC transitions to the RUN mode for the specified number of scans and then returns to the STOP mode. This feature can be used in conjunction with the Trace and Snapshot feature to locate problems within the user program.

- ***Force Function***. The Force function is useful in the commissioning of your application. Some examples of its use follow. You can use the force function to override input status temporarily in order to debug your application logic. It can function just like an input simulator. You can use the force function to override discrete output points, variable memory and other data. Forcing outputs can assist in the debugging of the I/O wiring before startup. You can use the force function to skip portions of your program by enabling a jump instruction with a forced memory bit. This allows you to test individual sections of your program logic.

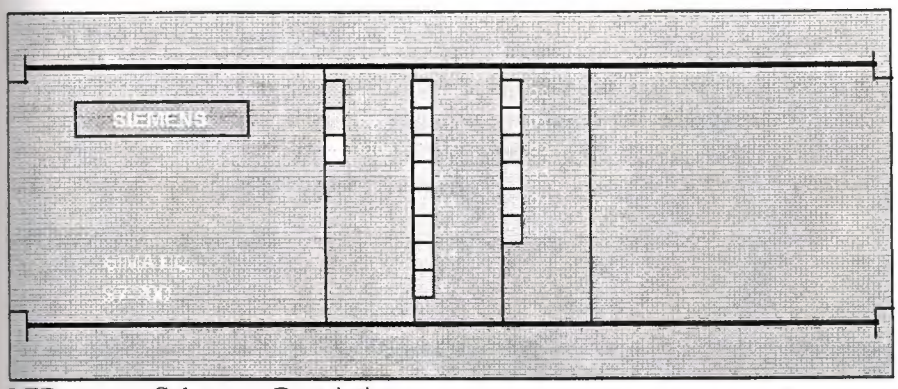
Hardware Features

The S7-200 base unit includes the Central Processing Unit (CPU), power supply and discrete input and output points. You can click on one of the topics below, or click on the picture to find out what hardware features are offered:



LEDs

There are five different types of status LEDs on the S7-200 series. These status LEDs describe both the current state of the base unit and the I/O points. The following table explains each type:



LED	Color	Description
SF	Red	System Fault - Indicates a System Fault. This LED lights up if the programmable controller has incurred a Fatal Error
RUN	Green	RUN Mode - Indicates that the PLC is in RUN mode and is executing its normal scan cycle
STOP	Yellow	STOP Mode - Indicates that the PLC is in STOP mode and that program execution has stopped
Ix.x	Green	The green input status LEDs indicates the current state of the input point. These are logic-side status indicators
Qx.x	Green	The green output status LEDs indicate the current state of the output point. These are logic-side status indicators.

Super Capacitor

The CPU 212 and CPU 214 programmable logic controllers have maintenance-free memory storage systems (EEPROM). They permanently store your logic program, force information, password, station number and output table information. This means that a battery is typically unnecessary. Since the program is stored on EEPROM, no battery is necessary to ensure that the program doesn't "evaporate into the bit bucket" if power is lost. A super capacitor has been used in the S7-200 series to provide for short-term storage of your data.

RAM memory is backed up by the power supply when it is turned on. When the power supply is turned OFF, the RAM memory is maintained by the super capacitor for a limited time after the PLC is powered down. The super capacitor typically maintains memory after power down for 190 hours in the CPU 214 and for 50 hours in the CPU 212 .

Not only does this eliminate the need for buying, stocking and changing batteries every 6 months, but this also eliminates the need to dispose of the lithium batteries which is environmentally friendly.

Mode Switch

The mode switch is located under the top access cover. This mode switch allows you to select the S7-200's operating mode. The following table describes the switch location and what it does:

Mode Position	Description
RUN	Causes the S7-200 CPU to execute your program
STOP	Causes the S7-200 CPU to stop program execution. The unit must be in STOP mode to allow your program to be edited
TERM	Terminal - allows the programming device to control transitions between RUN and STOP mode

Memory Module

Although the CPU 214 programmable logic controllers have an internal EEPROM to store your program, you also have the option to use an EEPROM memory cartridge. The memory cartridge is an optional device, and is not required for the programmable logic control to operate. The memory cartridge provides field upgrade capability of your program (bug fixes, additional functionality, etc.) without having to use a programming device or to transport a program from one PLC to another. This feature is especially useful for OEMs.

The programming of the memory cartridge is accomplished when you download the program to the PLC from a programming device and command the program to be copied to the memory cartridge. When the CPU 214 receives the command to copy the memory cartridge, it copies the following RAM data to the memory cartridge:

- User Program
- The first 512 bytes of the Data Area
- Station Address
- Retentive range definitions
- Freeze/Copy status and output table values for RUN to STOP transition
- Password and Restriction Class
- All forced operands and their values

When you power up the programmable logic controller with the memory cartridge installed, the contents of the memory cartridge are copied to the internal non-volatile memory. If you power up the PLC with a blank memory cartridge, the PLC will indicate a fault condition. You can install or remove the memory cartridge while the PLC is powered up.

The Memory Cartridge receptacle is located under the top access cover on the S7-214 CPU. The memory cartridge is keyed for proper installation and the receptacle is protected by a label. Remove and discard this label prior to installing the memory cartridge.

Analog Potentiometer

The S7-214 has two analog adjustments, located under the top access cover, to allow you to manually adjust a variable. Your program can then use this variable. The adjustment device permits 270 degrees of rotation and requires a small screwdriver to operate.

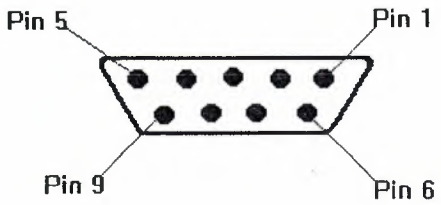
The CPU constantly monitors each of these adjustments and converts the position of the adjustment to a digital value within the range of 0 to 255. The values for these adjustments are stored in a special memory byte which is accessible by the user program. These locations are read-only locations so the value cannot be modified by the control program. To use these values from the analog adjustments, you must move the value to a read/write location where it can be scaled or limited. If scaling or limiting is not required, you can use the value as an input to any byte instruction.

Typically the values derived from these adjustments are used to program either timer or counter current or preset values, or to set limits, where minor operator adjustments are required.

Communication Port

The S7-200 series has a communication port located under the lower access cover on the right hand side of the CPU. The port uses a 9-pin Sub D connector that allows you to attach either a programming or an interconnecting cable. The baud rate for programming is 9600 baud. The baud rates supported by the PLC in Freeport mode are 300 to 38,400.

A pinout of the communications port is shown below:



Communication Port Pinout

Pin 1 = Logic Ground
Pin 2 = 24 VDC Return
Pin 3 = Transmit/Receive Data +
Pin 4 = Reserved
Pin 5 = Logic Ground
Pin 6 = 5 VDC (100 ohm Series R)
Pin 7 = 24 VDC (120 mA max)
Pin 8 = Transmit/Receive Data -
Pin 9 = Reserved

Connector Shell = Logic Ground

Panel Mounting Holes

You can panel-mount any of the S7-200 products using the two diagonally located mounting holes. These holes are located under the access covers, and accept either a DIN M4 or an American Standard number 8 screw. For proper mounting dimensions, refer to the section on Wiring and Mounting.

Field Wiring Connector

The wiring of your inputs and outputs are connected to the field wiring connectors, located under the upper and lower access covers. The wiring for the unit's power supply and DC sensor supply is also accomplished by connecting to the field wiring connectors. Labeling of the connector is located on the housing under the access covers. You can also apply terminal-identification labels, supplied with the unit, to the inside or outside of the access cover for easy identification of your field wiring.

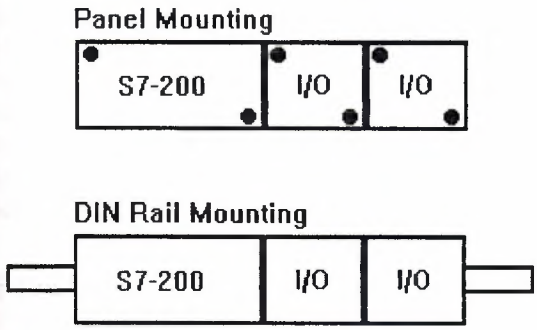
Field Wiring connector drawings are included within this help file under "Specifications".

Bus Expansion Port

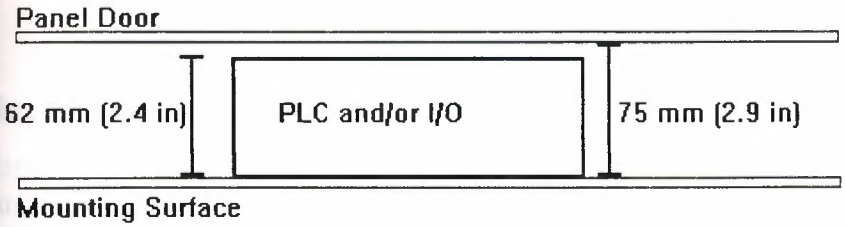
The Bus Expansion Port allows additional I/O expansion modules to be connected. The port is protected by a break-away cover located on the right-hand side of the CPU. This cover can be easily removed with a screwdriver, but once removed, it cannot be replaced. After the cover has been removed, expansion I/O modules can be attached to the CPU unit using an I/O bus connector which is supplied with the Expansion Module.

Mounting

The S7-200 Series can be either panel or DIN rail mounted. Vertical mounting is also possible. Mounting is accomplished as shown below:



Panel mounting can be accomplished with a minimum depth of at least 75 mm (dimensions are shown below):



If you plan to install additional modules, allow extra space of at least 25 mm (1 inch) on either side of the unit for installing and removing the module. This extra space is required to engage and disengage the bus expansion connector.

The S7-200 series is designed for natural convection cooling. You must provide a clear space of at least 65 mm (2.5 inches), both above and below the units for proper cooling.

-Programmable Parameters

The S7-200 Series has made it very easy to configure parameters. In addition to automatic I/O configuration, you can configure the following parameters through the programming software or the handheld programmer. **Retentive Memory**

You can define up to six retentive ranges to select the areas of memory you want to retain through power cycles. A retentive range is a programmable specification for an area of memory designating a from-to range. The range is not cleared after the S7-200 is powered up, provided that the super capacitor is able to maintain the contents of RAM.

Not all the data areas residing in RAM can be defined as retentive. The data areas that can be defined as retentive are V, M, T (T0 to T31 and T64 to T95), and C.

The following Table shows the default settings for the retentive ranges.

Retentive Range	CPU 212	CPU 214
Retentive Range 0	V0-V1023	V0-V4095
Retentive Range 1	Not used	Not used
Retentive Range 2	T0-T31	T0-T31
Retentive Range 3	Not used	T64-T95
Retentive Range 4	C0-C63	C0-C127
Retentive Range 5	M0-M15	M0-M31

CPU Clock

The CPU 214 provides a real-time clock which can be set and read by way of communication functions and your program instructions. The PLC indicates that the clock has not been set, or that the setting was lost due to the discharge of the super capacitor by initializing the time-of-day clock to:

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

The date and time values are coded in BCD. The date and time format is shown below:

Year / Month	yymmm	yy=0 - 99, mmm=1 - 12
Day / Hour	ddhh	dd=1 - 31, hh=0 - 23
Minute / Second	mmss	mm=0 - 59, ss=0 - 59
Day of week	000d	d=1 - 7 where 1 means Sunday

Hardware Interrupts

Hardware Interrupts allow you to perform high-speed processing based upon an event. Interrupt processing provides quick reaction to special external events. I/O interrupts include rising/falling edge interrupts, high-speed counter interrupts, and pulse train output interrupts.

In the CPU 214, I0.0 through I0.3 can generate an interrupt on rising and/or falling edges. In the CPU 212, I0.0 can generate an interrupt on rising and/or falling edges. The rising 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.

The high-speed counter interrupts allow you to respond to such conditions as the current value reaching the preset value, a change in counting direction that might correspond to a reversal in the direction a shaft is turning, and 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.

Communication Interrupts

The serial communication port of the programmable logic controller can be controlled by the user 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 communication.

In the simplest case, you can send a message to a printer or display using only the Transmit function. 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 to which the programmable logic controller communicates while in Freeport mode.

Cyclic Interrupts

The CPU 214 supports two timed interrupts, and the CPU 212 supports one timed interrupt. 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.

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.

Passwords and Protection Levels

Authorized access to the PLC functions and memory is provided through the use of a password. Without a password, the PLC provides unrestricted access. When password protected, the PLC prohibits all restricted operations, according to the configuration provided when the password was installed.

Authorization of the password is accomplished through communication between the programming device and the PLC. Since multiple devices could potentially be connected to the PLC, authorization of the password is tied to the device through which authorization was granted. An authorized user may co-exist with unauthorized users, and only one authorized user is given unrestricted access to the PLC functions. All other users must live with the restrictions assigned by the person who knows the password. Changes to the password and to the access restrictions may only be made by the authorized user who knows the password.

There are three protection levels available to you. The access restrictions are outlined in the following table:

Communication Function	Restriction Class 1	Restriction Class 2	Restriction Class 3
Read user data	Allowed	Allowed	Allowed
Write user data	Allowed	Allowed	Allowed
Start/Stop the execution of the user program	Allowed	Allowed	Allowed
Restart the PLC	Allowed	Allowed	Allowed
* Set/Read the Time-of-Day Clock	Allowed	Allowed	Allowed
Test (Functionality: Read)	Allowed	Allowed	Not Allowed
Upload user program, data and configuration	Allowed	Allowed	Not Allowed
Load user program, data and configuration	Allowed	Not Allowed	Not Allowed
Delete user program, data and configuration	Allowed	Not Allowed	Not Allowed
Test (Functionality: Write, Change, or Modify)	Allowed	Not Allowed	Not Allowed
* Copy user program, data and configuration data to a memory cartridge	Allowed	Not Allowed	Not Allowed

Note: * CPU 214 only

In the event the password is forgotten, use the master password **CLEARPC** to gain access to the programmable controller. You must have the PLC Mode Switch in either the STOP or TERM position. When you use the master password, the PLC performs the following:

- The PLC enters the STOP mode
- The user program is deleted
- The user data memory is deleted and all data space is cleared
- All configuration parameters except the station address are deleted
- All memory bits are cleared
- All special memory bits are set to their default state

- Analog outputs are frozen
- All system data memory is set to the default state
- All forced points are cleared and unforced
- The time of day clock is not changed (CPU 214)
- All timer/counter current data is cleared

Overview

The S7-200 series has some of its I/O built-in to the base unit. The I/O points on the CPU are numbered to provide you with unique identification in your program. There are two types of base units in the S7-200 product line. The specifications for both the CPU and the Expansion I/O modules are included in this help file:

The CPU 214 has 14 inputs and 10 outputs on the base unit and can be expanded to a total of 64 discrete I/O points.

The CPU 212 has 8 inputs and 6 outputs on the base unit, and can be expanded to a total of 30 discrete I/O points.

The base unit maintains an image of the I/O points in a memory area called the image register. There is an input image register and an output image register. There are 3 major reasons for the image register:

1. The sampling of all inputs at the top of the scan synchronizes and freezes the values of the inputs for the program execution phase of the scan cycle. The outputs are updated from the image register after the execution of the program is complete. This provides a stabilizing effect on the system being controlled.

2. You can access the image register much quicker than you can access I/O points. This allows faster execution of the program being executed.

3. I/O points are bit entities and must be accessed as bits, however, you can access the image register as bits, bytes, words, or double word values. Thus, the image registers provide the user with additional flexibility.

Expansion I/O provides you the means to expand the number of I/O points if needed. This keeps costs down so that you don't pay for I/O that you don't need. Expansion of I/O is possible on both the CPU 212 and the CPU 214.

There is some specialty I/O which is also built-in to the CPU unit. This specialty I/O consists of High Speed Counter operations, Pulse Output operations, and the ability to directly access physical I/O points.

Immediate I/O

The immediate I/O instructions give you the ability to circumvent the scan cycle with respect to reading inputs and writing outputs. This is extremely useful for fast response to interrupt events.

During a normal scan cycle, the inputs are read and then stored in the input image register. The program then references the input image register when executing the user's program. However, if you want to read the actual value of the input you must use an immediate I/O access instruction. Likewise, if you want to update an output before the end of the current scan, you must use an immediate I/O access instruction.

Immediate I/O instructions allow direct access to the actual input or output point, even though the image registers are normally used as either the source or destination for I/O access. The corresponding input image register location is not modified when you use an immediate instruction to access an input point. However, the corresponding output image register location is updated simultaneously when you use an immediate instruction to access an output point. Also, access to Analog I/O is always immediate.

High Speed Counter

High-speed counters count high-speed events that cannot be controlled at programmable logic controller scan rates. The CPU supports one high-speed counter and the CPU 214 supports three high-speed counters.

The high-speed counters in the CPU 214 are called HSC0, HSC1 and HSC2. The CPU 212 high-speed counter is called HSC0. HSC0 is an up/down 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.

Modes of Operation

In the CPU 214, HSC1 and HSC2 are versatile counters that can be configured for one of twelve different modes of operation. Each counter has dedicated inputs for clocks, direction control, reset, and start where these functions are supported. The maximum clock input frequency for HSC1 and HSC2 is 7 kHz. For the two phase counters, both clocks may run at 7 kHz rates. In quadrature modes, an option is provided to select 1x or 4x counting rates. At 1x rate, the maximum counting rate is 7 kHz and, at 4x rate the maximum counting rate is 28 kHz. HSC1 and HSC2 are completely independent of one another and do not affect other high-speed functions. Both counters run at maximum rates without interfering with one another.

Pulse outputs

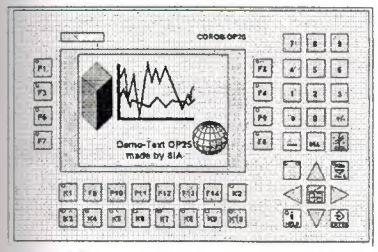
The CPU 214 allows Q0.0 and Q0.1 either to generate high-speed pulse train outputs (**PTO**) or to perform pulse width modulation (**PWM**) control.

The **PTO** 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 to 65,535 microseconds or from 2 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 to 65,535 microseconds or from 2 to 65,535 milliseconds. The pulse width timer 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 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.

-Operator Interfaces

The combination of Siemens Micro-PLCs and the COROS® line of operator panels provide you with powerful and cost effective control and monitoring for many processes. As the level of automation increases for both machines and systems, the need for operators to monitor and modify automated processes increases as well.

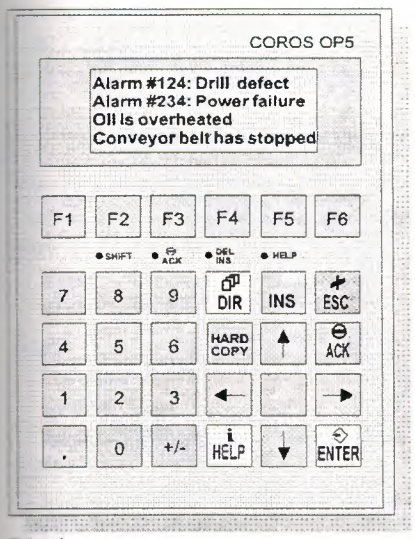


A variety of features are provided to you with these displays such as:

9 Levels of password protection with multiple passwords per user level
Extensive recipe management that allows you to store up to 255 recipes
Easy to use configuration software
Event messages generated by a bit within the PLC
Alarm messages with time stamping, and operator acknowledgment
Automatic logging of the last 256 event and alarm messages received
Plus an on-line help capability to document an event or alarm message

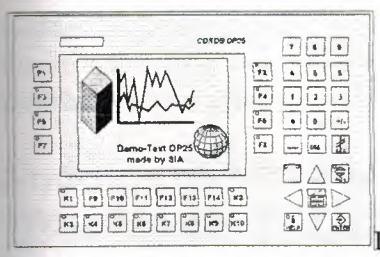
The full line of COROS operators - from the small, text based OP5 to the larger (but only 2" thin) OP35 with a color graphics screen and disk drive options, are flexible, reliable and dependable.

OP5



Display	
Type	Backlit LCD
Size	4 lines x 20 characters
Keys	
Function Keys	6
System Keys	24
Alpha-Numeric	Numeric
LEDs	
System	4
User Defined	0
Memory	
Type	Flash EPROM
Size	128 KB
Number of Screens	99
Entries per Screen	99
Messages	
Number	499
Alarm	499
Password	9 Levels
Physical (H x W x D)	
Dimensions - Mounting	6.2 x 4.3 x 1.6 in
Dimensions - Front	6.6 x 4.7 x 1.6 in
Environmental	
Ratings	IP65
Approvals	UL / CSA Pending
Operating Temperature	0 - 45° C
Storage Temperature	-10 to 60° C

OP25



Display	
Type	Backlit LCD
Size	320 x 240 pixels
Keys	
Function Keys	24
System Keys	24
Alpha-Numeric	Full Alpha-Numeric
LEDs	
System	4
User Defined	24
Memory	
Type	Flash EPROM
Size	1 MB
Number of Screens	Memory
Entries per Screen	N/A
Messages	
Number	2000
Alarm	2000
Password	9 Levels
Physical (H x W x D)	
Dimensions - Mounting	7 x 11.1 x 2.3 in
Dimensions - Front	7.6 x 11.6 x 2.3 in
Environmental	
Ratings	IP65
Approvals	UL / CSA Pending
Operating Temperature	0 - 45° C
Storage Temperature	-10 to 60° C

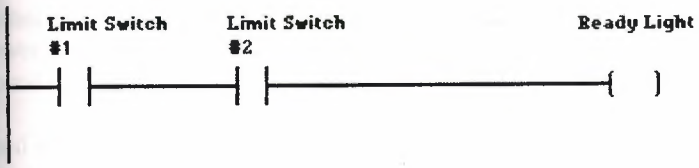
-Programming

There are two methods of programming an S7-200 PLC: Ladder Logic and Statement List. You also have the ability to switch back and forth between the two to provide you with a programming environment which is as easy as pointing and clicking.

Ladder Logic Programming

Ladder Logic is a graphic representation of the S7-200's programming language. Its syntax for the instructions is much like a relay ladder logic diagram: the ladder program tracks power flow between power rails as it passes through various inputs, outputs, and other instructions.

A typical example of a Ladder Logic programming methodology follows. The problem: If the input called 'Limit Switch #1' and the input called 'Limit Switch #2' both turn ON, then turn ON the output called 'Ready Light'. Using the device name instead of the PLC address is an example of a Symbolic Name



Statement List Programming

Statement List is a textual representation of the S7-200's programming language. Its syntax for the instructions is much like assembly language programming: there is a command or operation, followed by a memory address.

A typical example of a statement list programming methodology follows. The problem: If the input called 'Limit Switch #1' and the input called 'Limit Switch #2' both turn ON, then turn ON the output called 'Ready Light'. In statement list, we are using the absolute I/O address instead of symbolic names.

```
A  I0.0     // If Limit Switch #1 is ON
A  I0.1     // And Limit Switch #2 is ON
O  Q0.0     // Turn ON the output called Ready Light
```

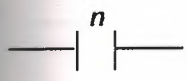
Instruction Set

The S7-200 has over 120 powerful and extensive instructions. The instruction set contains operations that you would not expect from a Micro-PLC controller.

Ladder Instruction Set

Normally Open Contact

Symbol:



Operands:

n (bit): I, Q, M, SM, T, C, V

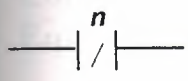
Description of operation:

The Normally Open Contact is closed when the scanned bit value stored at address n is equal to 1 . Power flows through a normally open contact when closed (activated).

Used in series, a normally open contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Normally Closed Contact

Symbol:



Operands:

n (bit): I, Q, M, SM, T, C, V

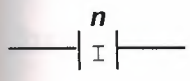
Description of operation:

The Normally Closed Contact is closed when the bit value stored at address n is equal to 0 . Power flows through the contact when closed (deactivated).

Used in series, a normally closed contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Normally Open Immediate Contact

Symbol:



Operands:

n (bit): I

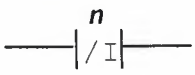
Description of operation:

The Normally Open Immediate Contact is closed when the Bit value stored at address n is equal to 1 . Power flows through the contact when closed (activated). A physical input read occurs immediately after the coil is scanned without waiting for scan cycle completion. The image register is not updated.

Used in series, a normally open immediate contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Normally Closed Immediate Contact

Symbol:



Operands:

n (bit): I

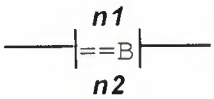
Description of operation:

The Normally Closed Immediate Contact is closed when the Bit value stored at address n is equal to 0 . Power flows through the contact when closed (deactivated). A physical input read occurs immediately after the coil is scanned without waiting for scan cycle completion. The image register is not updated.

Used in series, a normally closed immediate contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Compare Byte Equal Contact

Symbol:



Operands:

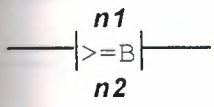
n1, n2 (unsigned byte): VB, IB, QB, MB, SMB, Constant, *VD, *AC

Description of operation:

The Compare Byte Equal Contact is closed when the byte value stored at address n1 is equal to the byte value stored at address n2 . Power flows through the contact when closed.

Compare Byte Greater Than Or Equal Contact

Symbol:



Operands:

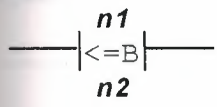
n1, n2 (unsigned byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Compare Byte Greater Than or Equal Contact is closed when the byte value stored at address n1 is greater than or equal to the byte value stored at address n2 . Power flows through the contact when closed.

Compare Byte Less Than Or Equal Contact

Symbol:



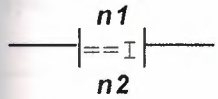
Operands:

n1, n2 (unsigned byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Compare Byte Less Than or Equal Contact is closed when the byte value stored at address n1 is less than or equal to the byte value stored at address n2 . Power flows through the contact when closed.

Compare Integer Equal Contact

Symbol:



Operands:

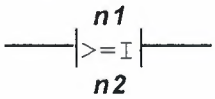
n1, n2 (signed integer word):	VW, T,C,IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Compare Integer Equal Contact is closed when the signed integer word value stored at address n1 is equal to the signed integer word value stored at address n2 . Power flows through the contact when closed.

Compare Integer Greater Than Or Equal Contact

Symbol:



Operands:

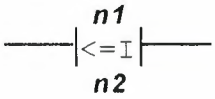
n1, n2 (signed integer word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Compare Integer Greater Than or Equal Contact is closed when the signed integer word value stored at address n1 is greater than or equal to the signed integer word value stored at address n2 . Power flows through the contact when closed.

Compare Integer Less Than Or Equal Contact

Symbol:



Operands:

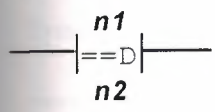
n1, n2 (signed integer word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Compare Integer Less Than or Equal Contact is closed when the signed integer word value stored at address n1 is less than or equal to the signed integer word value stored at address n2 . Power flows through the contact when closed.

Compare Double Integer Equal Contact

Symbol:



Operands:

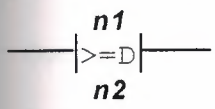
n1, n2 (signed integer double word):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The Compare Double Integer Equal Contact is closed when the double word value stored at address n1 is equal to the double word value stored at address n2 . Power flows through the contact when closed.

Compare Double Integer Greater Than Or Equal Contact

Symbol:



Operands:

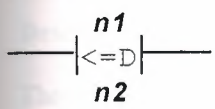
n1, n2 (signed integer double word):	VD, ID, QD, MD, SMD,AC HC, Constant, *VD, *AC

Description of operation:

Compare Double Integer Greater Than Or Equal Contact is closed when the double word value stored at address n1 is greater than or equal to the double word value stored at address n2 . Power flows through the contact when closed.

Compare Double Integer Less Than Or Equal Contact

Symbol:



Operands:

n1, n2 (signed integer double word):	VD, ID, QD, MD, SMD,AC, HC, Constant, *VD, *AC

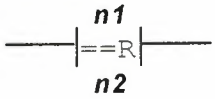
Description of operation:

The Compare Double Integer Less Than Or Equal Contact is closed when the double word value stored at address n1 is less than or equal to the double word value stored at address n2 . Power flows through the contact when closed.

Compare Real Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

n1, n2 (real):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

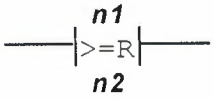
Description of operation:

The Compare Real Equal Contact is closed when the real value stored at address n1 is equal to the real value stored at address n2 . Power flows through the contact when closed.

Compare Real Greater Than Or Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

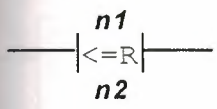
Description of operation:

Compare Real Greater Than Or Equal Contact is closed when the real value stored at address n1 is greater than or equal to the real value stored at address n2 . Power flows through the contact when closed.

Compare Real Less Than Or Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The Compare Real Less Than Or Equal Contact is closed when the real value stored at address n1 is less than or equal to the real value stored at address n2 . Power flows through the contact when closed.

Invert Power Flow Contact

Symbol:

Operands:

(none)

Description of operation:

The NOT (Invert Power Flow) contact changes the state of power flow. If power flow reaches the Not contact, then it stops. When power flow does not reach the Not contact, it sources power flow.

Positive Transition Contact

Symbol:

Operands:

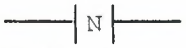
(none)

Description of operation:

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

Negative Transition Contact

Symbol:



Operands:

(none)

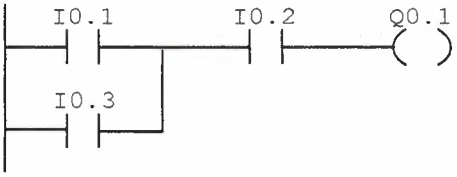
Description of operation:

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

Ladder Contact Examples

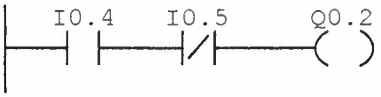
Network 1

When I0.1 or I0.3 is on and I0.2 is on then output Q0.1 is turned on.



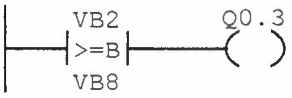
Network 2

When I0.4 is on and I0.5 is not on, then output Q0.2 is turned on.



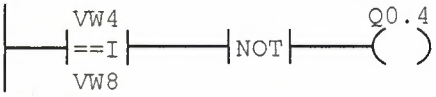
Network 3

When VB2 is greater than or equal to VB8, then output Q0.3 is turned on.



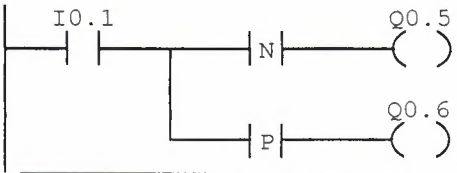
Network 4

When VB4 equals VB8, then output Q0.4 is turned off *(Note: The NOT instruction can be used to create a Not Equal comparison.)*



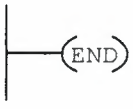
Network 5

When I0.1 transitions from on to off, then output Q0.5 is turned on for one scan cycle. When I0.1 transitions from off to on, then Q0.6 is turned on for one scan.



Network 6

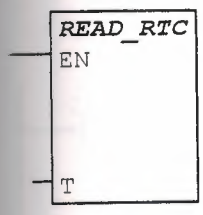
End of the main user program.



Read Real Time Clock

Note: Real Time Clock instructions are supported by the CPU 214 only.

Symbol:



Operands:

T (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:

The Read Real Time Clock (READ_RTC) box reads the current time and date from the clock and loads it in an 8-byte buffer (T).

Example Memory Data Starting at VB400:

READ_RTC (Clock is read)

VB400	95	Year
VB401	03	Month
VB402	24	Day
VB403	08	Hour
VB404	00	Minute
VB405	00	Second
VB406	00	
VB407	06	Day of Week

24-Mar-95
8:00:00
Friday

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

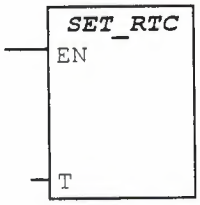
Date:	01-Jan-90
Time:	00:00:00
Day of Week	Sunday

Note:
Do not use the READ_RTC / SET_RTC instructions in both the main program and in an interrupt routine. If you do this and the clock instruction is executing when the the interrupt that also executes the clock instruction occurs, then the clock instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted.

Set Real Time Clock

Note: Real Time Clock instructions are supported by the CPU 214 only.

Symbol:



Operands:

T (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:

The Set Real Time Clock (SET_RTC) box writes the current time and date loaded in an 8-byte buffer (T) to the clock.

Example Memory Data Starting at VB400:

SET_RTC (New value is written to clock)

VB400	96	Year
VB401	03	Month
VB402	24	Day
VB403	08	Hour
VB404	00	Minute
VB405	00	Second
VB406	00	
VB407	06	Day of Week

24-Mar-96
8:00:00
Friday

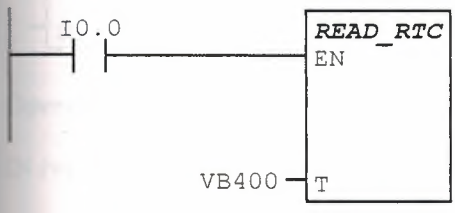
Note:
The time of day clock initializes the following date and time after extended power outages or memory has been lost:

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

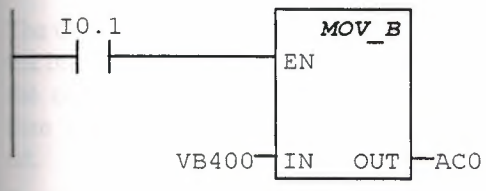
Note:
Do not use the READ_RTC / SET_RTC instructions in both the main program and in an interrupt routine. If you do this and the clock instruction is executing when the the interrupt that also executes the clock instruction occurs, then the clock instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted.

Real-time Clock Instruction Examples

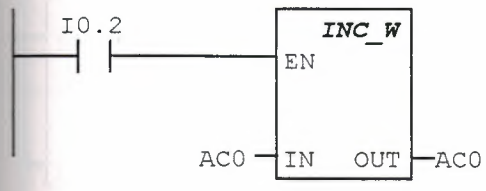
Network 1 — When I0.0 is on, the clock is read and the value is stored in the buffer, starting at VB400.



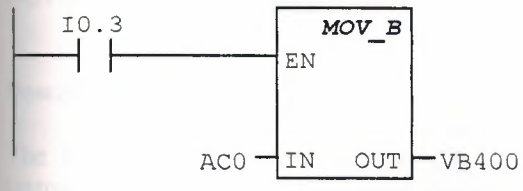
Network 2 — When I0.1 is on, the year value (95) from the first byte of VB400 is moved to AC0 .



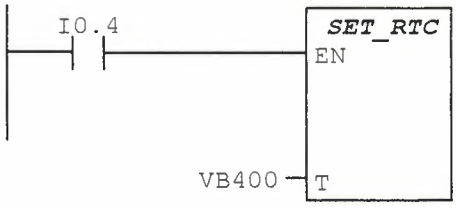
Network 3 — When I0.2 is on, the year value in AC0 is incremented by 1.



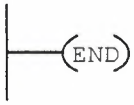
Network 4 — When I0.3 is on, the new year value (96) is stored in VB400 .



Network 5 — When I0.4 is on, the new year value is written to the clock.

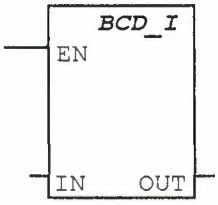


Network 6 — End of the main user program.



BCD to Integer

Symbol:



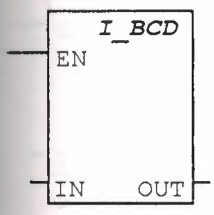
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert BCD to Integer (BCD_I) box converts the BCD value (IN) to an integer value (OUT). If the input value contains an invalid BCD digit, the BCD/BIN memory bit (SM1.6) is set.

Integer to BCD

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

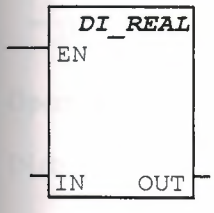
Description of operation:

The Convert Integer to BCD (I_BCD) box converts the integer value (IN) to the BCD value (OUT). If the conversion produces a BCD number greater than 9999, the BCD/BIN memory bit (SM1.6) is set.

Integer Double Word to Real

Note: CPU 214 only.

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

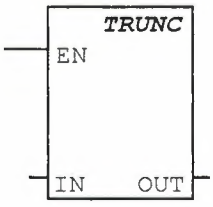
Description of operation:

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

Truncate

Note: CPU 214 only.

Symbol:



Operands:

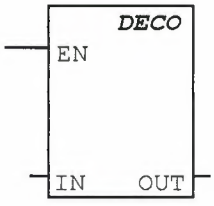
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Truncate (TRUNC) 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).

Decode

Symbol:



Operands:

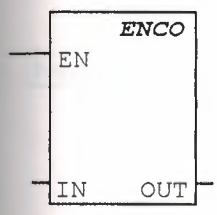
IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:

The Decode (DECO) box sets the bit in the output word (OUT) that corresponds to the bit number represented by the least-significant nibble (LSN) of the input byte (IN). All other bits of the output word are set to 0.

Encode

Symbol:



Operands:

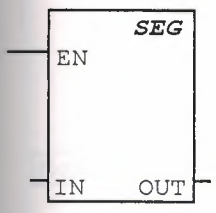
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:

The Encode (ENCO) box writes the bit number (bit #) of the least-significant bit set of the input word (IN) into the least-significant nibble (LSN) of the output byte (OUT).

Segment

Symbol:



Operands:

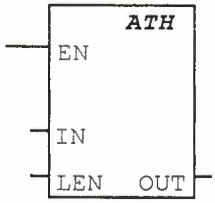
IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:

The Segment (SEG) box 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).

ASCII to Hex

Symbol:



Operands:

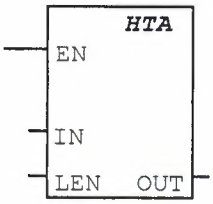
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:
The ASCII to HEX (ATH) box converts the ASCII string of length LEN, starting with the character IN, to hexadecimal digits starting at the location OUT. The maximum length of the ASCII string is 255 characters.

Legal ASCII characters are the hexadecimal values 30-39, and 41-46. If an illegal ASCII character is encountered, the conversion is terminated, and the NOT_ASCII memory bit (SM1.7) is set.

Hex to ASCII

Symbol:



Operands:

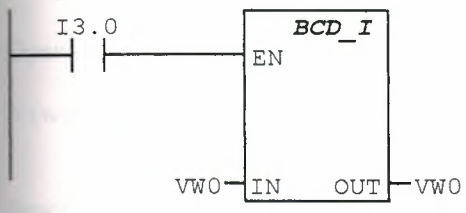
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:
The HEX to ASCII (HTA) box converts the hexadecimal digits, starting with the input byte IN, to an ASCII string starting at the 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.

Ladder Conversion Instruction Examples

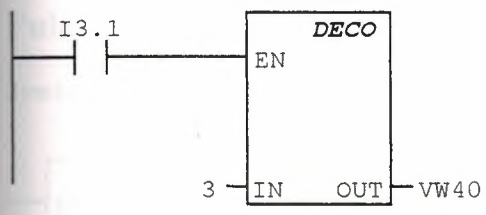
Network 1

When I3.0 is on, the Binary Coded Decimal value in VW0 is converted to an integer value.



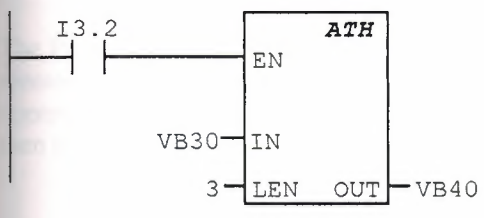
Network 2

When I3.1 is on, 3 is decoded and the corresponding bit of VW40 is set.



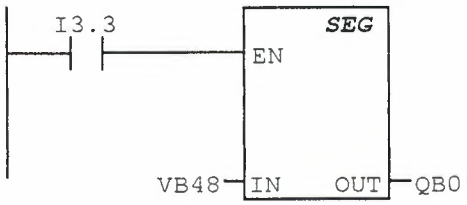
Network 3

When I3.2 is on, the 3-character ASCII string starting with the character at VB30 is converted to hexadecimal digits starting at VB40.



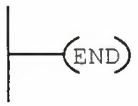
Network 4

When I3.3 is on, a bit pattern is generated at QB0 that illuminates the segments of the character represented by VB48.



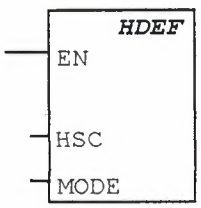
Network 5

End of the main user program.



HSC Definition

Symbol:



Operands:

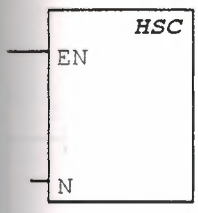
HSC (byte):	CPU 212: 0 CPU 214: 0-2
MODE (byte):	CPU 212: 0 CPU 214: 0 (HSC0), 0-11 (HSC1-2)

Description of operation:

When the High-speed Counter Definition (HDEF) box is enabled, the referenced counter (HSC) is assigned a high-speed counter type or MODE. Only one HDEF box may be used per counter.

High Speed Counter

Symbol:



Operands:

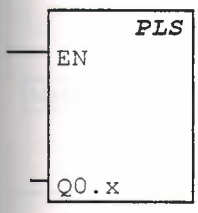
N (word):	CPU 212: 0 CPU 214: 0-2

Description of operation:

When the High-speed Counter (HSC) box is enabled, the state of the HSC special memory bits are examined. The HSC operation defined by the special memory bits is then invoked. The parameter N specifies the High-speed Counter number.

Pulse Output

Symbol:



Operands:

Q0.x (word):	CPU 214: 0-1

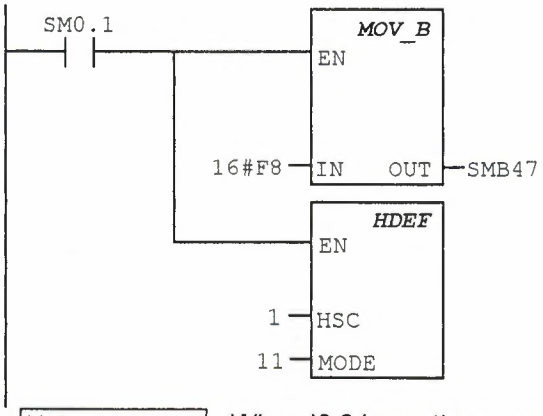
Description of operation:

The Pulse Output (PLS) box examines the special memory bits for that pulse output (Q0.x). The pulse operation defined by the special memory bits is then invoked.

Ladder High-speed Operation Instruction Examples

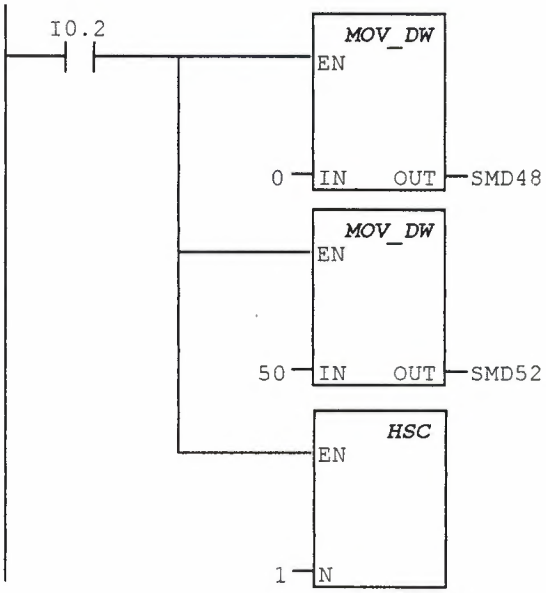
Network 1

On the first scan, the counter is enabled. Initial direction is set to count up. Start and reset inputs are set to active high. 4x mode is set.

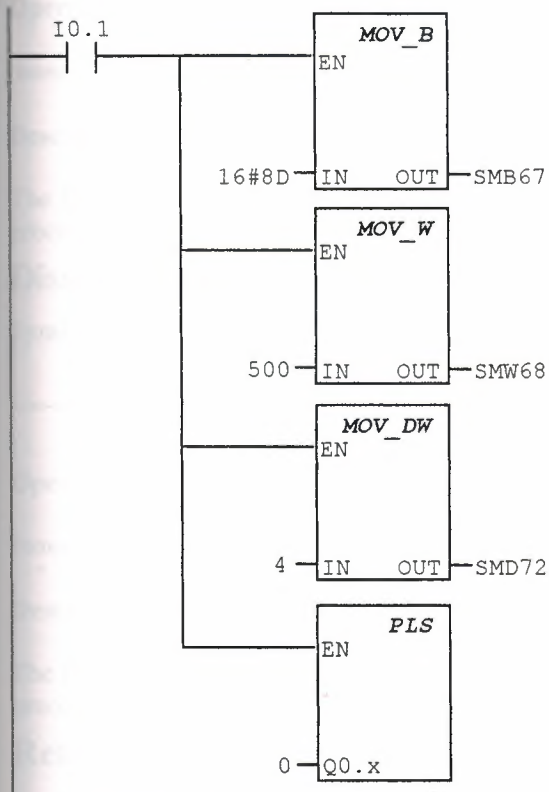


Network 2

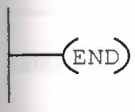
When I0.2 is on, the current value of HSC1 is cleared and its preset value is set to 50 .



Network 3 — When I0.1 is on, the Pulse Train Output control byte is set up, and the PTO operation is invoked: cycle time 500ms, pulse count 4, PLS 0 --> Q0.0 .

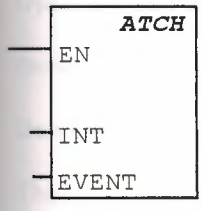


Network 4 — End of the main user program.



Attach Interrupts

Symbol:



Operands:

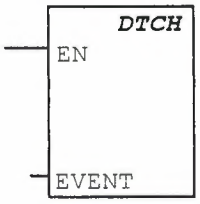
INT (byte):	CPU 212: 0-31 CPU 214: 0-127
EVENT (byte):	CPU 212: 0, 1, 8-10, 12 CPU 214: 0-20

Description of operation:

The Attach Interrupts (ATCH) box associates an interrupt event (EVENT) with an interrupt routine number (INT), and enables the interrupt event.

Detach Interrupts

Symbol:



Operands:

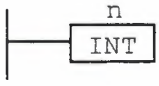
EVENT (byte):	CPU 212: 0, 1, 8-10, 12 CPU 214: 0-20

Description of operation:

The Detach Interrupts (DTCH) box disassociates an interrupt event (EVENT) from all interrupt routines, and disables the interrupt event.

Interrupt Routine

Symbol:



Operands:

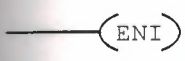
n (word):	CPU 212: 0-31 CPU 214: 0-127

Description of operation:

The Interrupt Routine (INT) label marks the beginning of the interrupt routine (n). The maximum number of interrupts supported by the CPU 212 is 32, and by the CPU 214, 128.

Enable Interrupts

Symbol:



Operands:

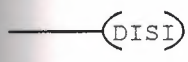
(none)

Description:

The Enable Interrupts (ENI) coil globally enables processing of all attached interrupt events.

Disable Interrupts

Symbol:



Operands:

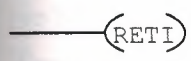
(none)

Description:

The Disable Interrupts (DISI) coil globally disables processing of all interrupt events.

Return from Interrupts

Symbol:



Conditional Return from Interrupts

Unconditional Return from Interrupts

Operands:

(none)

Description:

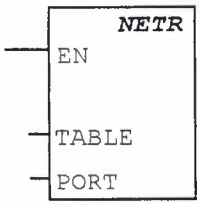
The Conditional Return from Interrupts (RETI) coil returns from an interrupt based upon the condition of the preceding logic.

The Unconditional Return from Interrupts (RETI) coil must be used to terminate each interrupt routine.

Network Read

Note: CPU 214 only.

Symbol:



Operands:

TABLE:	VB, MB, *VD, *AC
PORT:	Constant (CPU 214: 0)

Description of operation:

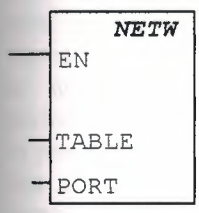
The Network Read (NETR) instruction initiates a communication operation to gather data from a remote device through the specified port (PORT), as defined in the description table (TABLE).

You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to 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 NETR and four NETW instructions, or two NETR and six NETW instructions.

Network Write

Note: CPU 214 only.

Symbol:



Operands:

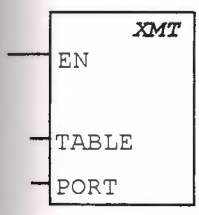
TABLE:	VB, MB, *VD, *AC
PORT:	Constant (CPU 214: 0)

Description of operation:
The Network Write (NETW) instruction initiates a communication operation to write data to a remote device through the specified port (PORT), as defined in the description table (TABLE).

You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to 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 NETR and four NETW instructions, or two NETR and six NETW instructions.

Transmit

Symbol:



Operands:

TABLE (byte):	VB, IB, QB, MB, SMB, *VD, *AC
PORT (byte):	0

Description of operation:
The Transmit (XMT) box 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. It must always be 0.

Data Sharing with Interrupt Events

Because interrupt events are asynchronous to the main user-program, they can occur at any point during execution of the main user-program. When the main program and an interrupt routine share data, you must understand the nature of the problems that can arise and how to avoid such problems.

Data-sharing problems can occur in situation where a sequence of operations are performed in the main program on data stored in a memory location shared by the main program and an interrupt routine. If an intermediate result is stored in the shared memory location, then an interrupt event occurring before the sequence is complete will cause the interrupt routine to be executed with invalid data, or it will corrupt an intermediate value in the main program.

The situations described above apply whether you write your programs in STL or LAD. If you write your programs in LAD, you should also be aware that many LAD instructions produce a sequence of STL instructions. If the LAD instruction is located in the main program and is operating on data stored in a shared memory location, an interrupt event can occur between the execution of the STL instructions, altering intermediate values and making it appear that the LAD instruction executed incorrectly. For techniques to avoid problems with data sharing, see Programming Techniques for Data Sharing .

Programming Techniques for Data Sharing

The following programming techniques should be followed to avoid problems with data sharing between your main program and interrupt routines. These techniques either restrict the way access is made to shared memory locations, or they make instruction sequences using shared memory locations uninterruptible. The appropriate technique depends upon the size of the data being shared (simple elements such as a byte, word, or double-word variable or complex elements such as multiple variables) and the programming language (STL or LAD).

If the shared data is a single byte, word, or double-word variable and your program is written in STL, then make sure that intermediate or temporary values are not stored in shared memory locations. A shared location should be accessed in the main program only as the initial source value or the final destination value in a sequence of operations.

If the shared data is a single byte, word, or double-word variable and your program is written in LAD, then access shared memory locations using a Move instruction. If the main program performs one or more operations on a data value provided by an interrupt routine, the Move instruction must be used to move the data value from the shared memory location to a non-shared memory location or to an accumulator. If the main program performs one or more operations on data in order to provide a value to an interrupt routine, then the last operation must be a Move instruction that moves the data value from an accumulator or non-shared memory location to the shared memory location. Other instructions in the sequence must not directly access the shared memory location.

If the shared data is composed of related bytes, words, or double-words whose values must agree; for example, the pressure and temperature of a gas in a tank, then the interrupt disable/enable instructions, DISI and ENI, must be used to control interrupt routine execution. At the point in your main program (STL or LAD) where operations on shared memory locations are to begin, interrupts must be disabled. Once all actions affecting shared locations are complete, interrupts must be re-enabled. During the time that interrupts are disabled, interrupt routines cannot execute and access shared memory locations.

Interrupt Event Priority Table

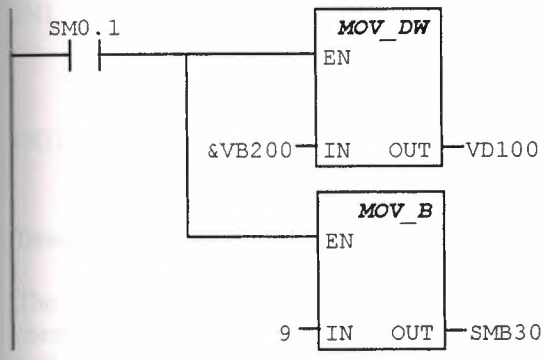
Interrupt Description (By group priority)	Event #	**In-Group Priority**	**Supported in CPU 21**
Comm. (Highest Priority)			
Receive interrupt	8	0	Y
Transmit complete interrupt	9	0*	Y
Discrete (Middle Priority)			
Rising edge, I0.0**	0	0	Y
Rising edge, I0.1	2	1	
Rising edge, I0.2	4	2	
Rising edge, I0.3	6	3	
Falling edge, I0.0**	1	4	Y
Falling edge, I0.1	3	5	
Falling edge, I0.2	5	6	
Falling edge, I0.3	7	7	
HSC0 CV=PV** (*current value = preset value*)	12	0	Y
HSC1 CV=PV (*current value = preset value*)	13	8	
HSC1 direction input changed	14	9	
HSC1 external reset	15	10	
HSC2 CV=PV (*current value = preset value*)	16	11	
HSC2 direction input changed	17	12	
HSC2 external reset	18	13	
PLS0 pulse count complete interrupt	19	14	
PLS1 pulse count complete interrupt	20	15	
Timed (Lowest Priority)			
Timed interrupt 0	10	0	Y
Timed interrupt 1	11	1	

* Since communication is inherently half-duplex, both transmit and receive are the same priority.
**If event 12 (HSC0 CV=PV) is attached to an interrupt, then neither event 0 nor event 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.

Ladder Interrupt / Communication Instruction Examples

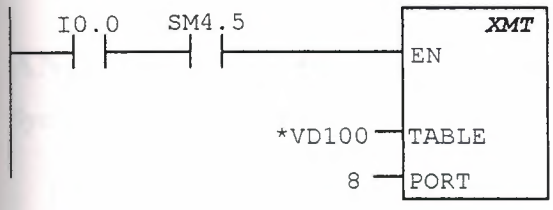
Network 1

On the first scan, create a pointer to the data to be transmitted. Select freeport mode, 9600 baud, no parity, 8 bits per character. SMB30 is the freeport control byte.



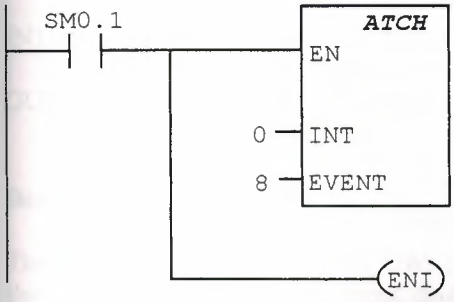
Network 2

When I0.0 and SM4.5 are both on, the message in the buffer (pointed to by VD100) is transmitted. SM4.5 is on when the transmitter is idle.



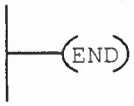
Network 3

Assign receive interrupt event 8 to interrupt routine 0, and enable the routine.



Network 4

End of main ladder program.



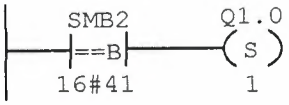
Network 5

Begin interrupt routine 0 .

Network 6

6

Compare received character in special memory byte SMB2 with capital letter "A" If character is "A", Q0.1 is set.



Network 7

Return from interrupt to main program.

Horizontal Lines

In ladder logic, horizontal lines represent wires connecting elements in series.

All lines in a network must be connected to valid elements.
All networks must terminate in a coil or a box.

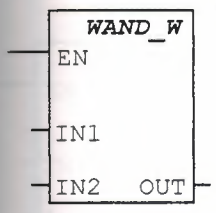
Vertical Lines

In ladder logic, vertical lines represent wires connecting to parallel branches.

All lines in a network must be connected to valid elements.
All networks must terminate in a coil or a box.

AND Word

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

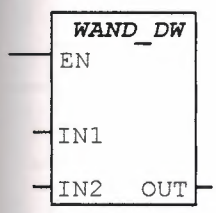
The AND Word (WAND_W) box ANDs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

AND Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

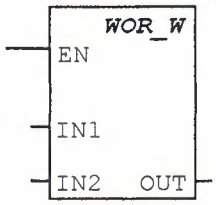
The AND Double Word (WAND_DW) box ANDs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

OR Word

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

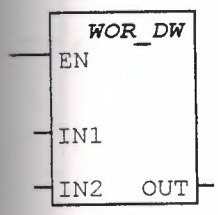
The OR Word (WOR_W) box ORs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

OR Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

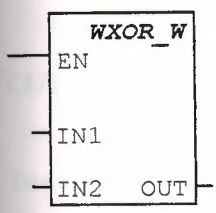
The OR Double Word (WOR_DW) box ORs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

XOR Word

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

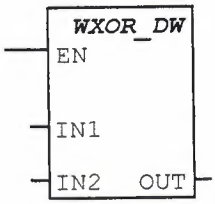
The Exclusive OR Word (WXOR_W) box XORs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

XOR Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

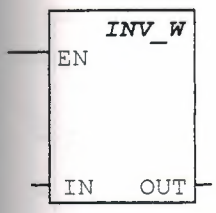
The Exclusive OR Double Word (WXOR_DW) box XORs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Invert Word

Symbol:



Operands:

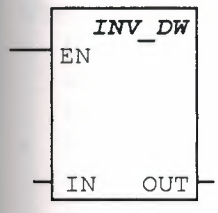
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Invert Word (INV_W) box takes the ones complement of the input word value (IN) and loads the result in a word value (OUT).

Invert Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

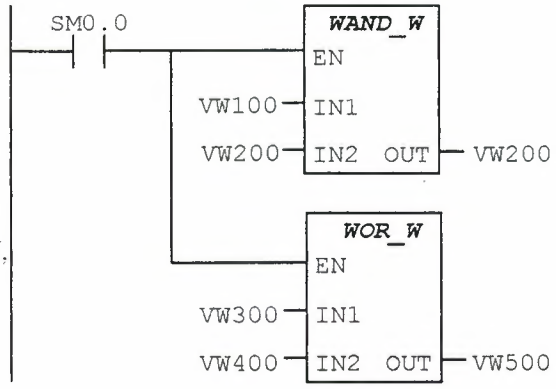
Description of operation:

The Invert Double Word (INV_DW) box takes the ones complement of the input double word value (IN) and loads the result in a double word value (OUT).

Ladder Logical Operations Examples

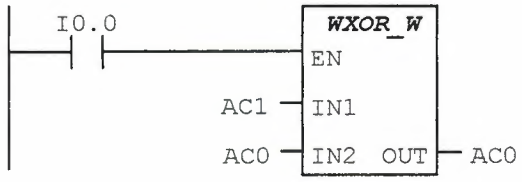
Network 1

Every scan, AND VW100 and VW200 together and store the result in VW200. Also, OR VW300 and VW400 together and store the result in VW500.



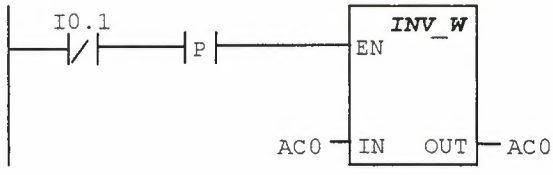
Network 2

When I0.0 is on, "XOR" AC1 and AC0 together and store the result in AC0 .



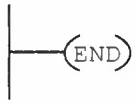
Network 3

When I0.1 transitions from off to on, invert AC0 (ones complement) and store it in AC0 .



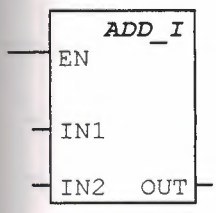
Network 4

End of main user program.



Add Integer

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Add Integer (ADD_I) box adds two 16-bit integers (IN1, IN2), and produces a 16-bit result (OUT), as is shown in the equation:

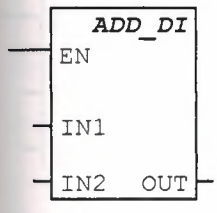
IN1 + IN2 = OUT

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Add Double Integer

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Add Double Integer (ADD_DI) box adds two 32-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:

IN1 + IN2 = OUT

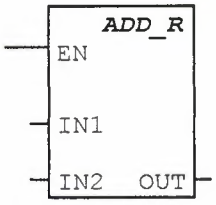
Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Add Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:

The Add Real (ADD_R) box adds two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:

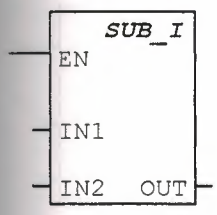
IN1 + IN2 = OUT

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Subtract Integer

Symbol:



Operands:

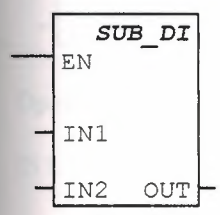
IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Subtract Integer (SUB_I) box subtracts two 16-bit integers (IN1, IN2), and produces a 16-bit result (OUT), as is shown in the equation:
IN1 – IN2 = OUT
Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Subtract Double Integer

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Subtract Double Integer (SUB_DI) box subtracts two 32-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:

IN1 – IN2 = OUT

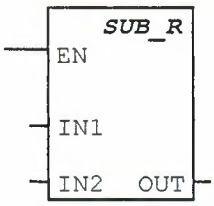
Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Subtract Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Sutract Real (SUB_R) box subtracts two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:

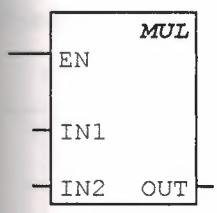
IN1 – IN2 = OUT

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Multiply Integer

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Integer (MUL) box multiplies two 16-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:

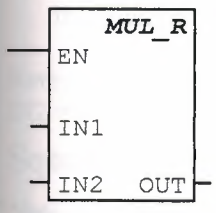
IN1 * IN2 = OUT

Note:
Some overlapping input and output operands are invalid.

Multiply Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Real (MUL_R) box multiplies two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:

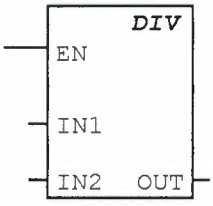
IN1 * IN2 = OUT

Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands. and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Divide Integer

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Divide Integer (DIV) box divides two 16-bit integers (IN1, IN2), and produces a 32-bit result (OUT) composed of of a 16-bit quotient and a 16-bit remainder, as is shown in the equation:

IN1 / IN2 = OUT

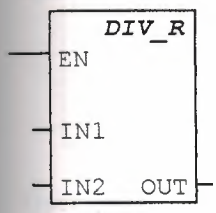
Notes:

- Some overlapping input and output operands are invalid.
- The 32-bit result (OUT) cannot be the same as the second input (IN2).

Divide Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:

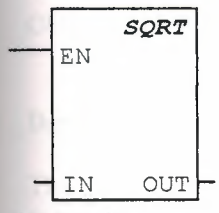
The Divide Real (DIV_R) box divides two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number quotient (OUT), as is shown in the equation:
IN1 / IN2 = OUT
Note:
When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Note:
IN2 = OUT is not valid for Ladder programming.

Square Root Real

Note: CPU 214 only.
Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

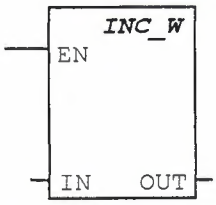
Description of operation:

The Square Root of Real Numbers (SQRT) box takes the square root of a 32-bit real number (IN) and produces a 32-bit real number result (OUT), as is shown in the equation:

$\sqrt{IN} = OUT$

Increment Word

Symbol:



Operands:

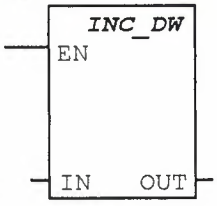
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Increment Word (INC_W) box adds 1 to the input word value (IN) and loads the result in a word value (OUT), as is shown in the equation:
IN + 1 = OUT

Increment Double Word

Symbol:



Operands:

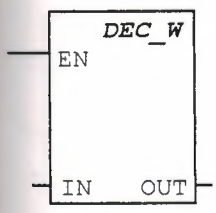
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Increment Double Word (INC_DW) box adds 1 to the input double word value (IN) and loads the result in a double word value (OUT), as is shown in the equation:

IN + 1 = OUT

Decrement Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

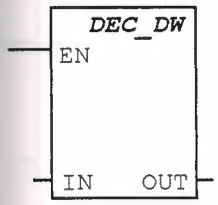
Description of operation:

The Decrement Word (DEC_W) box subtracts 1 from the input word value (IN) and loads the result in a word value (OUT), as is shown in the equation:

IN – 1 = OUT

Decrement Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD,AC , HC, Constant, *VD,*AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

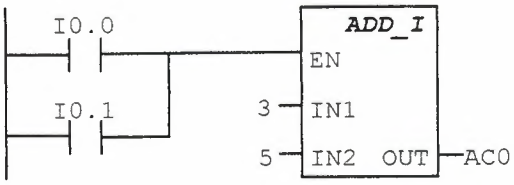
Description of operation:

The Decrement Double Word (DEC_DW) box subtracts 1 from the input double word value (IN) and loads the result in a double word value (OUT), as is shown in the equation:

IN – 1 = OUT

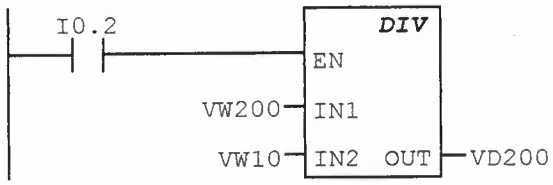
Math/Inc/Dec Examples

Network 1 — When I0.0 or I0.1 is on then AC0 equals the sum of IN1 and IN2.

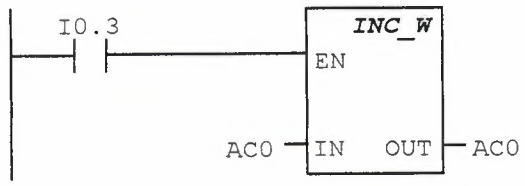


Network 2 — If AC0 equals 8, turn on Q0.0 .

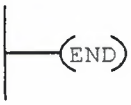
Network 3 — VW200 is divided by VW10. The quotient is put in VW202, and the remainder is put in VW200. (*Note: VD200 contains VW200 and VW202.*)



Network 4 — When I0.3 is on, then the value in AC0 is incremented by 1 and stored in AC0.

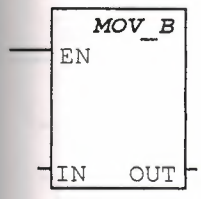


Network 5 — End of the main user program.



Move Byte

Symbol:



Operands:

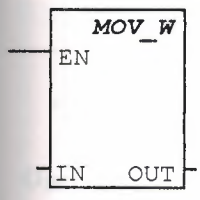
IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:

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

Move Word

Symbol:



Operands:

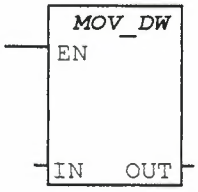
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:

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

Move Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC, &VB, &IB, &QB, &MB, &T, &C
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

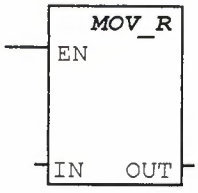
Description of operation:

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

Move Real

Note: CPU 214 only.

Symbol:



Operands:

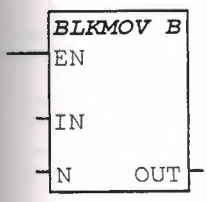
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Move Real (MOV_R) box 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.

Block Move Byte

Symbol:



Operands:

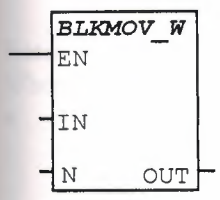
IN (byte):	VB, IB, QB, MB, SMB *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Block Move Byte (BLKMOV_B) box 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.

Block Move Word

Symbol:



Operands:

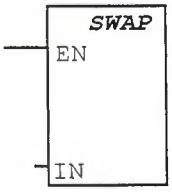
IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Block Move Word (BLKMOV_B) box 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.

Swap

Symbol:



Operands:

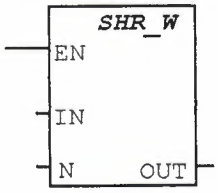
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Swap Byte box exchanges the most-significant byte with the least-significant byte of the word (IN).

Shift Right Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Shift Right Word (SHR_W) box shifts the word value (IN) right by the shift count (N), and loads the result in the output word (OUT).

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit shifted out = 0

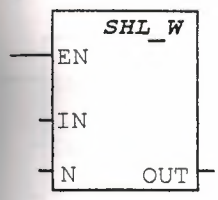
Note:

When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Left Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Shift Left Word (SHL_W) box shifts the word value (IN) left by the shift count (N), and loads the result in the output word (OUT).

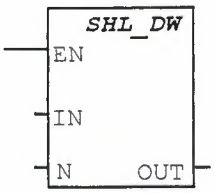
SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit shifted out
= 0	

Note:
When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Left Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Shift Left Double Word (SHL_DW) box shifts the double word value (IN) left by the shift count (N), and loads the result in the output double word (OUT).

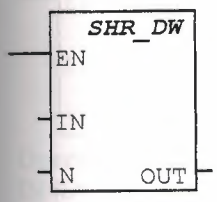
SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit shifted out
= 0	

Note:
When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Right Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Shift Right Double Word (SHR_DW) box shifts the double word value (IN) right by the shift count (N), and loads the result in the output double word (OUT).

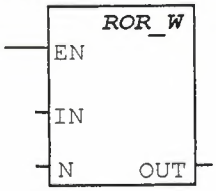
SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit shifted out = 0

Note:
When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Right Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Rotate Right Word (ROR_W) box rotates the word value (IN) right by the shift count (N), and loads the result in the output word (OUT).

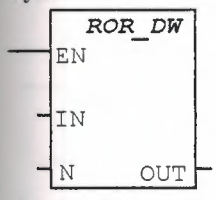
SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

Note:
When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Right Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Rotate Right Double Word (ROR_DW) box rotates the double word value (IN) right by the shift count (N), and loads the result in the output double word (OUT).

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

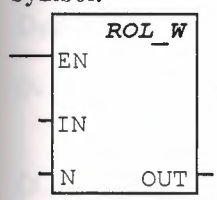
Note:

When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Left Word

Symbol:



Operands:

IN (word):	VW,T,C,IW,QW,MW,SMW, AC,AIW, Constant, *VD,*AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Rotate Left Word (ROL_W) box rotates the word value (IN) left by the shift count (N), and loads the result in the output word (OUT).

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

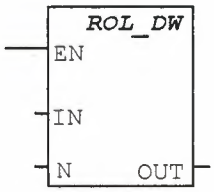
Note:

When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Left Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Rotate Left Double Word (ROL_DW) box rotates the double word value (IN) left by the shift count (N), and loads the result in the output double word (OUT).

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

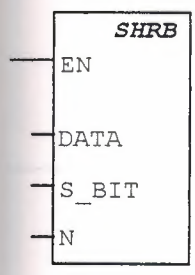
Note:

When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Register Bit

Symbol:



Operands:

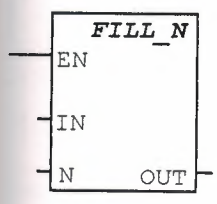
DATA, S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

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

Fill Memory

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

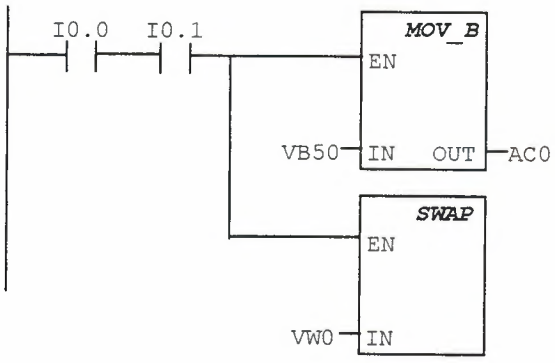
Description of operation:

The Fill Memory Box (FILL_N) 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.

Move / Shift / Rotate / Fill Examples

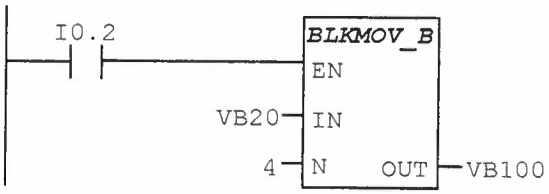
Network 1

When I0.0 and I0.1 are on then move VB50 to AC0, and swap the most significant byte (MSB) of VW0 with the LSB of VW0.



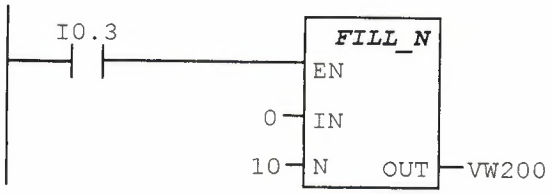
Network 2

When I0.2 is on then move VB20-VB23 toVB100-VB103.



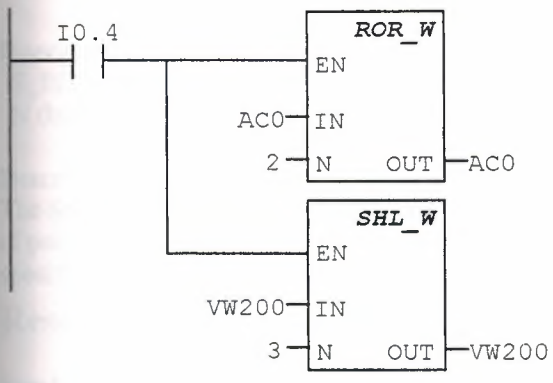
Network 3

When I0.3 is on then fill VW200-VW218 with 0's.



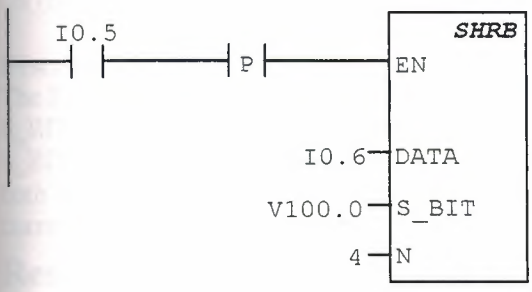
Network 4

When I0.4 is on, then the word value in AC0 is rotated right twice and stored in AC0, and the word value in VW200 is shifted left 3 times and stored in VW200.



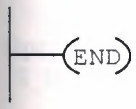
Network 5

Upon every 0 to 1 transition of I0.5, the value of I0.6 is shifted into the shift register starting at V100.0 and of length 4.



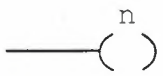
Network 6

Main end of the user program.



Output

Symbol:



Operands:

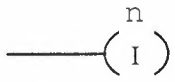
n (bit):	I, Q, M, SM, T, C, V

Description of operation:
An Output coil is turned on and the Bit stored at address ***n*** is set to 1 when power flows to the coil.

A negated output can be created by placing a NOT (Invert Power Flow) contact before an output coil.

Output Immediate Coil

Symbol:



Operands:

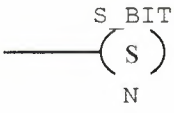
n (bit):	Q

Description of operation:

An Output Immediate Coil is turned on and the Bit at output address ***n*** is set to 1 when power flows to the coil. An update of the addressed image register output Bit and also the corresponding physical output Bit occurs immediately after the coil is scanned without waiting for scan cycle completion.

Set

Symbol:



Operands:

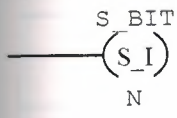
S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Set Coil sets the range of points starting at S_BIT for the number of points specified by N .

Set Immediate Coil

Symbol:



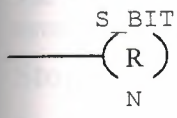
Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Set Immediate Coil immediately sets the range of points starting at S_BIT for the number of points specified by N .

Reset Coil

Symbol:



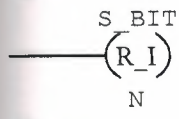
Operands:

S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Reset Coil resets the range of points starting at S_BIT for the number of points specified by N. If S_BIT is specified to be either a T or a C bit, then both the timer/counter bit and the timer/counter current value are reset.

Reset Immediate Coil

Symbol:



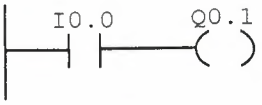
Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

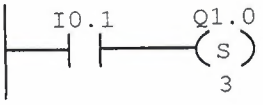
Description of operation:
The Reset Immediate Coil immediately resets the range of points starting at S_BIT for the number of points specified by N .

Ladder Output Coil Examples

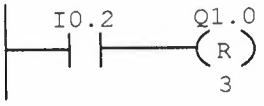
Network 1 — When I0.0 is on, then output Q0.1 is turned on.



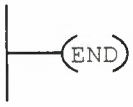
Network 2 — When I0.1 is on, then outputs Q1.0, Q1.1 and Q1.2 are set (turned on).
These outputs will remain on, even if I0.1 is turned off, until they are reset.



Network 3 — When I0.2 is turned on, then outputs Q1.0, Q1.1 and Q1.2 are reset (turned off



Network 4 — End of the main user program.



End

Symbols:

Conditional End

Unconditional End

Operands:

(none)

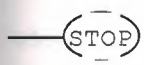
Description of operation:

The Conditional End coil terminates the main user program based on the condition of the preceding logic.

The Unconditional End coil must be used to terminate the user program.

Stop

Symbol:



Operands:

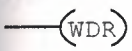
(none)

Description of operation:

The Stop coil terminates execution of the user program by causing a transition to the stop mode.

Watchdog Reset

Symbol:



Operands:

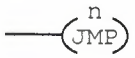
(none)

Description of operation:

The Watchdog Reset (WDR) coil allows the watchdog timer to be retriggered. This extends the time the scan takes without getting a watchdog error.

Jump

Symbol:



Operands:

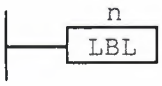
n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:

The Jump to Label (JMP) coil performs a branch to the specified label (*n*) within the program.

Label

Symbol:



Operands:

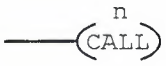
n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:

The Label (LBL) instruction marks the location of the jump destination (*n*). The CPU 212 allows 64 labels, and the CPU 214 allows 256.

Call

Symbol:



Operands:

n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:

The Subroutine Call (CALL) coil transfers control to the subroutine (n).

Subroutine

Symbol:

Operands:
n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:
The Subroutine (SBR) label marks the beginning of the subroutine (*n*). The CPU 212 supports 16 subroutines, and the CPU 214 supports 64.

Return

Symbols:

Conditional Return from Subroutine

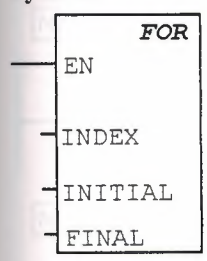
Unconditional Return from Subroutine

Operands:
(none)

Description of operation:
The Conditional Return from Subroutine coil may be used to terminate a subroutine, based on the condition of the preceding logic.
The Unconditional Return from Subroutine coil must be used to terminate each subroutine.

For

Symbol:



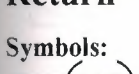
Operands:

INDEX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
INITIAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
FINAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The FOR box executes the code between the FOR and the NEXT. You must specify the current loop count (INDEX), the starting value (INITIAL), and the ending value (FINAL). 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.
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 incrementing 1,2,3, . . 10.

Next

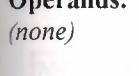
Symbol:



Operands:
(none)
Description of operation:
The NEXT coil marks the end of the FOR loop, and sets the top of stack to 1.

No Operation

Symbol:



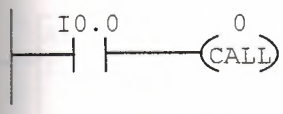
Operands:
n: 0-255

Description of operation:
The No Operation (NOP) coil has no effect on the user program execution. The operand *n* is a number from 0-255.

Ladder Program Control Examples

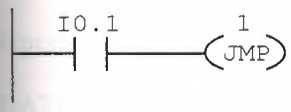
Network 1

When I0.0 is on, execute Subroutine 0.



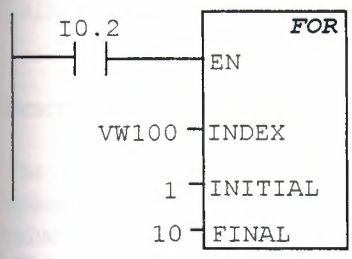
Network 2

When I0.1 is on, jump to Label 1.



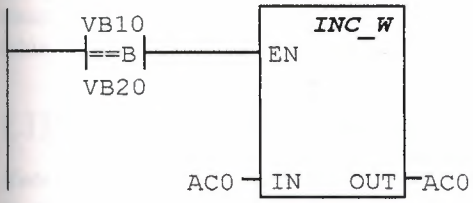
Network 3

When I0.2 is on, execute the For/Next loop 10 times.



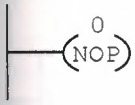
Network 4

If VB10 = VB20, then increment AC0 by 1.



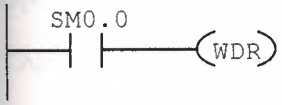
Network 5

This network does nothing.



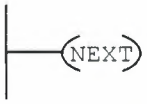
Network 6

SM0.0 is always on, therefore the Watchdog Timer is extended to allow a longer scan.



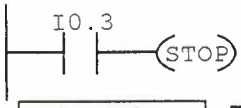
Network 7

This is the end of the For/Next loop.



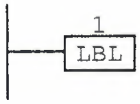
Network 8

If I0.3 comes on, then the CPU goes to Stop mode.



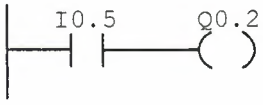
Network 9

The Jump in Network #2 jumps to this location.



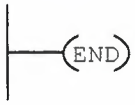
Network 10

When I0.5 is on, turn on Q0.2.



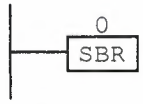
Network 11

End of the main user program.



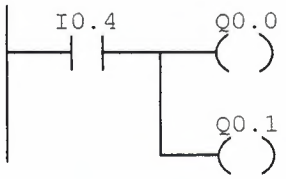
Network 12

Start of Subroutine 0.



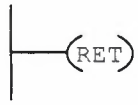
Network 13

If I0.4 is on, then turn on Q0.0 and Q0.1.



Network 14

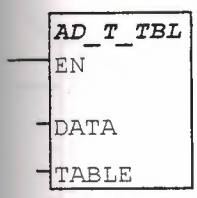
End of Subroutine 0.



Add to Table

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC

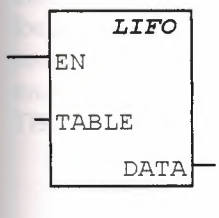
Description of operation:

The Add To Table (AD_T_TBL) box adds word values (DATA) to the table (TABLE). The first value of the table is the maximum table length (TL). The second value is the entry count (EC) that specifies the number of entries in the table. New data are added to the table after the last entry. Each time new data are added to the table, the entry count (EC) is incremented. If you try to overfill the table, the Table Full memory bit (SM1.4) is set.

LIFO (Last In First Out)

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

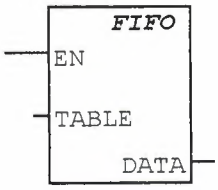
Description of operation:

The Last In First Out (LIFO) box removes the last entry in the table (TABLE), and outputs the value to the location (DATA). The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

FIFO (First In First Out)

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

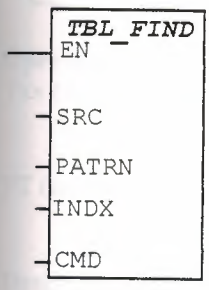
Description of operation:

The First In First Out (FIFO) box removes the first entry in the table (TABLE), and outputs the value to the location (DATA). All other entries of the table are shifted up one location. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

Find Table

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AIW, AC, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
CMD:	1-4

Description of operation:

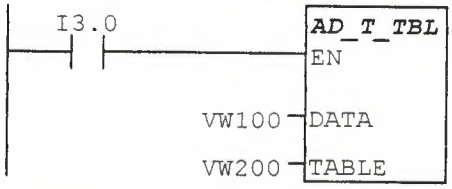
The Find Table (TBL_FIND) box searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the criteria (CMD). The CMD parameter is given a numeric value 1-4 that corresponds to =, <>, <, and >, respectively.

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

Ladder Table / Find Instruction Examples

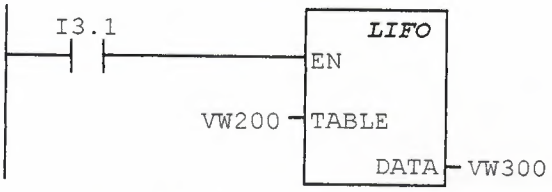
Network 1

When I3.0 is on, the value VW100 is added to the table starting at VW200. The EC (entry count) is incremented by one.



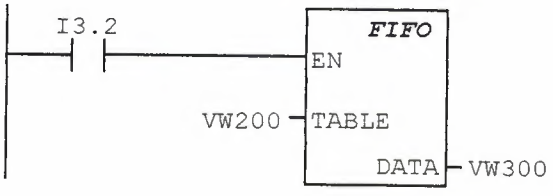
Network 2

When I3.1 is on, the last data value of the table starting at VW200 is output to the data location VW300. The EC is decremented by one.



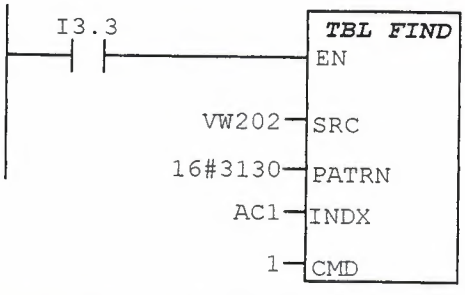
Network 3

When I3.2 is on, the first data value of the table starting at VW200 is output to the data location VW300. The EC is decremented by one.



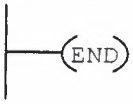
Network 4

When I3.3 is on, the table VW202 is searched for a value equal to 3130 Hex.



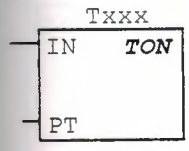
Network 5

End of the main user program.



Timer – On Delay

Symbol:



Operands:

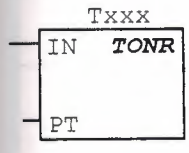
Txx (word):	CPU 212: 32-63 CPU 214: 32-63, 96-127
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The On–Delay Timer (TON) box times up to the maximum value when the enabling Input (IN) comes on. When the current value (Txxx) is >= the Preset Time (PT), the timer bit turns on. It resets when the enabling input goes off. Timing stops upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T32	T96
10 ms	T33-T36	T97-T100
100 ms	T37-T63	T101-T127

Timer – Retentive On Delay

Symbol:



Operands:

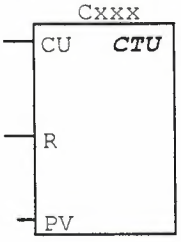
Txxx (word):	CPU 212: 0-31 CPU 214: 0-31, 64-95
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Retentive On Delay Timer (TONR) box times up to the maximum value when the enabling Input (IN) comes on. When the current value (Txxx) is >= the Preset Time (PT), the timer bit turns on. Timing stops when the enabling input goes off, or upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T0	T64
10 ms	T1-T4	T65-T68
100 ms	T5-T31	T69-T95

Count Up

Symbol:



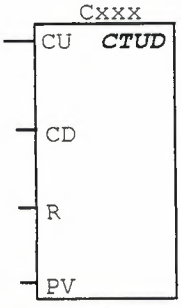
Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Count Up (CTU) box counts up to the maximum value on the rising edges of the Count Up (CU) input. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. It resets when the Reset (R) input turns on. It stops counting upon reaching the maximum value (32,767).

Count Up / Down

Symbol:



Operands:

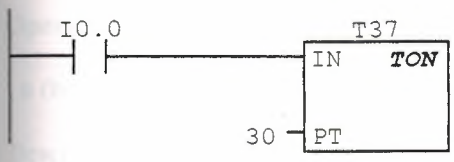
Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Count Up/Down (CTUD) box 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 >= to the Preset Value (PV), the counter bit (Cxxx) turns on. It stops counting up upon reaching the maximum value (32,767), and stops counting down upon reaching the minimum value (-32,768). It resets when the Reset (R) input turns on.

Ladder Timer / Counter Examples

Network 1

When I0.0 is on then the timer will start. After 3 seconds (30 X 100ms) T37 bit will come on.

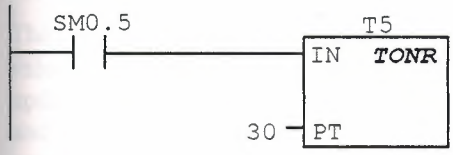


Network 2

When Timer 37 reaches its preset, turn on Q0.0 .

Network 3

When SM0.5 (1 sec. clock pulse, .5 sec. on and .5 sec. off) is ON, then the timer will time. The T5 bit will come on after 6 seconds.

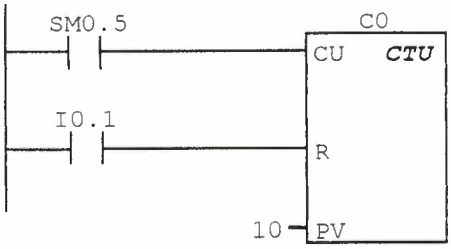


Network 4

When Timer 5 reaches its preset, turn on Q0.1 .

Network 5

By using SM0.5 (1 sec. clock pulse) the counter will count pulses and turn on the C0 bit when a count of 10 is reached. I0.0 resets the counter.

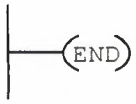


Network 6

When C0 reaches its preset, turn on Q0.2 .

Network 7

End of the main user program.



Statement List Instruction Set

Out (STL)

Format:

```
=       n
```

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Out (=) instruction copies the bit value on the top of the logic stack to address ***n***.

Example:

```
LD    I0.0
=     Q2.0
```

Out Immediate (STL)

Format:

```
=I      n
```

Operands:

n (bit): Q

Description of operation:

The Out Immediate (=I) instruction copies the bit value on the top of the logic stack to address ***n***. An update of the addressed image register output bit and also the corresponding physical output bit occurs immediately after =I execution without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
```

And (STL)

Format:

```
A       n
```

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The And (A) instruction performs a logical And of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of logic stack value.

Example:

```
LD    I0.1
A     I0.2
=     Q1.0
```

And Immediate (STL)

Format:

```
AI      n
```

Operands:

n (bit): I

Description of operation:

The And Immediate (AI) instruction performs a logical And of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after AI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI   I0.1
AI    I0.2
=I    Q1.0
```

And Load (STL)

Format:

```
ALD
```

Operands:

(none)

Description of operation:

The And Load (ALD) instruction performs a logical And on the bit values in the first (top) and second levels of the logic stack. The result is loaded to the top of stack and stack depth is reduced by one.

Example:

```
LD    I0.0
LD    I0.1
LD    I2.0
A     I2.1
OLD
ALD
```

```
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
```

And Not (STL)

Format:

AN *n*

Operands:

n (bit):	I, Q, M, SM, T, C, V

Description of operation:
The And Not (AN) instruction performs a logical And Not of the bit value at address **n** with the top of stack value. The result becomes the new top of stack value.

Example:

```
LD      I0.1
AN      I0.2
=       Q1.0
```

And Not Immediate (STL)

Format:

ANI *n*

Operands:

n (bit):	I

Description of operation:

The And Not Immediate (ANI) instruction performs a logical And Not of the bit value at address ***n*** with the top of stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after ANI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI     I0.1
ANI     I0.2
=I      Q1.0
```

Edge Down (STL)

Format:

ED

Operands:

(none)

Description of operation:
The Edge Down (ED) instruction detects a scan-to-scan transition from 1 to 0 in top of stack bit value. Upon detection of such a transition, the top of stack value is set to 1; otherwise it is set to 0.

Example:

```
LD      I0.2
ED
=       Q2.2
```

Edge Up (STL)

Format:

EU

Operands:

(none)

Description of operation:

The Edge Up (EU) instruction detects a scan-to-scan transition from 0 to 1 in top of stack bit value. Upon detection of such a transition, the top of stack value is set to 1; otherwise it is set to 0.

Example:

```
LD      I0.1
EU
=       Q2.1
```

Load (STL)

Format:

LD *n*

Operands:

n (bit):	I, Q, M, SM, T, C, V

Description of operation:

The Load (LD) instruction copies the bit value at address ***n*** to the top of the logic stack. Other stack bit values move down one level.

Example:

```
LD      I0.1
A       I0.2
=       Q1.0
```

Load Immediate (STL)

Format:

LDI *n*

Operands:

n (bit): I

Description of operation:

The Load Immediate (LDI) instruction copies the bit value at address ***n*** to the top of the logic stack immediately after execution without waiting for scan cycle completion. The image register is not updated. Other stack bit values move down one level.

Example:

```
LDI   I0.1
AI    I0.2
=I    Q1.0
```

Load Not (STL)

Format:

LDN *n*

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Load Not (LDN) instruction copies the logical Not of the bit value at image register address ***n*** to the top of the logic stack. Other stack bit values move down one level.

Example:

```
LDN   I0.1
AN    I0.2
=     Q1.0
```

Load Not Immediate (STL)

Format:

LDNI *n*

Operands:

n (bit): I

Description of operation:

The Load Not Immediate (LDNI) instruction copies the logical Not of the bit value at address ***n*** to the top of the logic stack immediately after execution without waiting for scan cycle completion. Other stack bit values move down one level.

Example:

```
LDNI  I0.1
ANI   I0.2
=I    Q1.0
```

Logic Pop (STL)

Format:

LPP

Operands:

(none)

Description of operation:
The Logic Pop (LPP) instruction pops one value off of the stack. The second level bit value becomes the new top of stack value. Other stack bit values move up one level.

Example:

```
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
```

Logic Push (STL)

Format:

LPS

Operands:

(none)

Description of operation:

The Logic Push (LPS) instruction duplicates the top of stack bit value and pushes this value onto the stack. The bottom of the stack is pushed off and lost.

Example:

```
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
```

Logic Read (STL)

Format:
LRD

Operands:
(none)

Description of operation:
The Logic Read (LRD) 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.

Example:

```
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
```

Logical Negation (STL)

Format:

NOT

Operands:

(none)

Description of operation:
The Logical Negation (NOT) instruction changes the top of stack bit value from 0 to 1, or from 1 to 0.

Example:

```
LD    I0.0
NOT
=     Q2.0
```

Or (STL)

Format:

O *n*

Operands:
n (bit): I, Q, M, SM, T, C, V

Description of operation:
The Or (O) instruction performs a logical Or of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of stack value.

Example:

```
LD    I1.1
O     I1.2
=     Q1.1
```

Or Immediate (STL)

Format:
OI *n*

Operands:
n (bit): I

Description of operation:
The Or Immediate (OI) instruction performs a logical Or of the bit value at input module address ***n*** with the top of stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after OI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI   I1.1
OI    I1.2
=I    Q1.1
```

Or Load (STL)

Format:
OLD

Operands:

(none)

Description of operation:
The Or Load (OLD) instruction performs a logical Or with the bit values in the first (top) and second levels of the stack. The result is loaded to the top of stack. After execution of OLD, stack depth is reduced by one.

Example:

```
LD    I0.0
LD    I0.1
LD    I2.0
A     I2.1
OLD
ALD
```

Or Not (STL)

Format:

ON *n*

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Or Not (ON) instruction performs a logical Or Not of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of stack value.

Example:

```
LD    I1.1
ON    I1.2
=     Q1.1
```

Or Not Immediate (STL)

Format:

ONI *n*

Operands:

n (bit): I

Description of operation:

The Or Not Immediate (ONI) instruction immediately performs a logical Or Not of the bit value at physical input address ***n*** with the top of logic stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after ONI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI   I1.1
ONI   I1.2
=I    Q1.1
```

Reset (STL)

Format:

R *S_BIT, N*

Operands:

S_BIT (bit): I, Q, M, SM, T, C, V

N (byte): IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Reset (R) instruction resets a range of bit values. Bit values of 0 are written to a range starting at address S_BIT for the number of bits specified by N. If S_BIT is specified to be either a T or a C bit, then both the timer/counter bit and the timer/counter current value are reset to 0.

Example:

```
LD    I0.0
=     Q2.0
S     Q2.1, 1
R     Q2.2, 1
R     Q1.0, 3
```

Reset Immediate (STL)

Format:

RI *S_BIT, N*

Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Reset Immediate (RI) instruction immediately resets a range of bit values. Bit values of 0 are written to a range starting at S_BIT for the number of bits specified by N. Specified bits in the image register and corresponding physical outputs are updated at execution time without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
SI    Q2.1, 1
RI    Q2.2, 1
RI    Q1.0, 3
```

Set (STL)

Format:

S *S_BIT, N*

Operands:

S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Set (S) instruction sets a range of bit values. Bit values of 1 are written to a range starting at address S_BIT for the number of bits specified by N.

Example:

```
LD    I0.0
=     Q2.0
S     Q2.1, 1
R     Q2.2, 1
```

Set Immediate (STL)

Format:

SI *S_BIT, N*

Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Set Immediate (SI) instruction immediately resets a range of bit values. A bit value of 1 is written to a range starting at S_BIT for the number of bits specified by N. Specified bits in the image register and physical output modules are updated at execution time without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
SI    Q2.1, 1
RI    Q2.2, 1
```

Read Time of Day (STL)

Note: Real Time Clock instructions are supported by the CPU 214 only.

Format:

TODR *T*

Operands:

T (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of STL operation:

Read Time of Day (TODR) reads the current date and time from the Real Time Clock. The 8 bytes of time data are written to memory with the area and starting address specified by ***T***.

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 - 1 to 7	1 = Sunday
		d - 0	Day of week remains 0

Example:

```
      LD    I2.1          //Enable
READ_RTC
      TODR  VB400         //Read
clock
      MOVB  VB400, AC0    //Move
year value
                          //to
accumulator
```

Example Memory Data Starting at VB400:

Note:
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

Note:
Do not use the TODR/TODW instructions in both the main program and in an interrupt routine. If you do this and the TOD instruction is executing when the the interrupt that also executes the TOD instruction occurs, then the TOD instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted.

Write Time of Day (STL)

Note: Real Time Clock instructions are supported by the CPU 214 only.

Format:

```
TODW    T
```

Operands:

T (byte): VB, IB, QB, MB, SMB, *VD, *AC

Description of STL operation:

Write Time of Day (TODW) sets a date and time into the Real Time Clock. The 8 bytes of time data are read from a memory area with the starting address specified by ***T***.

The Date and Time setting data must be in BCD format (4 bits per digit; decimal digits 0-9 only) and previously stored in the specified memory location before execution of TODW.

Year/Month	yymm	yy - 0 to 99	mm - 1 to 12
Day/Hour	ddhh	dd - 1 to 31	hh - 0 to 23
Min/Sec	mmss	mm - 0 to 59	ss - 0 to 59
Day of week	000d	d - 1 to 7	1 = Sunday
		d - 0	Day of week remains 0

Example:

```
      INCW  AC0
      //Increment year to 96
      MOVB  AC0, VB400  //Store
new year
      TODW  VB400       //Write
new year
                        //to clock
```

Example Memory Data Starting at VB400:

Note:
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

Note:
Do not use the TODR/TODW instructions in both the main program and in an interrupt routine. If you do this and the TOD instruction is executing when the interrupt that also executes the TOD instruction occurs, then the TOD instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted.

Compare Byte Equal Instructions (STL)

Format:

```
LDB= n1, n2
AB=  n1, n2
OB=  n1, n2
```

Operands:

n1, n2 (byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The **Load Byte (LDB)**, **And Byte (AB)**, and **Or Byte (OB) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
      LD    Q0.0
      AB=   VB4, VB8
      =     Q2.0
```

Compare Byte Greater Than or Equal Instructions (STL)

Format:

```
LDB>= n1, n2
AB>=  n1, n2
OB>=  n1, n2
```

Operands:

n1, n2 (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The **Load Byte (LDB), And Byte (AB),** and **Or Byte (OB) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD    Q0.0
AB>=  VB4, VB8
=     Q2.0
```

Compare Byte Less Than or Equal Instructions (STL)

Format:

```
LDB<=  n1, n2
AB<=   n1, n2
OB<=   n1, n2
```

Operands:

n1, n2 (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The **Load Byte (LDB), And Byte (AB),** and **Or Byte (OB) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD    Q0.0
AB<=  VB4, VB8
=     Q2.0
```

Compare Word Equal Instructions (STL)

Format:

```
LDW=  n1, n2
AW=   n1, n2
OW=   n1, n2
```

Operands:

n1, n2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD    Q0.0
AW=   VW4, VW8
=     Q2.0
```

Compare Word Greater Than or Equal Instructions (STL)

Format:

```
LDW>=  n1, n2
AW>=   n1, n2
OW>=   n1, n2
```

Operands:

n1, n2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD    Q0.0
AW>=  VW4, VW8
=     Q2.0
```

Compare Word Less Than or Equal Instructions (STL)

Format:

```
LDW<=   n1, n2
AW<=    n1, n2
OW<=    n1, n2
```

Operands:

n1, n2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD    Q0.0
AW<=  VW4, VW8
=     Q2.0
```

Compare Double Word Equal Instructions (STL)

Format:

```
LDD=    n1, n2
AD=     n1, n2
OD=     n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD    Q0.0
OD=   VD6, VD20
=     Q2.0
```

Compare Double Word Greater Than or Equal Instructions (STL)

Format:

```
LDD>=   n1, n2
AD>=    n1, n2
OD>=    n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD    Q0.0
OD>=  VD6, VD20
=     Q2.0
```

Compare Double Word Less Than or Equal Instructions (STL)

Format:

```
LDD<=   n1, n2
AD<=    n1, n2
OD<=    n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD    Q0.0
OD<=  VD6, VD20
=     Q2.0
```

Compare Real Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR=    n1, n2
AR=     n1, n2
OR=     n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD    Q0.0
OR=   VD6, VD20
=     Q2.0
```

Compare Real Greater Than or Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR>=   n1, n2
AR>=    n1, n2
OR>=    n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD    Q0.0
OR>=  VD6, VD20
=     Q2.0
```

Compare Real Less Than or Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR<=   n1, n2
AR<=    n1, n2
OR<=    n1, n2
```

Operands:

n1, n2 (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD    Q0.0
OR<=  VD6, VD20
=     Q2.0
```

ASCII to Hex (STL)

Format:

```
ATH    IN, OUT, LEN
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The ASCII to HEX (ATH) instruction converts the ASCII string of length LEN, starting with the character IN, to hexadecimal digits starting at the location OUT. The maximum length of the ASCII string is 255 characters.
Legal ASCII characters are the hexadecimal values 30-39, and 41-46. If an illegal ASCII character is encountered, the conversion is terminated, and the NOT_ASCII memory bit (SM1.7) is set.
Example:

```
LD    I3.2
ATH   VB30, VB40, 3
```

Convert BCD to Integer (STL)

Format:
BCDI *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert BCD to Integer (BCDI) instruction converts the BCD value (IN) to an integer value. The result replaces the original input value. If the input value contains an invalid BCD digit, the BCD/BIN memory bit (SM1.6) is set.

Example:

```
LD    I3.0
BCDI  AC0
```

Decode (STL)

Format:

DECO *IN, OUT*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

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

Example:

```
LD    I3.1
DECO  AC2, VW40
```

Encode (STL)

Format:

ENCO *IN, OUT*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:

The Encode (ENCO) instruction writes the bit number of the least-significant bit set in the input word (IN) into the least-significant nibble (LSN) of the output byte (OUT).

Example:

```
LD    I3.1
ENCO  AC2, VB40
```

Integer Double Word to Real(STL)

Note: Real Conversion instructions are supported by the CPU 214 only.

Format:

DTR *IN, OUT*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, SD, AC, *VD, *AC

Description of operation:
The Integer Double Word to Real (DTR) instruction converts a 32-bit signed integer (IN) into a 32bit real number (OUT).

Example:

```
LD    I3.1
DTR   AC1, VD40
```

Segment (STL)

Format:

SEG *IN, OUT*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Segment (SEG) 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).

Example:

```
LD    I3.1
SEG   VB48, AC1
```

Hex to ASCII (STL)

Format:

```
HTA   IN, OUT, LEN
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The HEX to ASCII (HTA) instruction converts the hexadecimal digits, starting with the input byte IN, to an ASCII string starting at the 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.

Example:

```
      LD    I3.2
      HTA   VB30, VB40, 3
```

Convert Integer to BCD (STL)

Note: CPU 214 only.

Format:

```
IBCD  IN
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Convert Integer to BCD (IBCD) instruction converts the integer value (IN) to a BCD value (OUT). The result replaces the original input value. If the conversion produces a BCD number greater than 9999, the BCD/BIN memory bit (SM1.6) is set.

Example:

```
      LD    I3.0
      IBCD  AC0
```

Truncate (STL)

Note: CPU 214 only.

Format:

```
TRUNC  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, SD, AC, *VD, *AC

Description of operation:

The Truncate (TRUNC) 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.

Example:

```
      LD     I3.1
      TRUNC  AC1, VD40
```

Count Up (STL)

Format:

```
CTU   Cxxx, PV
```

Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Count Up (CTU) instruction counts up to the maximum value on the rising edges of the Count Up (CU) input (the value loaded in the second stack location). The counter resets when the reset input turns on. The reset input is the top of stack value. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter stops counting upon reaching the maximum value (32,767).

Example:

```
      LD    I4.0  //Count up input
      LD    I2.0  //Reset input
      CTU   48, 4
```

Count Up/Down (STL)

Format:
```
CTUD  Cxxx, PV
```

Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Count Up/Down (CTUD) instruction counts up on rising edges of the count-up input. The count-up input is the value loaded in the third stack location. The counter counts down on the rising edges of the count-down input. The count-down input is the value loaded in the second stack location. The counter resets when the reset input turns on. The reset input is the top of stack value or the first stack location. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter stops counting up upon reaching the maximum value (32,767), and stops counting down upon reaching the minimum value (-32,768).

Example:

```
      LD    I4.0  //Count Up Clock
      LD    I3.0  //Count Down
Clock
      LD    I2.0  //Reset
      CTUD  C48, 4
```

Attach Interrupt (STL)

Format:
```
ATCH  INT, EVENT
```

Operands:

INT (byte):	CPU 212: 0-31 CPU 214: 0-127
EVENT (byte):	CPU 212: 0, 1, 8-10, 12 CPU 214: 0-20

Description of operation:
The Attach Interrupt (ATCH) instruction associates an interrupt event (EVENT) with an interrupt routine number (INT), and enables the interrupt event.

Example:

```
      LD    SM0.1
      ATCH  4, 0
      ENI
```

Detach Interrupt (STL)

Format:

```
DTCH  EVENT
```

Operands:

EVENT (byte):	CPU 212: 0, 1, 8-10, 12 CPU 214: 0-20

Description of operation:

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

Example:

```
      LD    SM5.0
      DTCH  0
```

Interrupt Routine (STL)

Format:

```
INT   n
```

Operands:

n (word):	CPU 212: 0-31 CPU 214: 0-127

Description of operation:
The Interrupt Routine (INT) instruction marks the beginning of the interrupt routine (n). The maximum number of interrupts supported by the CPU 212 is 32, and by the CPU 214, 128.

Example:

```
      INT   4
```

Enable Interrupt (STL)

Format:

```
ENI
```

Operands:

(*None*)

Description of operation:
The Enable Interrupt (ENI) instruction globally enables processing of all attached interrupt events.

Example:

```
      LD    SM0.1
      ATCH  4, 0
      ENI
```

Disable Interrupt (STL)

Format:

```
DISI
```

Operands:

(*None*)

Description of operation:

The Disable Interrupt (DISI) instruction globally disables processing of all interrupt events.

Example:

```
      LD    M5.0
      DISI
```

Conditional Return from Interrupt (STL)

Format:

```
CRETI
```

Operands:

(*None*)

Description of operation:

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

Example:

```
      LD    SM5.0
      CRETI
```

Return from Interrupt (STL)

Format:
RETI

Operands:
(*None*)

Description of operation:
The Return from Interrupt (RETI) instruction is an unconditional return and must be used to terminate each interrupt routine.

Example:

```
      LD    SM5.0
      CRETI

      RETI
```

High-speed Counter Definition (STL)

Format:

```
HDEF  HSC, MODE
```

Operands:

HSC (byte):	CPU 212: 0 CPU 214: 0-2
MODE (byte):	CPU 212: 0 CPU 214: 0 (HSC0), 0-11 (HSC1-2)

Description of operation:

The High-speed Counter Definition (HDEF) instruction assigns a MODE to the referenced high-speed counter (HSC). Only one HDEF box may be used per counter.

Example:

```
      LD    SM0.0
      MOVB  16#F8, SMB47
      HDEF  1, 11
      MOVD  0, SMD48
      MOVD  50, SMD52
      ATCH  0, 13
      ENI
      HSC   1
```

High-speed Counter (STL)

Format:

```
HSC   N
```

Operands:

N (word):	CPU 212: 0 CPU 214: 0-2

Description of operation:

The High-speed Counter (HSC) instruction invokes the operation defined by the special memory bit for the referenced high-speed counter. The parameter *N* specifies the high-speed counter number.

Example:

```
      LD    SM0.0
      MOVB  16#F8, SMB47
      HDEF  1, 11
      MOVD  0, SMD48
      MOVD  50, SMD52
      ATCH  0, 13
      ENI
      HSC   1
```

Pulse (STL)

Format:

```
PLS    x
```

Operands:

x (word):	CPU 214: 0-1

Description of operation:

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

Example:

```
LD     SM0.0
MOVB   16#85, SMB67
MOVW   500, SMW68
MOVD   4, SMD72
ATCH   3, 19
ENI
PLS    0
```

Transmit (STL)

Format:

```
XMT    TABLE, PORT
```

Operands:

TABLE (byte):	VB, IB, QB, MB, SMB, *VD, *AC
PORT (byte):	0

Description of operation:
The Transmit (XMT) 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. It must always be 0.

Example:

```
LD     M6.3
A      SM4.5
XMT    *VD100, 0
```

Add Integer (STL)

Format:

```
+I     IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Add Integer (+I) instruction adds two 16-bit integers (IN1, IN2), and produces a 16-bit result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD     I4.0
+I     AC1, AC0
MUL    AC1, VD100
DIV    VW10, VD200
```

Subtract Integer (STL)

Format:

```
-I     IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Subtract Integer (-I) instruction subtracts two 16-bit integers (IN1, IN2), and produces a 16-bit result (IN2), as is shown in the equation:

IN2 - IN1 = IN2

Example:

```
LD     I4.0
-I     AC1, AC0
MUL    AC1, VD100
DIV    VW10, VD200
```

Add Double Integer (STL)

Format:

+D IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Add Double Integer (+D) instruction adds two 32-bit integers (IN1, IN2), and produces a 32bit result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD    I4.0
+D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Subtract Double Integer (STL)

Format:

-D IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Subtract Double Integer (-D) instruction subtracts two 32-bit integers (IN1, IN2), and produces a 32-bit result (IN2), as is shown in the equation:

IN2 - IN1 = IN2

Example:

```
LD    I4.0
-D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Add Real (STL)

Note: CPU 214 only.

Format:

+R IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:

The Add Real (+R) instruction adds two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD    I4.0
+R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Subtract Real (STL)

Note: CPU 214 only.

Format:

-R IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Subtract Real (-R) instruction subtracts two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (IN2), as is shown in the equation:

IN2 - IN1 = IN2

Example:

```
LD    I4.0
-R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Multiply Real (STL)

Note: CPU 214 only.

Format:

```
*R    IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Real (*R) instruction multiplies two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number product (IN2), as is shown in the equation:

IN1 * IN2 = IN2

Example:

```
LD    I4.0
*R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Divide Real (STL)

Note: CPU 214 only.

Format:

```
/R    IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Divide Real (/R) instruction divides two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number quotient (IN2), as is shown in the equation:

IN2 / IN1 = IN2

Example:

```
LD    I4.0
/R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Multiply Integer (STL)

Format:

```
MUL   IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Integer (MUL) instruction multiplies a 16-bit integer (IN1) by the least-significant 16 bits of a 32-bit integer (IN2) and produces a 32-bit result (IN2), as is shown in the equation:

IN1 * IN2 = IN2

Example:

```
LD    I4.0
+D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Divide Integer (STL)

Format:

```
DIV   IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Divide Integer (DIV) instruction divides a 16-bit integer (IN1) into the least-significant 16 bits of a 32-bit integer (IN2) and produces a 32-bit result (IN2) composed of a 16-bit quotient (least significant) and a 16-bit remainder (most significant), as is shown in the equation:

IN2 / IN1 = IN2

Example:

```
LD    I4.0
+D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Square Root (STL)

Note: CPU 214 only.

Format:

```
SQRT  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

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

Example:

```
LD    I4.0
SQRT  AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Block Move Byte (STL)

Format:

```
BMB   IN, OUT, N
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Block Move Byte (BMB) 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.

Example:

```
LD    I2.1
BMB   VB20, VB100, 4
FILL  0, VW200, 10
```

Block Move Word (STL)

Format:

```
BMW   IN, OUT, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Block Move Word (BMW) 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.

Example:

```
LD    I2.1
BMW   VW20, VW100, 4
FILL  0, VW200, 10
```

Memory Fill (STL)

Format:

```
FILL  IN, OUT, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Memory Fill (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.

Example:

```
LD    I2.1
BMW   VW20, VW100, 4
FILL  0, VW200, 10
```

Move Byte (STL)

Format:

```
MOVB  IN, OUT
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Move Byte (MOVB) instruction moves the input byte (IN) to the output byte (OUT). The input byte is not altered by the move.

Example:

```
LD    I2.1
MOVB  VB50, AC0
SWAP  AC0
```

Move Double Word (STL)

Format:

```
MOVD  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC, &VB, &IB, &QB, &MB, &T, &C
OUT (Dword):	OUT: VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Move Double Word (MOVD) instruction moves the input double word (IN) to the output double word (OUT). The input double word is not altered by the move.

Example:

```
LD    I2.1
MOVD  VD50, AC0
SWAP  AC0
```

Move Real (STL)

Note: CPU 214 only.

Format:

```
MOVR  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Move Real (MOVR) 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.

Example:

```
LD    I2.1
MOVR  VD50, AC0
SWAP  AC0
```

Move Word (STL)

Format:

```
MOVW  IN, OUT
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:

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

Example:

```
LD    I2.1
MOVW  VW50, AC0
SWAP  AC0
```

Swap Bytes (STL)

Format:
```
SWAP   IN
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Swap Bytes (SWAP) instruction exchanges the most-significant byte with the least-significant byte of the word (IN).

Example:
```
LD     I2.1
MOVR   VD50, AC0
SWAP   AC0
```

Network Read (STL)

Note: Network instructions are supported by the CPU 214 only.

Format:
```
NETR   t, p
```

Operands:

t:	VB, MB, *VD, *AC
p:	Constant (CPU 214: 0)

Description of operation:
The Network Read (NETR) instruction initiates a communication operation to gather data from a remote device through the specified port (p), as defined in the description table (t). The format of the description table is CPU-specific.

You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to 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 NETR and four NETW instructions, or two NETR and six NETW instructions.

Example:
```
LDN    SM0.1
AN     V200.6
AN     V200.5
MOVB   2, VB201

MOVD   &VB100, VD202
MOVB   3, VB206
NETR   VB200, 0
```

Network Write (STL)

Note: Network instructions are supported by the CPU 214 only.

Format:
```
NETW   t, p
```

Operands:

t:	VB, MB, *VD, *AC
p:	Constant (CPU 214: 0)

Description of operation:
The Network Write (NETW) instruction initiates a communication operation to write data to a remote device through the specified port (p), as defined in the description table (t).

You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to 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 NETR and four NETW instructions, or two NETR and six NETW instructions.

Example:
```
LD     V200.7
AW=    VW208, 100
MOVB   2, VB301
MOVD   &VB101, VD302
MOVB   2, VB306
MOVW   0, VW307
NETW   VB300, 0
```

Subroutine Call (STL)

Format:
```
CALL   n
```

Operands:

n:	CPU 212: 0-15 CPU 214: 0-63

Description of operation:
The Subroutine Call (CALL) instruction transfers control to the subroutine (n).

Example:
```
LD     SM0.1
CALL   10
```

Conditional Return from Subroutine (STL)

Format:

```
CRET
```

Operands:

(none)

Description of operation:
The Conditional Return from Subroutine (CRET) instruction may be used to terminate a subroutine, based on the condition of the preceding logic.

Example:

```
LD    M14.3
CRET
```

Conditional End (STL)

Format:

```
END
```

Operands:

(none)

Description of operation:

The Conditional End (END) instruction terminates the main user program based on the condition of the preceding logic.

Example:

```
LD    SM5.0
STOP
END
```

For (STL)

Format:

```
FOR   INDEX, INITIAL, FINAL
```

Operands:

INDEX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
INITIAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
FINAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

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

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 incrementing 1,2,3, . . 10.

Example:

```
LD    I2.1
FOR   VW225, 1, 2
.
.
.
NEXT
```

Jump to Label (STL)

Format:

```
JMP   n
```

Operands:

n:	CPU 212: 0-63 CPU 214: 0-255

Description of operation:

The Jump to Label (JMP) instruction performs a branch to the specified label within the program.

Example:

```
LDN   SM0.2
JMP   4
.
.
.
LBL   4
```

Label (STL)

Format:

```
LBL    n
```

Operands:

n:	CPU 212: 0-63 CPU 214: 0-255

Description of operation:

The Label (LBL) instruction marks the location of the jump destination (n). The CPU 212 allows 64 labels, and the CPU 214 allows 256.

Example:

```
LDN    SM0.2
JMP    4
.
.
.
LBL    4
```

Main Program End (STL)

Format:

```
MEND
```

Operands:

(none)

Description of operation:

The Main Program End (MEND) instruction must be used to terminate the main user program.

Example:

```
.
.
.
MEND

SBR    10

LD     M14.3
CRET
.
.
.
```

Next (STL)

Format:

```
NEXT
```

Operands:

(none)

Description of operation:

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

Example:

```
LD     I2.1
FOR    VW225, 1, 2
.
.
.
NEXT
```

No Operation (STL)

Format:

```
NOP    N
```

Operands:

N:	0-255

Description of operation:

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

Example:

```
LDN    SM0.2
JMP    4

NOP

LBL    4
```

Unconditional Return from Subroutine (STL)

Format:

```
RET
```

Operands:

(none)

Description of operation:

The Unconditional Return from Subroutine (RET) instruction must be used to terminate each subroutine.

Example:

```
      SBR    10

      LD     M14.3
      CRET

      RET
```

Subroutine (STL)

Format:

```
SBR    n
```

Operands:

n:	CPU 212: 0-15 CPU 214: 0-63

Description of operation:

The Subroutine (SBR) instruction marks the beginning of the subroutine (n). The CPU 212 supports 16 subroutines, and the CPU 214 supports 64.

Example:

```
      .
      .
      .
      MEND

      SBR    10

      LD     M14.3
      CRET
      .
      .
      .
```

Stop (STL)

Format:

```
STOP
```

Operands:
(none)

Description of operation:
The Stop (STOP) instruction terminates execution of the user program by causing a transition to the Stop mode.

Example:

```
      LD     SM5.0
      STOP
```

Watchdog Reset (STL)

Format:

```
WDR
```

Operands:
(none)

Description of operation:
The Watchdog Reset (WDR) instruction allows the watchdog timer to be retriggered. This extends the time the scan is allowed to take without getting a watchdog error.

Example:

```
      LD     M5.6
      WDR
```

Rotate Left Double Word (STL)

Format:

```
RLD    IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Rotate Left Double Word (RLD) instruction rotates the double word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if IN = 0
SM1.1 (overflow)	= 1 if last bit rotated = 1

Example:

```
      LD     I4.0
      RLD    AC0, 2
      SLW    VW200, 3
```

Rotate Left Word (STL)

Format:

```
RLW    IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Rotate Left Word (RLW) instruction rotates the word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit rotated = 1

Example:

```
LD     I4.0
RLD    AC0, 2
RLW    VW200, 3
```

Rotate Right Double Word (STL)

Format:

```
RRD    IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Rotate Right Double Word (RRD) instruction rotates the double word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if IN = 0
SM1.1 (overflow)	= 1 if last bit rotated = 1

Example:

```
LD     I4.0
RRD    AC0, 2
SLW    VW200, 3
```

Rotate Right Word (STL)

Format:

```
RRW    IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Rotate Right Word (RRW) instruction rotates the word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit rotated = 1

Example:

```
LD     I4.0
RRW    AC0, 2
SLW    VW200, 3
```

Shift Register Bit (STL)

Format:

```
SHRB   DATA, S_BIT, N
```

Operands:

DATA, S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

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

Example:

```
LD     I0.2
EU
SHRB   I0.3, V100.0, 4
```

Shift Left Double Word (STL)

Format:

```
SLD   IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Left Double Word (SLD) instruction shifts the double word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if IN = 0
SM1.1 (overflow)	= 1 if last bit shifted out = 1

Example:

```
LD    I4.0
SLD   AC0, 2
SLW   VW200, 3
```

Shift Left Word (STL)

Format:

```
SLW   IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Left Word (SLW) instruction shifts the word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit shifted out = 1

Example:

```
LD    I4.0
RLD   AC0, 2
SLW   VW200, 3
```

Shift Right Double Word (STL)

Format:

```
SRD   IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Right Double Word (SRD) instruction shifts the double word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if IN = 0
SM1.1 (overflow)	= 1 if last bit shifted out = 1

Example:

```
LD    I4.0
SRD   AC0, 2
SLW   VW200, 3
```

Shift Right Word (STL)

Format:

```
SRW   IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Right Word (SRW) instruction shifts the word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero)	= 1 if OUT = 0
SM1.1 (overflow)	= 1 if last bit shifted out = 1

Example:

```
LD    I4.0
RLD   AC0, 2
SRW   VW200, 3
```

Add To Table (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
ATT   DATA, TABLE
```

Operands:

DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC

Description of operation:
The Add To Table (ATT) instruction adds word values (DATA) to the table (TABLE). The first value of the table is the maximum table length (TL). The second value is the entry count (EC) that specifies the number of entries in the table. New data are added to the table after the last entry. Each time new data are added to the table, the entry count (EC) is incremented. If you try to overfill the table, the Table Full memory bit (SM1.4) is set.

Example:

```
      LD    I3.0
      ATT   VW100, VW200
```

First In First Out (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FIFO  TABLE, DATA
```

Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The First In First Out (FIFO) instruction removes the first entry in the table (TABLE), and outputs the value to the location DATA. All other entries of the table are shifted up one location. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

Example:

```
      LD    I3.0
      FIFO  VW200, VW300
```

Find Less Than (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND<  SRC, PATRN, INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Find Less Than (FND<) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.
If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Example:

```
      LD    I3.0
      FND<  VW202, 16#3130, AC1
```

Find Not Equal To (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND<> SRC, PATRN, INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Find Not Equal To (FND<>) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.
If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Example:

```
      LD    I3.0
      FND<> VW202, 16#3130, AC1
```

Find Equal To (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND=  SRC, PATRN, INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Equal To (FND=) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

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

Example:

```
LD    I3.0
FND=  VW202, 16#3130, AC1
```

Find Greater Than (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND>  SRC, PATRN, INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Greater Than (FND>) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

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

Example:

```
LD    I3.0
FND=  VW202, 16#3130, AC1
```

Last In First Out (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
LIFO  TABLE, DATA
```

Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:

The Last In First Out (LIFO) instruction removes the last entry in the table (TABLE), and outputs the value to the location DATA. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

Example:

```
LD    I3.0
LIFO  VW200, VW300
```

On Delay Timer (STL)

Format:
TON Txxx, PT

Operands:

Txxx (word):	CPU 212: 32-63 CPU 214: 32-63, 96-127
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The On-Delay Timer (TON) times up to the maximum value when the top of stack =1. When the current value (Txxx) is >= the Preset Time (PT), the timer bit (Txxx) turns on. It resets when the top of stack =0. Timing stops upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T32	T96
10 ms	T33-T36	T97-T100
100 ms	T37-T63	T101-T127

Example:

```
LD    I2.0
TON   T33, 3
```

Retentive On Delay Timer (STL)

Format:
TONR Txxx, PT

Operands:

Txxx (word):	CPU 212: 0-31 CPU 214: 0-31, 64-95
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Retentive On Delay Timer (TONR) times up to the maximum value when the top of stack =1. When the current value (Txxx) is >= the Preset Time (PT), the timer bit (Txxx) turns on. Timing stops when the top of stack =0, or upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T0	T64
10 ms	T1-T4	T65-T68
100 ms	T5-T31	T69-T95

Example:

```
LD    I2.1
TONR  T2, 10
```

AND Word (STL)

Format:

ANDW *IN1, IN2*

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The AND Word (ANDW) instruction logically ANDs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
LD    I4.0
ANDW  AC1, AC0
```

OR Word (STL)

Format:

ORW *IN1, IN2*

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The OR Word (ORW) instruction logically ORs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
LD    I4.0
ORW   AC1, VW100
```

Exclusive OR Word (STL)

Format:

```
XORW  IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The Exclusive OR Word (XORW) instruction logically XORs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
      LD    I4.0
      XORW  AC1, VW100
```

AND Double Word (STL)

Format:

```
ANDD  IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The AND Dword (ANDD) instruction logically ANDs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
      LD    I4.0
      ANDD  AC1, AC0
```

OR Double Word (STL)

Format:

```
ORD   IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The OR Dword (ORD) instruction logically ORs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
      LD    I4.0
      ORD   AC1, VD100
```

Exclusive OR Double Word (STL)

Format:

```
XORD  IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The Exclusive OR Dword (XORD) instruction logically XORs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
      LD    I4.0
      XORD  AC1, VD100
```

Increment Word (STL)

Format:

INCW *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:
The Increment Word (INCW) instruction adds 1 to the input word value IN, and loads the result in that word.

IN + 1 = IN

Example:

```
LD    I4.0
INCW  AC0
```

Decrement Word (STL)

Format:
DECW *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:
The Decrement Word (DECW) instruction subtracts 1 from the input word value IN, and loads the result in that word.

IN - 1 = IN

Example:

```
LD    I4.0
DECW  VW100
```

Increment Double Word (STL)

Format:

INCD *IN*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:
The Increment Dword (INCD) instruction adds 1 to the input double word value IN, and loads the result in that double word.
IN + 1 = IN
Example:

```
LD    I4.0
INCD  AC0
```

Decrement Double Word (STL)

Format:
DECD *IN*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:
The Decrement Dword (DECD) instruction subtracts 1 from the input double word value IN, and loads the result in that double word.

IN - 1 = IN

Example:

```
LD    I4.0
DECD  VD100
```

Invert Word (STL)

Format:

INVW *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:
The Invert Word (INVW) instruction takes the Ones Complement of the input word value IN, and loads the result in that word.
Example:

```
LD    I4.0
INVW  AC0
```

Invert Double Word (STL)

Format:

INVD *IN*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The Invert Dword (INVD) instruction takes the ones complement of the input double word value IN, and loads the result in that double word.
Example:

```
LD    I4.0
INVD  AC0
```

-Specifications

CPU 212 DC

General Features

Physical Size (L x W x D)	160 x 80 x 62 mm	6.3 x 3.15 x 2.4 in
Weight	.31 kg	.68 lbs
User Program Size / Storage	512 Words / EEPROM	
User Data Size / Storage	512 Words / RAM	
Data Retention	50 hr typical	
Local I/O	8 Inputs / 6 Outputs	
Total I/O (maximum)	30 points	
Maximum Number of Expansion Modules	2	
Boolean Execution Speed	1.3 µs / Instruction	
Internal Memory Bits	128	
Timers	64	
Counters	64	
High-Speed Counters	1 Software (2 kHz max.)	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Power Supply

Voltage Range	20.4 - 28.8 VDC
Input Current	60 mA typical, CPU only
	500 mA maximum load
UL/CSA Rating	50 VA
Hold Up Time	10 ms minimum from 24 VDC
Inrush Current	10 A peak at 28.8 VDC
Fusing (non-replaceable	1 A, 125 V, Slow Blow
5 VDC Available Current	340 mA
Isolated	No

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	5 W at 1.75 A load
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter

CPU 212 DC, DC In, DC Out

Input Points

Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	15-30 VDC, 4 mA minimum 35 VDC, 500 mS surge
ON State Nominal	24 VDC, 7 mA
OFF State Maximum	5 VDC, 1 mA
Response Time	
I0.0 - I0.7	3.5 mS typical / 4.5 mS maximum
I0.0 as used by Interrupt or HSC0	30 µS typical / 70 µS maximum
Optical Isolation	500 VAC, 1 minute

Output Points

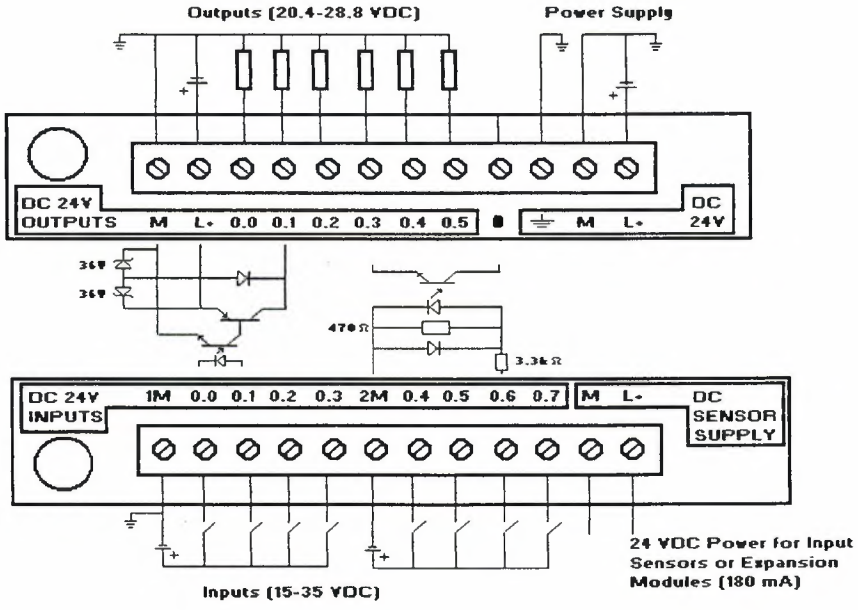
Output Type	Sourcing Transistor	
Voltage Range	20.4 - 28.8 VDC	
Maximum Load Current*	0-40° C	55° C
Per Single Point	0.75 A	0.50 A
Per 2 Adjacent Points	1.00 A	0.75 A
All Points Total	2.25 A	1.75 A

*Linear Derate 40 - 55 ° C
Vertical Mount Derate 10° C

Inductive Load Clamping	Per Common
Single Pulse	2 A L/R = 10 ms 1 A L/R = 100 ms
Repetitive	1 W energy dissipation (1/2 Li^2 x switch rate < 1 W)
Leakage Current	100 µA
Switching Delay	25 µS ON, 120 µS OFF
Voltage Drop	4 A, 100 ms
Optical Isolation	500 VAC, 1 minute

DC Sensor Supply

Voltage Range	16.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	Same as supplied Voltage
24 VDC Available Current	180 mA
Isolated	No



CPU 212 DC, DC In, DC Out Terminal Connector

CPU 212 AC

General Features

Physical Size (L x W x D)	160 x 80 x 62 mm	6.3 x 3.15 x 2.4 in
Weight	.39 kg	.86 lbs
User Program Size / Storage	512 Words / EEPROM	
User Data Size / Storage	512 Words / RAM	
Data Retention	50 hr typical	
Local I/O	8 Inputs / 6 Outputs	
Total I/O (maximum)	30 points	
Maximum Number of Expansion Modules	2	
Boolean Execution Speed	1.3 µs / Instruction	
Internal Memory Bits	128	
Timers	64	
Counters	64	
High-Speed Counters	1 Software (2 kHz max.)	
Agency Approvals	UL 508 CSA C22.2 142 VDE 0160 compliant	

Power Supply

Voltage Range	85-264 VAC at 47-63 Hz
Input Current	4 VA typical, CPU only 50 VA maximum load
Hold Up Time	20 ms minimum from 110 VAC
Inrush Current	20 A peak at 264 VAC
Fusing (non-replaceable	2 A, 250 V, Slow Blow
5 VDC Available Current	340 mA
Isolated	Yes - Transformer, 1500 VAC, 1 minute

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	6 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter

CPU 212 AC, DC In, Relay Out

Input Points

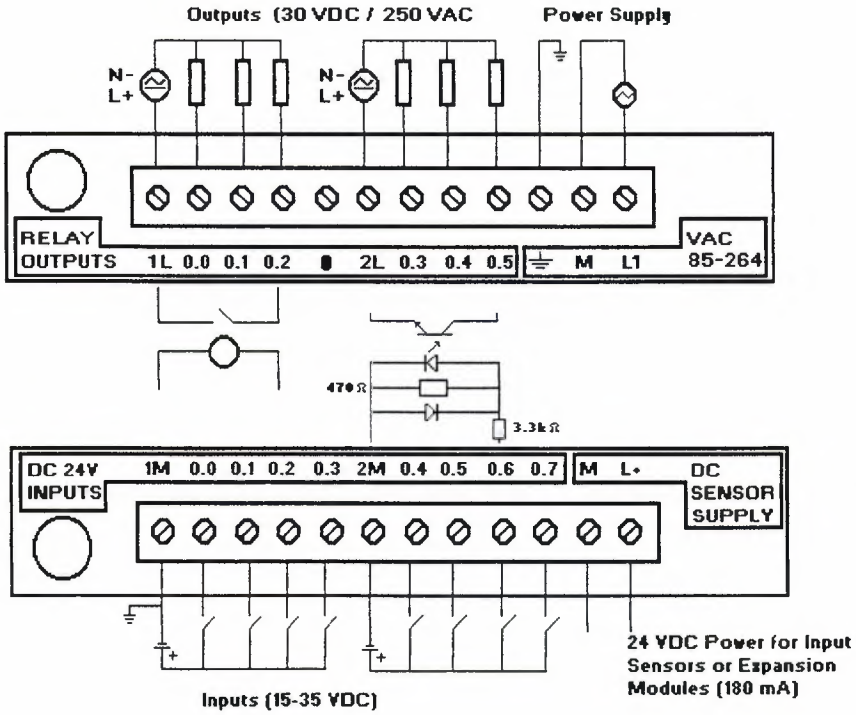
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	15-30 VDC, 4 mA minimum 35 VDC, 500 mS surge
ON State Nominal	24 VDC, 7 mA
OFF State Maximum	5 VDC, 1 mA
Response Time	
I0.0 - I0.7	3.5 mS typical / 4.5 mS maximum
I0.0 as used by Interrupt or HSC0	30 µS typical / 70 µS maximum
Optical Isolation	500 VAC, 1 minute

Output Points

Output Type	Relay, dry contact
Voltage Range	5 - 30 VDC / 250 VAC
Maximum Load Current*	2 A / Point
Overcurrent Surge	7 A with contacts closed
Isolation Resistance	100 MOhm minimum
Switching Delay	10 ms maximum
Lifetime	10,000,000 Mechanical 100,000 with Rated Load
Contact Resistance	200 MOhm maximum
Isolation	
Coil to Contact	1500 VAC, 1 minute
Contact to Contact	1000 VAC, 1 minute

DC Sensor Supply

Voltage Range	20.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	1 V peak-to-peak maximum
24 VDC Available Current	180 mA
Isolated	No



CPU 212 AC, DC In, Relay Out Terminal Connector

CPU 212 AC, AC In, AC Out

Input Points

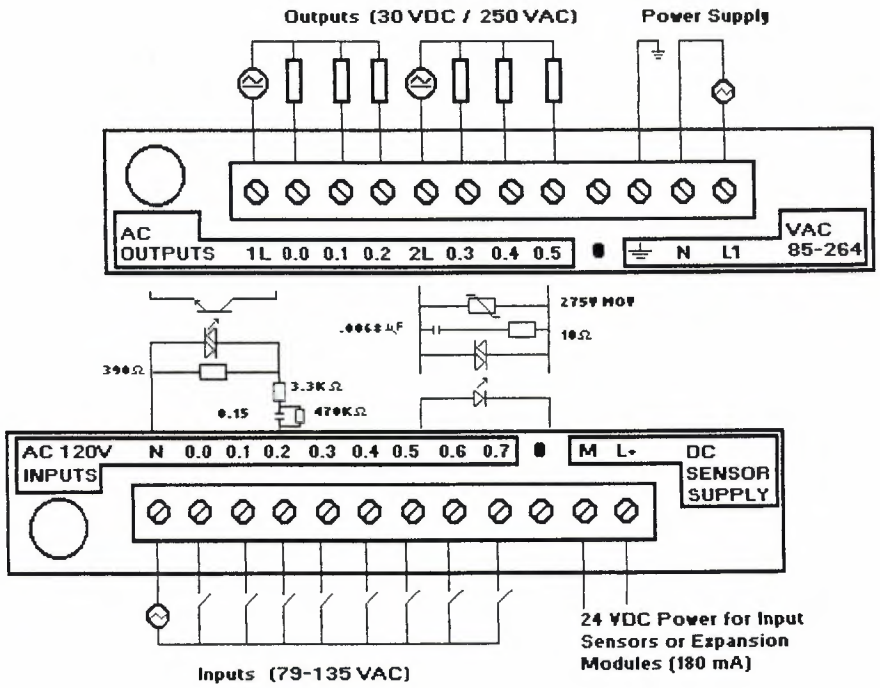
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	79 - 135 VAC, 47-63 Hz 4 mA minimum
ON State Nominal	120 VAC, 60 Hz, 7 mA
OFF State Maximum	20 VAC, 1 mA
Response Time	10 typical, 15 ms maximum
Optical Isolation	1500 VAC, 1 minute

Output Points

Output Type	Triac, zero-crossing	
Voltage / Frequency Range	20 - 264 VAC, 47 - 63 Hz	
Load Circuit Power Factor	0.3 to 1.0	
Inductive Load Clamping	MOV 275 V working voltage	
Maximum Load Current	0 - 40° C	55° C
Per Single Point	1.20 A	1.00 A
Per 2 Adjacent Points	1.50 A	1.25 A
All Points Total	3.50 A	2.50 A
*Linear Derate 40 - 55 ° C		
Vertical Mount Derate 10° C		
Maximum Load Current	10 mA	
Leakage Current	2.5 mA, 120 VAC / 2.0 mA, 240 VAC	
Switching Delay	1/2 Cycle	
Surge Current	30 A peak, 1 cycle / 10 A peak, 5 cycle	
Voltage Drop	1.5 V maximum at maximum current	
Optical Isolation	1500 VAC, 1 minute	

DC Sensor Supply

Voltage Range	16.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	1 V peak-to-peak maximum
24 VDC Available Current	180 mA
Isolated	No



CPU 212 AC, AC In, AC Out Terminal Connector

CPU 214 DC

General Features

Physical Size (L x W x D)	197 x 80 x 62 mm	7.75 x 3.15 x 2.4 in
Weight	.39 kg	.86 lbs
User Program Size / Storage	2K Words / EEPROM	
User Data Size / Storage	2K Words / RAM	
Data Retention	190 hr typical	
Local I/O	14 Inputs / 10 Outputs	
Total I/O (maximum)	64 points	
Maximum Number of Expansion Modules	5	
Boolean Execution Speed	0.8 µs / Instruction	
Internal Memory Bits	256	
Timers	128	
Counters	128	
High-Speed Counters	1 Software (2 kHz max.)	
	2 Hardware (7 kHz max.)	
Pulse Outputs	2 (4 kHz max. each)	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Power Supply

Voltage Range	20.4 - 28.8 VDC
Input Current	85 mA typical, CPU only
	900 mA maximum load
UL/CSA Rating	50 VA
Hold Up Time	10 ms minimum from 24 VDC
Inrush Current	10 A peak at 28.8 VDC
Fusing (non-replaceable	1 A, 125 V, Slow Blow
5 VDC Available Current	660 mA
Isolated	No

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	8 W at 3 A load
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter

CPU 214 DC, DC In, DC Out

Input Points

Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	15-30 VDC, 4 mA minimum 35 VDC, 500 mS surge
ON State Nominal	24 VDC, 7 mA
OFF State Maximum	5 VDC, 1 mA
Response Time	
I0.0 - I0.7	3.5 mS typical / 4.5 mS maximum
I0.0 as used by Interrupt or HSC0	30 µS typical / 70 µS maximum
Optical Isolation	500 VAC, 1 minute

Output Points

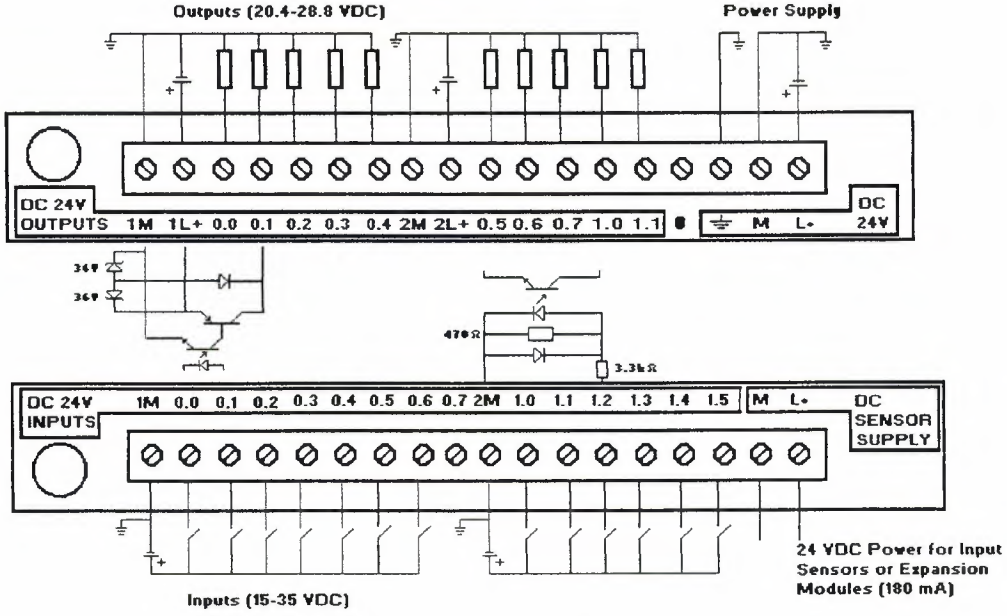
Output Type	Sourcing Transistor	
Voltage Range	20.4 - 28.8 VDC	
Maximum Load Current*	0-40° C	55° C
Per Single Point	0.75 A	0.50 A
Per 2 Adjacent Points	1.00 A	0.75 A
All Points Total	2.25 A	1.75 A

*Linear Derate 40 - 55 ° C
Vertical Mount Derate 10° C

Inductive Load Clamping	Per Common
Single Pulse	2 A L/R = 10 ms 1 A L/R = 100 ms
Repetitive	1 W energy dissipation (1/2 Li² x switch rate < 1 W)
Leakage Current	100 µA
Switching Delay	25 µS ON, 120 µS OFF
Voltage Drop	4 A, 100 ms
Optical Isolation	500 VAC, 1 minute

DC Sensor Supply

Voltage Range	16.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	Same as supplied Voltage
24 VDC Available Current	180 mA
Isolated	No



CPU 214 DC, DC In, DC Out Terminal Connector

CPU 214 AC

General Features

Physical Size (L x W x D)	197 x 80 x 62 mm	7.75 x 3.15 x 2.4 in
Weight	.49 kg	1.0 lbs
User Program Size / Storage	2K Words / EEPROM	
User Data Size / Storage	2K Words / RAM	
Data Retention	190 hr typical	
Local I/O	14 Inputs / 10 Outputs	
Total I/O (maximum)	64 points	
Maximum Number of Expansion Modules	5	
Boolean Execution Speed	0.8 µs / Instruction	
Internal Memory Bits	256	
Timers	128	
Counters	128	
High-Speed Counters	1 Software (2 kHz max.)	
	2 Hardware (7 kHz max.)	
Pulse Outputs	Not Recommended	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Power Supply

Voltage Range	85-264 VAC at 47-63 Hz
Input Current	4.5 VA typical, CPU only
	50 VA maximum load
Hold Up Time	20 ms minimum from 110 VAC
Inrush Current	20 A peak at 264 VAC
Fusing (non-replaceable	2 A, 250 V, Slow Blow
5 VDC Available Current	660 mA
Isolated	Yes - Transformer, 1500 VAC, 1 minute

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	9 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter

CPU 214 AC, DC In, Relay Out

Input Points

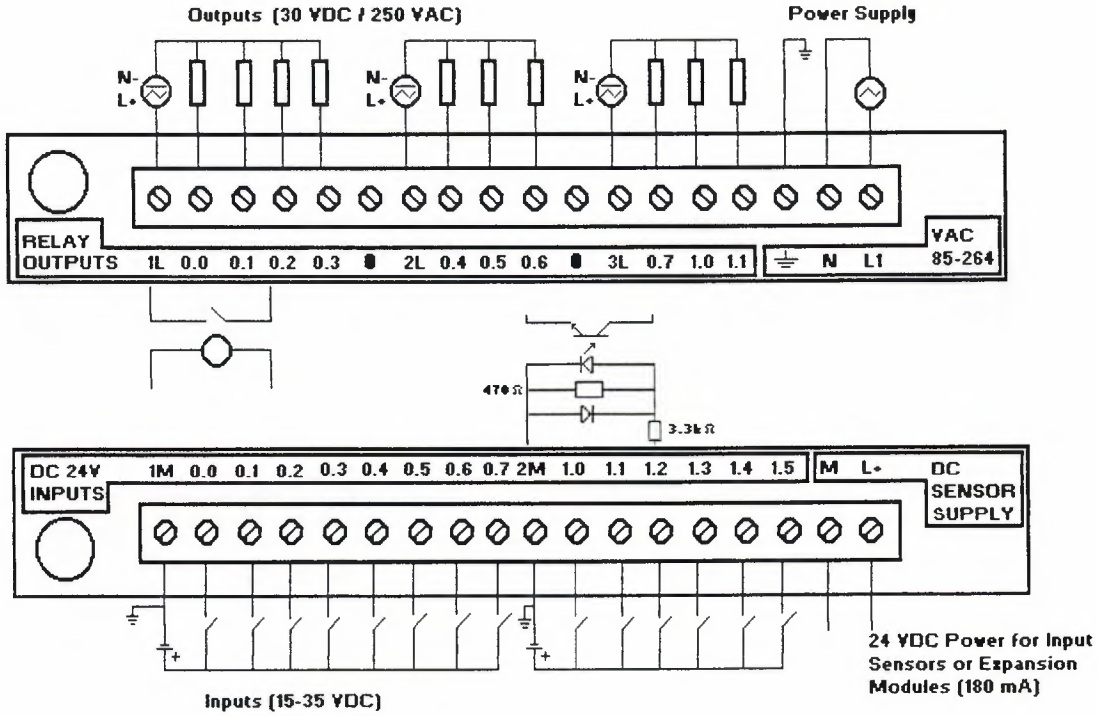
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	15-30 VDC, 4 mA minimum 35 VDC, 500 mS surge
ON State Nominal	24 VDC, 7 mA
OFF State Maximum	5 VDC, 1 mA
Response Time	
I0.0 - I0.3	0.2 ms maximum
I0.4 - I1.5	1.2 ms maximum
I0.6 - I1.5 as used by HSC1 and HSC2	30 µS typical / 70 µS maximum
Optical Isolation	500 VAC, 1 minute

Output Points

Output Type	Relay, dry contact
Voltage Range	5 - 30 VDC / 250 VAC
Maximum Load Current*	2 A / Point
Overcurrent Surge	7 A with contacts closed
Isolation Resistance	100 MOhm minimum
Switching Delay	10 ms maximum
Lifetime	10,000,000 Mechanical 100,000 with Rated Load
Contact Resistance	200 MOhm maximum
Isolation	
Coil to Contact	1500 VAC, 1 minute
Contact to Contact	1000 VAC, 1 minute

DC Sensor Supply

Voltage Range	20.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	1 V peak-to-peak maximum
24 VDC Available Current	280 mA
Isolated	No



CPU 214 AC, DC In, Relay Out Terminal Connector

CPU 214 AC, AC In, AC Out

Input Points

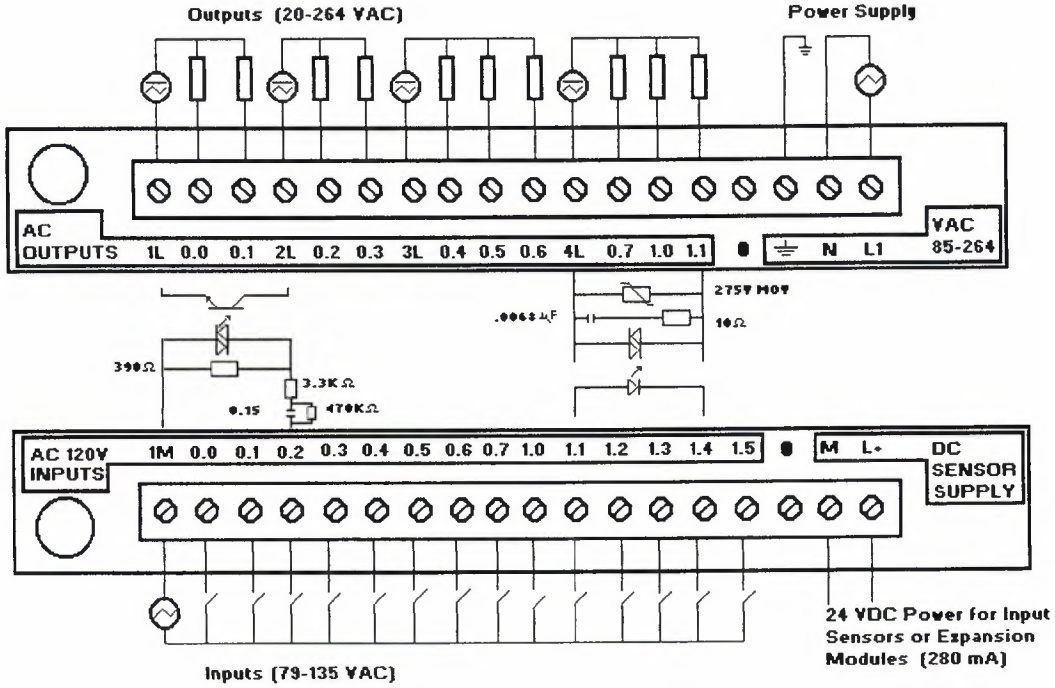
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	79 - 135 VAC, 47-63 Hz 4 mA minimum
ON State Nominal	120 VAC, 60 Hz, 7 mA
OFF State Maximum	20 VAC, 1 mA
Response Time	10 µS typical, 15 µs maximum
Optical Isolation	1500 VAC, 1 minute

Output Points

Output Type	Triac, zero-crossing	
Voltage / Frequency Range	20 - 264 VAC, 47 - 63 Hz	
Load Circuit Power Factor	0.3 to 1.0	
Inductive Load Clamping	MOV 275 V working voltage	
Maximum Load Current	0 - 40° C	55° C
Per Single Point	1.20 A	1.00 A
Per 2 Adjacent Points	1.50 A	1.25 A
All Points Total	6.00 A	4.25 A
*Linear Derate 40 - 55 ° C		
Vertical Mount Derate 10° C		
Maximum Load Current	10 mA	
Leakage Current	1.5 mA, 120 VAC / 2.0 mA, 240 VAC	
Switching Delay	1/2 Cycle	
Surge Current	30 A peak, 1 cycle / 10 A peak, 5 cycle	
Voltage Drop	1.5 V maximum at maximum current	
Optical Isolation	1500 VAC, 1 minute	

DC Sensor Supply

Voltage Range	16.4 - 28.8 VDC
Ripple/Noise (<10 MHz)	1 V peak-to-peak maximum
24 VDC Available Current	280 mA
Isolated	No



CPU 214 AC, AC In, AC Out Terminal Connector

Digital Input, 8 Point, 24 VDC

General Features

Physical Size (L x W x D)	90 x 80 x 62 mm	3.54 x 3.15 x 2.4 in
Weight	.17 kg	.37 lbs
Points	8 Inputs	
Agency Approvals	UL 508 CSA C22.2 142 VDE 0160 compliant	

Input Points

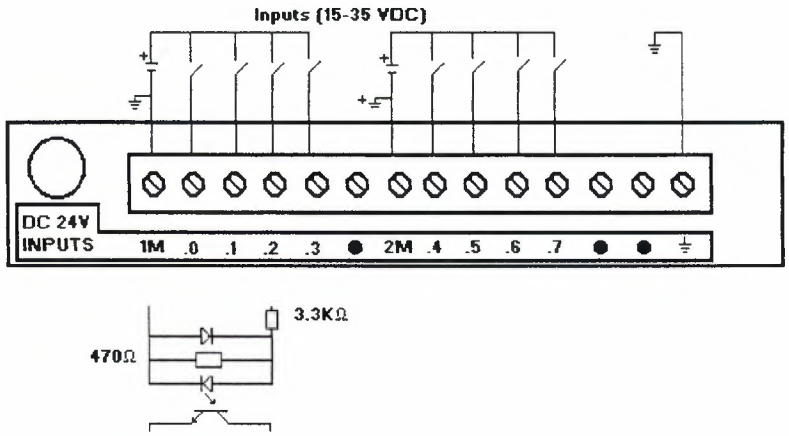
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	15 - 30 VDC, 4 mA minimum 35 VDC, 500 ms surge
ON State Nominal	24 VDC, 7 mA
OFF State Maximum	5 VDC, 1 mA
Response Time	35 ms typical, 4.5 ms maximum
Optical Isolation	500 VAC, 1 minute

Current Requirements

5 VDC Logic Current	60 mA from Base Unit
24 VDC Sensor Current	60 mA from Base Unit or External Power Supply

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	2 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter



Expansion Module, 24 VDC, 8 Point Input

Digital Input, 8 Point, 120 VAC

General Features

Physical Size (L x W x D)	90 x 80 x 62 mm	3.54 x 3.15 x 2.4 in
Weight	.18 kg	.39 lbs
Points	8 Inputs	
Agency Approvals	UL 508	
	CSA C22.2 142	

Input Points

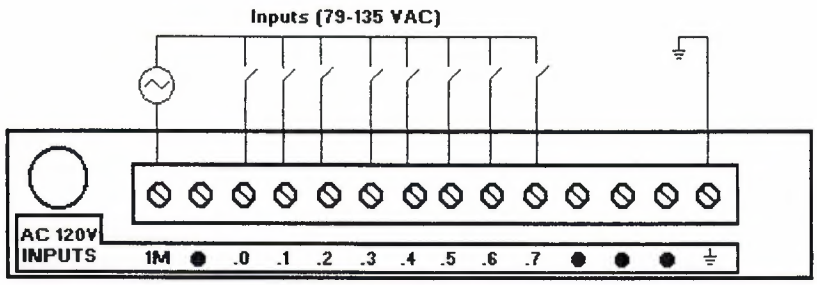
Input Type (IEC 1131-2)	Type 1 Sinking
ON State Range	79 - 135 VAC, 47 - 63 Hz
	4 mA minimum
ON State Nominal	120 VAC, 60 Hz, 7 mA
OFF State Maximum	20 VAC, 1 mA
Response Time	15 ms maximum
Optical Isolation	1500 VAC, 1 minute

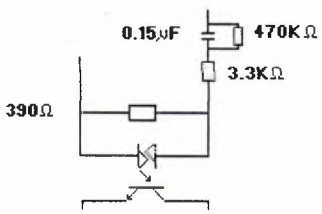
Current Requirements

5 VDC Logic Current	70 mA from Base Unit

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	2 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter





Expansion Module, 120 VAC, 8 Point Input

Digital Output, 8 Points, 24 VDC

General Features

Physical Size (L x W x D)	90 x 80 x 62 mm	3.54 x 3.15 x 2.4 in
Weight	.18 kg	.39 lbs
Points	8 Outputs	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Output Points

Output Type	Sourcing Transistor	
Voltage Range	20.4 - 28.8 VDC	
Maximum Load Current	0 - 40° C	55° C*
Per Single Point	0.75 A	0.50 A
Per 2 Adjacent Points	1.00 A	0.75 A
All Points Total	4.00 A	3.00 A

*Linear Derate 40 - 55 ° C
Vertical Mount Derate 10° C

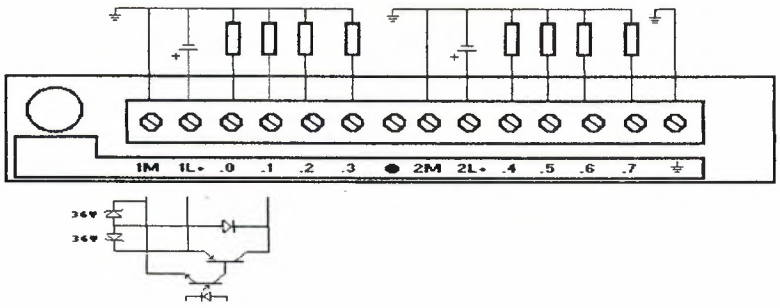
Inductive Load Clamping	Per Common
Single Pulse	2 A L/R = 10 ms
	1 A L/R = 100 ms
Repetitive	2W energy dissipation
	(1/2 Li^2 x switch rate < 1 W)
Leakage Current	100 µA
Switching Delay	50 µs ON, 200 µs OFF
Surge Current	4 A, 100 ms
Voltage Drop	1.8 V maximum at maximum current
Optical Isolation	500 VAC, 1 minute

Current Requirements

5 VDC Logic Current	80 mA from Base Unit
Output Point Current	Supplied by user at module common

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	2 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter



Expansion Module, 24 VDC, 8 Point Output

Digital Output, 8 Point, Relay

General Features

Physical Size (L x W x D)	90 x 80 x 62 mm	3.54 x 3.15 x 2.4 in
Weight	.20 kg	.44 lbs
Points	8 Outputs	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Output Points

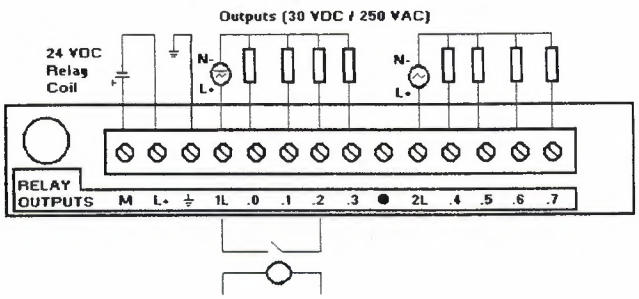
Output Type	Relay, dry contact
Voltage Range	5 - 30 VDC / 250 VAC
Maximum Load Current*	2 A / Point
Overcurrent Surge	7 A with contacts closed
Isolation Resistance	100 MOhm minimum
Switching Delay	10 ms maximum
Lifetime	10,000,000 Mechanical
	100,000 with Rated Load
Contact Resistance	200 MOhm maximum
Isolation	
Coil to Contact	1500 VAC, 1 minute
Contact to Contact	1000 VAC, 1 minute

Current Requirements

5 VDC Logic Current	80 mA from Base Unit
24 VDC Coil Current	85 mA from Base Unit or
	External Power Supply
Output Point Current	Supplied by user at module common

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	3 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter



Expansion Module, Relay, 8 Point Output

Digital Output, 8 Point, 120/230 VAC

General Features

Physical Size (L x W x D)	90 x 80 x 62 mm	3.54 x 3.15 x 2.4 in
Weight	.20 kg	.44 lbs
Points	8 Outputs	
Agency Approvals	UL 508	
	CSA C22.2 142	
	VDE 0160 compliant	

Output Points

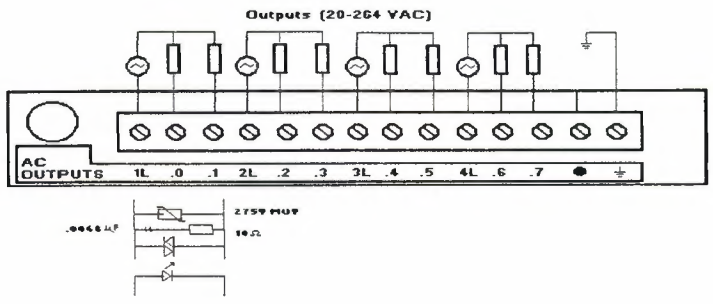
Output Type	Triac, zero-cross turn on	
Voltage / Frequency Range	20 - 264 VAC, 47 - 63 Hz	
Load Circuit Power Factor	0.3 to 1.0	
Inductive Load Clamping	MOV 275 V working voltage	
Maximum Load Current	0 - 40° C	55° C
Per Single Point	1.20 A	1.00 A
Per 2 Adjacent Points	1.50 A	1.25 A
All Points Total	4.75 A	3.50 A
*Linear Derate 40 - 55 ° C		
Vertical Mount Derate 10° C		
Maximum Load Current	10 mA	
Leakage Current	1.5 mA, 120 VAC / 2.0 mA, 240 VAC	
Switching Delay	1/2 Cycle	
Surge Current	30 A peak, 1 cycle / 10 A peak, 5 cycle	
Voltage Drop	1.5 V maximum at maximum current	
Optical Isolation	1500 VAC, 1 minute	

Current Requirements

5 VDC Logic Current	120 mA from Base Unit
Output Point Current	Supplied by user at module common

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	5 W at 3.5 A load
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity	IEC 801-4 / 2kV
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter



Expansion Module, 120/230 VAC, 8 Point Output

Memory Cartridge Specification

General Features

Physical Size (L x W x D)	28 x 10 x 16 mm	1.1 x 0.4 x 0.6 in
Weight	3.5 kg	0.01 lbs
Memory Type	EEPROM	
User Storage	4096 bytes Program 512 bytes User Data	
Agency Approvals	UL 508 CSA C22.2 142 VDE 0160 compliant	

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	0.5 mW
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Static Discharge Immunity	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity	IEC 801-3 / 10 V/meter

PC/PPI Cable Specification

General Features

Cable Length	5 meters	
Connector Size (L x W x D)	5 x 3.2 x 1.5 cm	
Weight	.304 kg	0.67 lbs
Connector Type PC	9 pin Sub D (socket)	
Connector Type PLC	9 pin Sub D (pins)	
Cable Type	RS232 to RS485 non-isolated	
Baud Rate (Supported by dip switch)	38.4K, 19.2K, 9.6K, 2.4K, 1.2K	
Supply Current	20 mA idle, 40 mA peak Supplied by communication port connector	
Agency Approvals	UL 508 CSA C22.2 142 VDE 0160 compliant	

Environmental

Operating Temperature Range	0° - 55° C
Power Dissipation	0.5 W
Storage Temperature Range	-40° - 70° C
Maximum Humidity	95% non-condensing
Vibration	IEC 68-2-6 .35 mm Amplitude / 2G maximum
Conducted Noise Immunity*	IEC 801-4 / 2kV
Static Discharge Immunity*	IEC 801-2 / 4 kV contact / 8 kV air
Radiated Noise Immunity*	IEC 801-3 / 10 V/meter

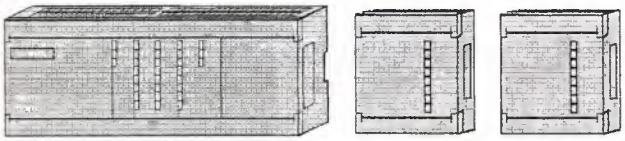
* Temporary errors in data communication may occur

(Symbolic Name)

A symbolic name is an alpha-numeric tag which is attached to a specific memory location. Symbolic addressing enables you to create a symbolic data base that identifies a specific absolute address with a symbolic name.

For example: you can associate output Q4.0 with the symbolic name MOTOR_ON. Each symbolic name must be unique. When you define symbols, you indicate a data type such as Boolean, byte, word, double-word, integer, etc.

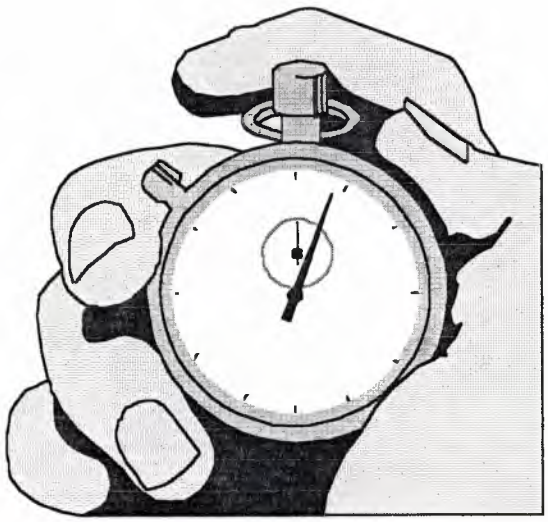
(Modular I/O Expansion)



This practical feature allows you to expand your performance conveniently, module by module. Every S7-200 CPU includes a compliment of built-in I/O. For larger applications requiring either additional I/O points, or a mixture of voltage levels, the S7-200 family includes a range of plug-in discrete and analog I/O expansion modules.

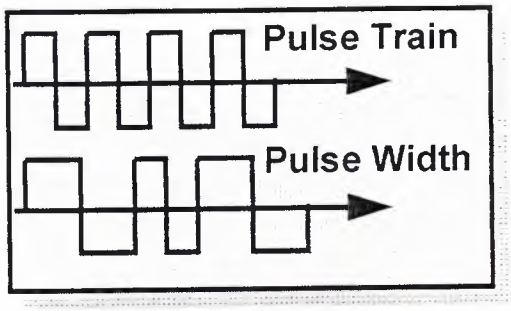
(Real-time Clock)

It is extremely practical to be able to include a time-stamp on printouts, to record machine operating hours or pre-warm an oven before the start of production. The S7-214 CPUs offer a built-in real-time clock calendar that gives timing resolution down to the second. If that's not enough, the S7-200 has three special memory bits which provide for 30 seconds on/30 seconds off, 0.5 seconds on/0.5 seconds off, and a clock bit which is active every other scan! Only time will tell how many applications will benefit from this feature.



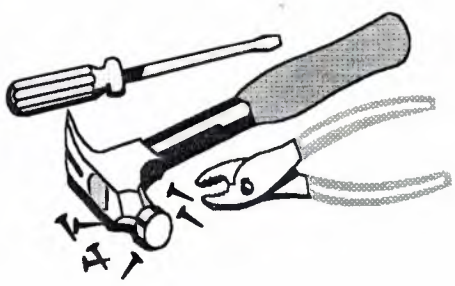
(Integrated Pulse Outputs)

At last, an economical way of connecting stepper motors and DC motors to Micro-PLCs! The S7-214 uses a frequency modulated pulse output to control stepper motors and it can even perform a controlled start-up by means of a pulse ramp! The S7-214 provides integrated pulse output controls that makes stepper motor control as easy as loading a register value. It's also handy for applications like on/off heater control and for generating frequency modulated pulse trains.

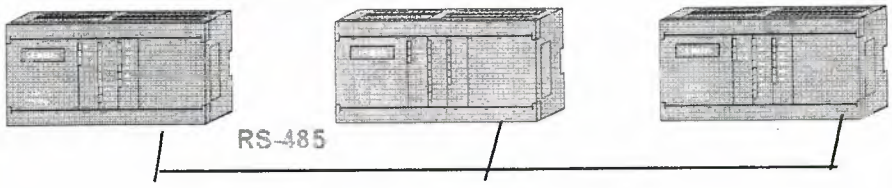


(Powerful Instruction Set)

Over 120 instructions are available to do operations you wouldn't expect from a micro controller. Examples are data table operations such as FILL and FIND, bit shift and rotate, subroutines, and even FOR-NEXT loops! Of course, the S7-200 has all of the standard RLL Boolean instructions.



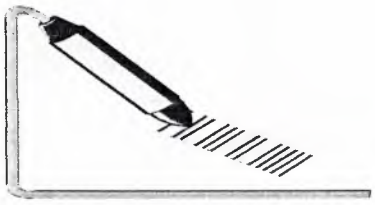
(On-board Communication Port)



The S7-200 PLCs have an integrated interface for serial connections. Up to 31 SIMATIC S7-200 PLCs can be connected for programming and operator interface connections (up to 127 units with repeaters!).

(Exclusive Free Port Mode)

This exclusive feature gives you real control of the serial communications port from your application program. It's now easier than ever to integrate bar code readers, scanners, weigh scales, and other devices with serial communications into your micro control applications.



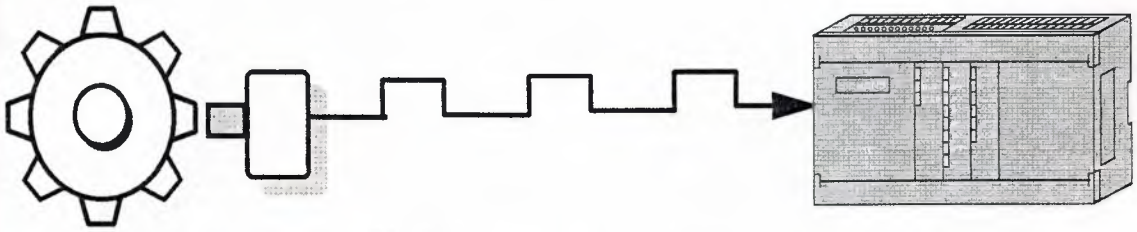
(Password Protection)

You don't want just anyone tinkering with your application programs. The S7-200 offers you three-level password protection against unauthorized access.

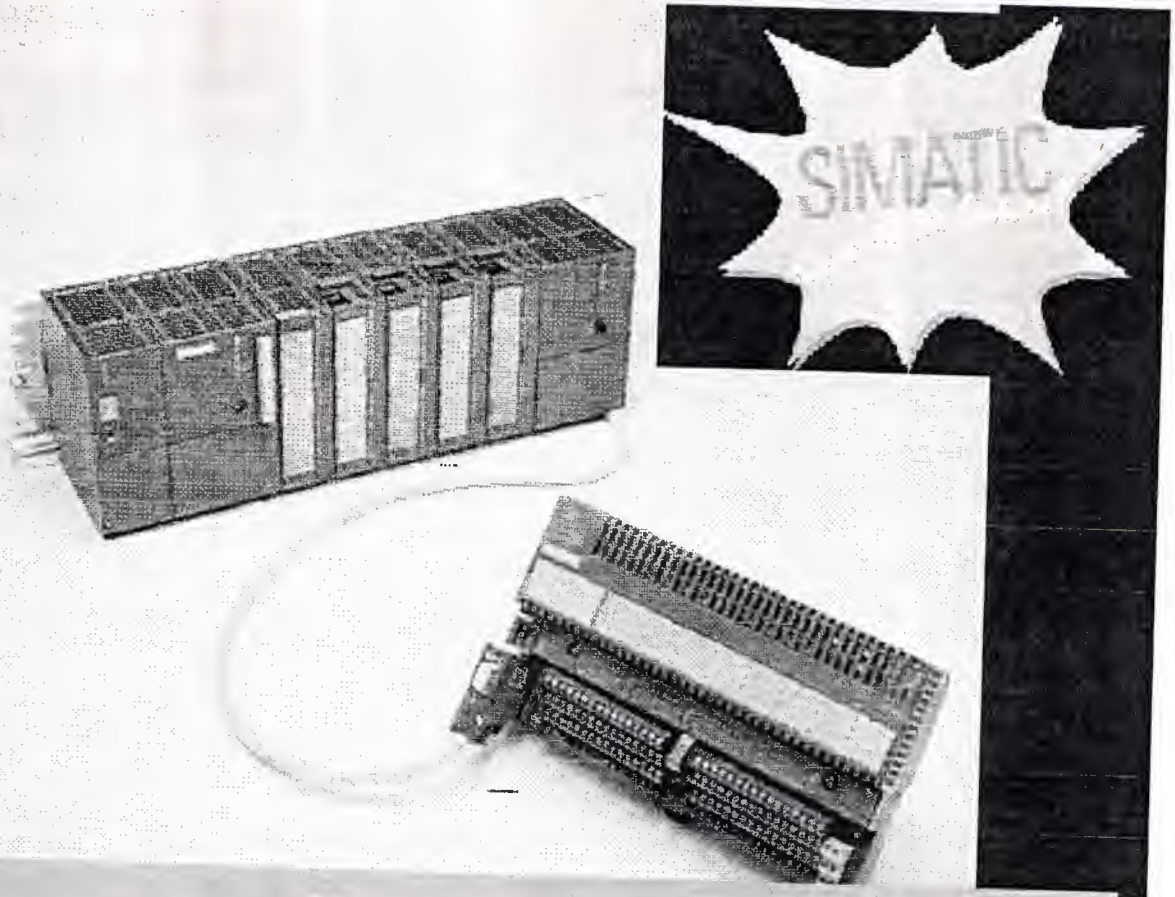
(Maintenance-Free)

This micro has eliminated the need for a backup battery! Program storage is maintained in an on-board EEPROM. There is a super capacitor in the CPU which will provide short-term (up to 190 hours on a CPU 214 and up to 50 hours on the CPU 212!) protection of your scratchpad memory.

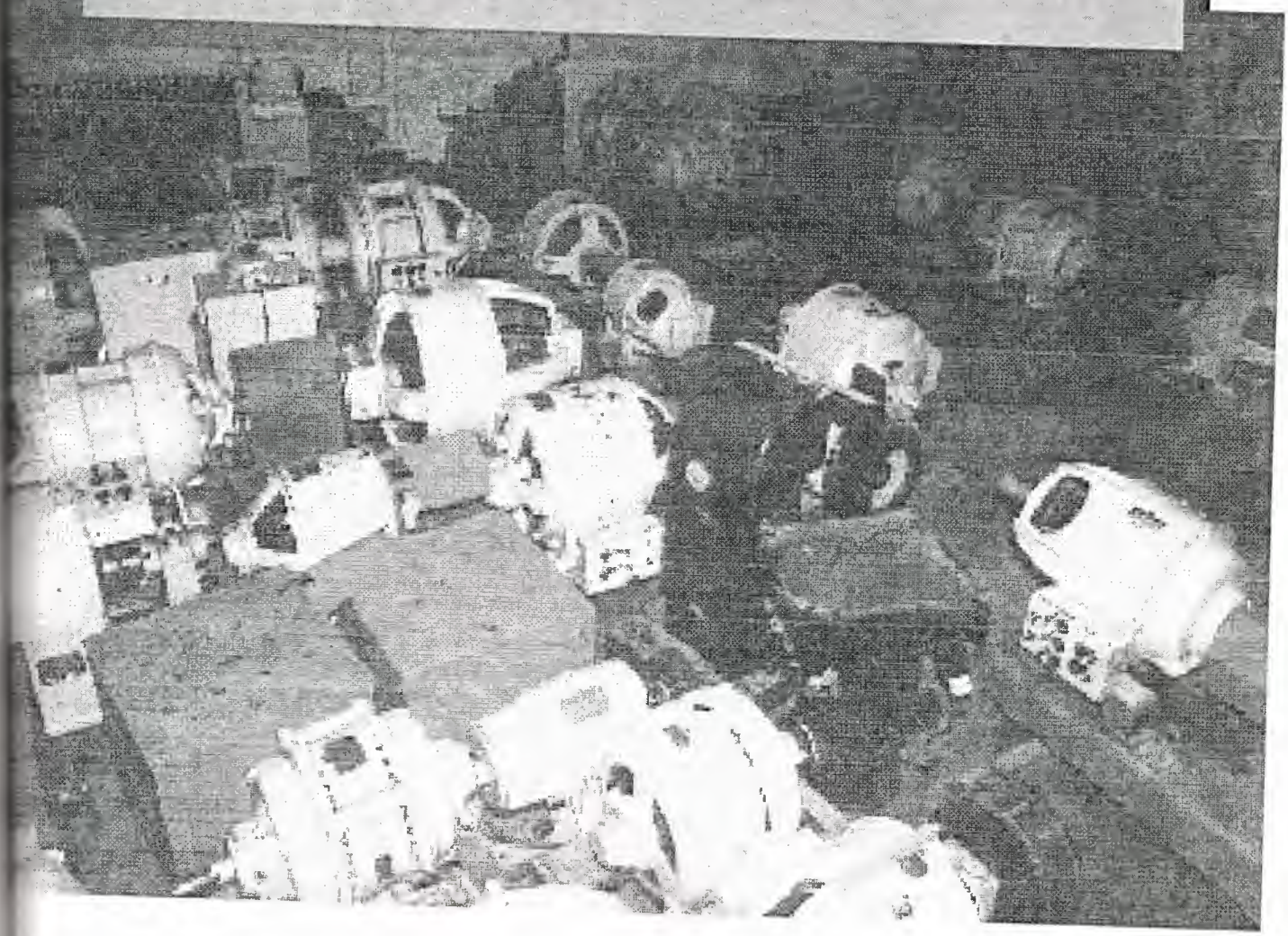
(High Speed Counter - Free)



Even the smallest SIMATIC PLC can count at high speed: forwards or backwards . This makes this PLC suitable for positioning applications. And the S7-214 adds two additional 7 kHz counters offering 12 different operating modes and even as a two-phase counter! This makes the S7-200 just right whether you are counting parts, positioning packages, or even measuring pipeline flow. Plus, since the high-speed counters are build-in to the hardware there's no restrictions against running them at the same time. Now that's High Speed.



Step Motor Control Application With Using PLC



STEP MOTOR CONTROL WITH PLC (SIEMENS S7_CPU212,DC)

INTRODUCTION

Nowadays, in many applications we can meet the Step motors. For example; harddisk tracking control, position controlling, valve controlling. What are the reasons to use step motors in this applications? In this project, we will try to find the answers of this questions..

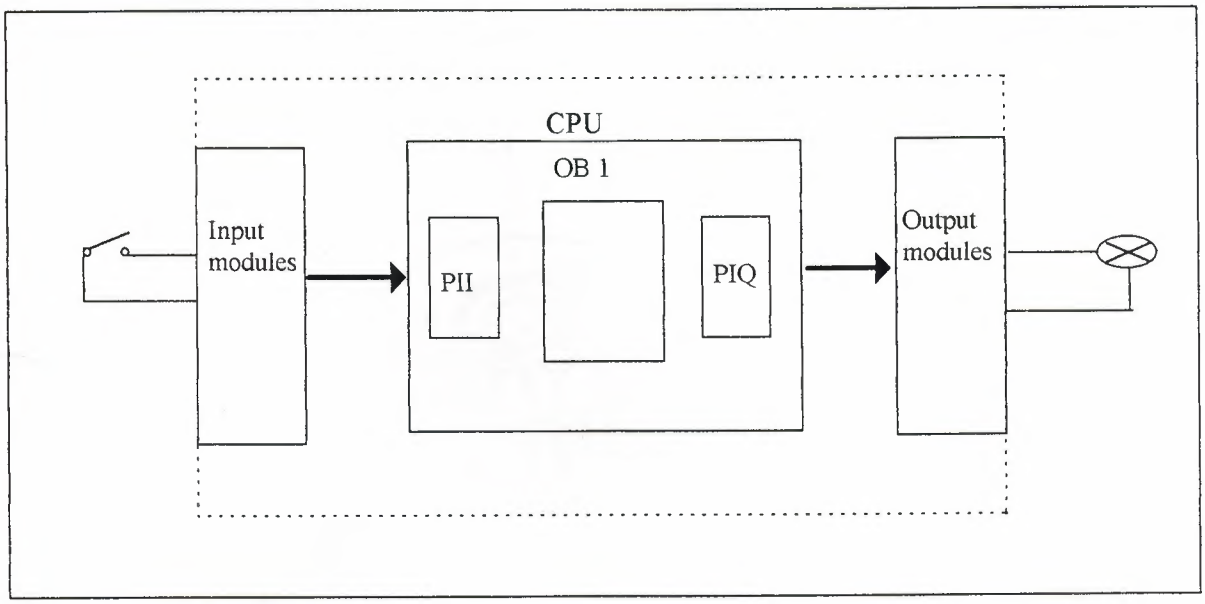
Why do we use PLC?

PLC 's very useably control element to constructing and programming. Electrical control systems with using contactors and timers spends many times and money so that systems are no very good stability. Logical PLC application are more easy with help of computers. Constructing the logical electric circuits needs more qualification. PLC producers says "You can enjoy your coffee in office when programming. You can set up your sophisticated machines system with the fun of playing TV- game.

Working of the PLC

Input Modules :

The sensor signals call up certain signal states on this modules. The CPU scans the signal states. The signal states for the input modules are transferred to the **p**rocess **i**mage **i**nput table (PII) at the beginning of each cycle.



Process Image :

The process image is a special memory area in the programmable logic controller. Differentiation is made between the process image input table (PII) and the **p**rocess **i**mage **o**utput table (PIQ). The signal states of the inputs are transferred to the PII at the beginning of program processing. The PIQ is transferred as a signal state to the outputs at the end of program processing.

CPU :

The CPU processes the sensor signals corresponding to the program commands stored in the user program (OB1). Signals to the actuators are transferred via the process image output table to the output modules.

Cycle :

A cycle includes reading the inputs, processing the user program once, and writing to the outputs. This priority is repeated constantly during cyclical program processing in the CPU.

Output Modules :

The output modules convert the signals coming from the user program to the I/Os (such as potential and analog-digital). The output modules send these signals on to the connected actuators (such as signal lamps and coils).

Step motors

A *stepper motor* converts electrical pulses applied to it into discrete rotor movements called *steps.* A one-degree-per-step motor will require 360 pulses to move through one revolution. Microstep motors, with thousands of steps per revolution, are also available. The rating of the stepper motor is generally given in steps per revolution of the motor. They are generally low speed and low torque, and they provide precise position control of movement. Figure 1 illustrates the basic operation of a dc stepper motor. It consists of an electromagnetically excited stator and a permanent magnet rotor. When the polarity of the exciting winding is suitably reversed, the rotor turns in the chosen direction by exactly one step to a new rest position. The number of steps per revolution is determined by the number of pole pairs on the rotor and stator. The greater the number of poles on both parts, the greater the steps per revolution of the motor.

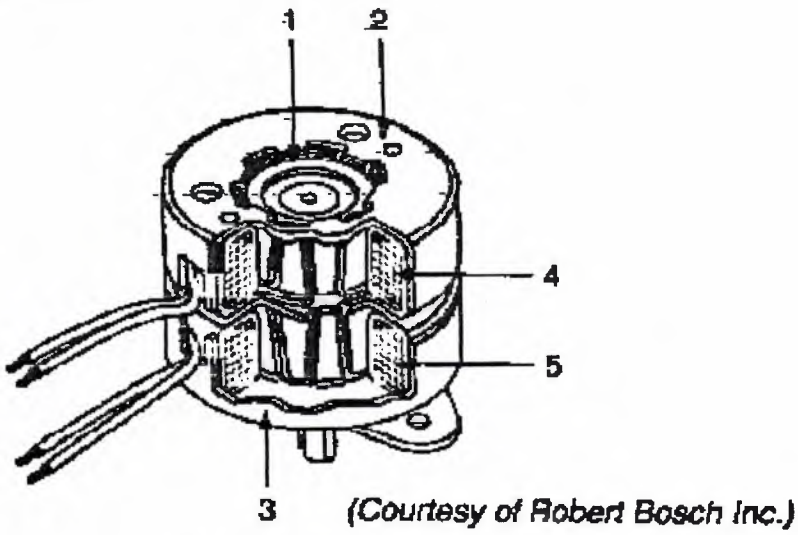
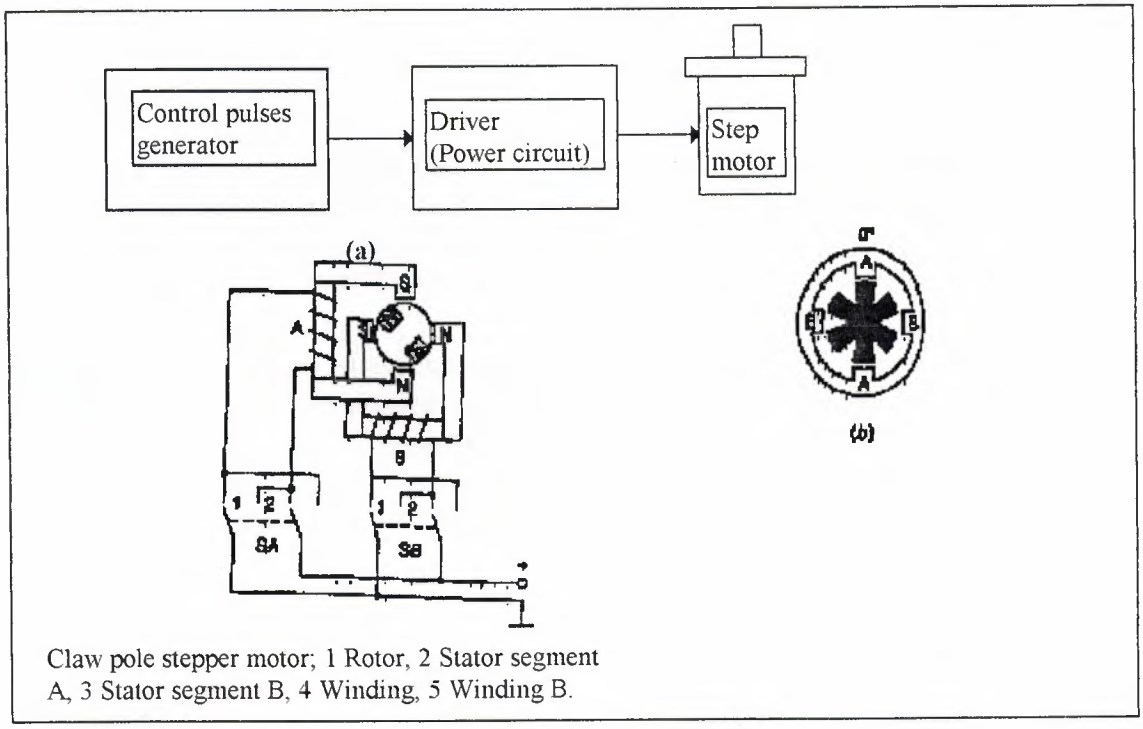


Figure 1; Permanent dc stepper motor.

The operation of a stepper motor greatly depends upon the power supply generates the pulses, which , in turn, are usually initiated by a microcomputer. The computer initiates a series of pulses in order to move the controlled device to whatever position is desired. In this way, the stepper motor provides precise position control of movement. By keeping count of the pulses applied, the computer knows exactly what position the motor is in; therefore, it is unnecessary to use a feedback signal.

A typical stepper motor control system consist of a stepper motor and a drive package that contains control electronics and a power supply (Figure 2-a). The driver is the interface between the computer and the stepper motor. It contains the logic to convert or "translate" digital information into motor shaft rotation. The motor will move one step for each pulse received by the driver. The computer provides the desired number of pulses at a specified or programmed rate which translates into distance and speed.

The number of steps per revolution is determined by the number of pole pairs on the rotor and stator. Once voltage is applied to the windings, the permanent magnet rotor a stepper motor assumes its unloaded holding position. This means that the permanent magnet poles of the rotor are aligned according to the electromagnetic poles of the stator. The maximum torque with which this excited motor can now be loaded without causing a continuous rotation is termed the *stepper motor holding torque.* A torque can also be perceived with nonexcited motor. This is because of the pole induction of the permanent magnet on the stator. This effect (cogging), together with he motor internal friction, produces *detent torque,* which is the torque with which a nonexcited motor can be statically loaded (Figure 2-b).



Claw pole stepper motor; 1 Rotor, 2 Stator segment A, 3 Stator segment B, 4 Winding, 5 Winding B.

Figure 2; Stepper motor control.

There are three types of stepper motors; permanent magnet motors, variable reluctance motors, and hybrid motors. In the permanent magnet type, shown in Figure 3-a, the motor construction result in relatively large step angles. This type is suited to applications in fields such as computer peripherals. *Variable reluctance motors* have no permanent magnet, so they require a different driving arrangement from the other types (Figure3-b). The rotor spins freely without "detent" torque. Torque output for a given frame size is restricted. This type is used in small sizes for applications such as micropositioning tables. As the name implies, *hybrid motors*(Figure 3-c) combine the operating principle of the other two types. The hybrid type is

the most widely used stepper motor in industrial applications. The teeth on each. In between the pole pieces is a permanent magnet which is magnetized along the axis of the rotor, making one end a north pole and the other a south pole. The teeth are offset at the north and south ends as shown in the diagram. The stator consists of a shell having four teeth which run the full length of the rotor. Coils are wound on the stator teeth and are connected together in pairs.

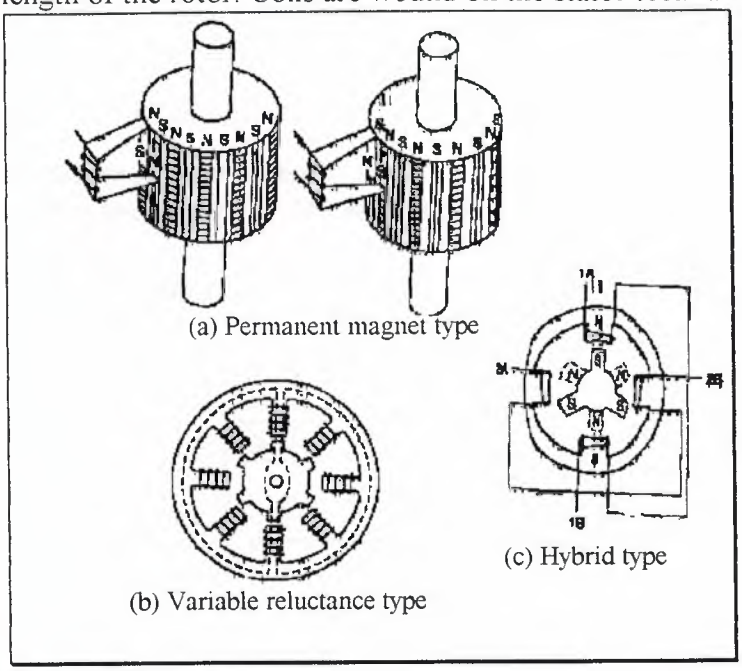


Figure 3; Types of stepper

The increasing trend toward digital control of machines and process functions has generated a demand for stepper motors. Industrial uses include a wide variety of control and positioning applications. A stepper motor can replace devices such as brakes, clutches, and gears with an overall improvement in life expectancy and accuracy.

Advantages of the Step motors over controlling

Step motors have some advantages in controlling. As it is clear in its name "step", the movements of this motor were doing step by step. This property supplies us to use the motor in easiest way in the applications.

It works with electric pulses, when the pulses applies in specified time and regularly, we can see rotation of the motor.

Overview of the Project Application:

Materials:

1- SIEMENS S7-200 CPU212,DC PLC

2- PC-PPI Cable (For connecting PLC with Personal Computer)

3- Software STEP 7 MICRO/WIN

4- STEP MOTOR; Inner resistance 23Ω - 24 V DC - Resolution 24 Step ($\frac{360}{24}=15°$)

5- Driver (Power Circuit)

6- Power Supply 24V DC regulated for PLC and 24V DC for Step Motor (Driver)

Wiring Diagram

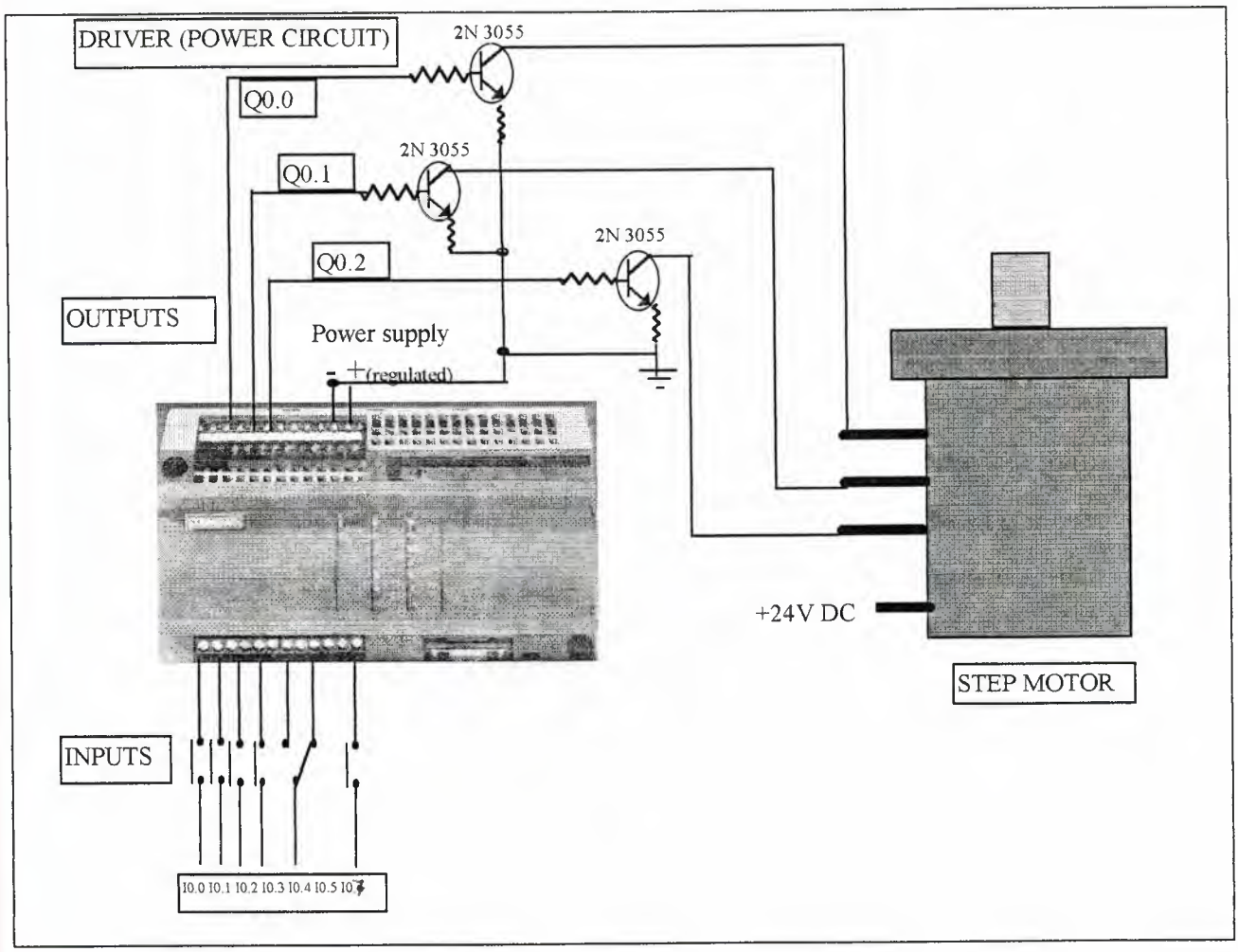


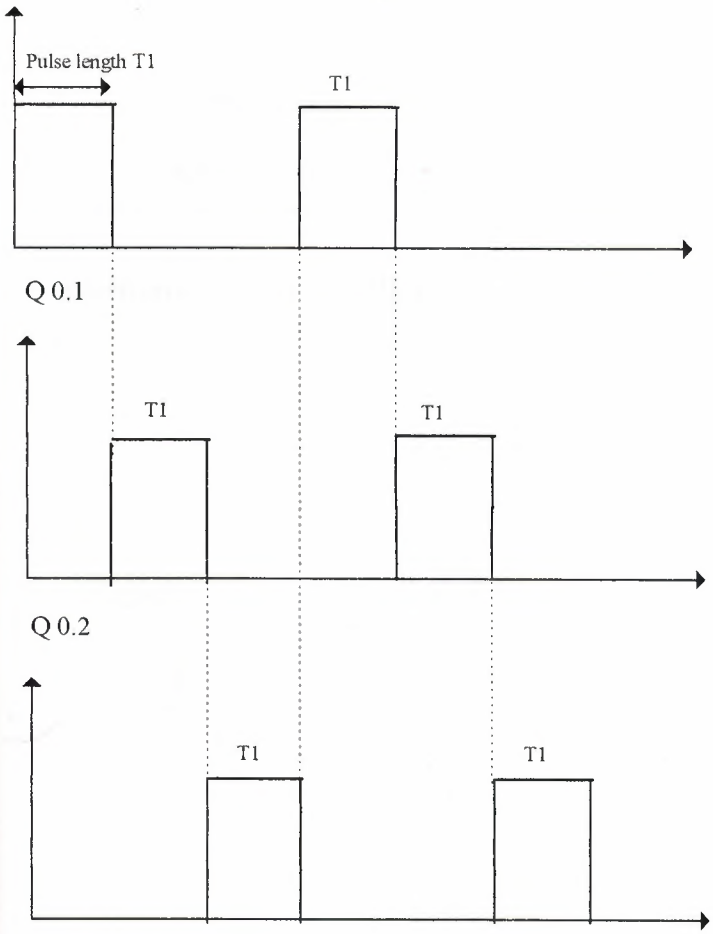
Figure shows wiring of Step motor control with using PLC

Inputs of the PLC

I 0.0	Starts turning of the motor
I 0.1	Stop turning of the motor
I 0.2	Speed Up
I 0.3	Speed Down
I 0.5	Rotation direction selector (Left)
I 0.6	Rotation direction selector (Right)
I 0.7	Get the seated speed value in memory

Outputs of the PLC

Q0.0



Rotation control

When this output cycles repeated after each step we can see the rotation of the step motor. If the cycle pulses reverse direction, we can see rotation of the opposite direction.

So we can store total turning number after starting and also we can actuate another direction when the turning number is equal to stored value. With this techniques we can use to positioning of the step motor with PLC.

Speed Control
When the derived pulses length (T1) is varied, speed of the step motor is change.
If T1 is increase speed is down.
If T1 is decrease speed is up.

Example:
If T1 is set at 10ms, what is the Revolution Per Minute (RPM) value.

Step motor resolution is 24.
Applied outputs (Q0.0_Q0.1_Q0.2) is 3.

In to the 24/3=8 repeated cycle one turn is done. That means each outputs (Q0.0_Q0.1_Q0.2) sends 8 pulses for one turn.
Totally we send 24 pulses, also
24*T1= 24*10ms=240ms passed for 1 revolution.

1 minute = 60second

calculating RPM (revolution per 1 minute)
$60/240*10^{-3}$ sec = **250 RPM at value of T1=10 ms**

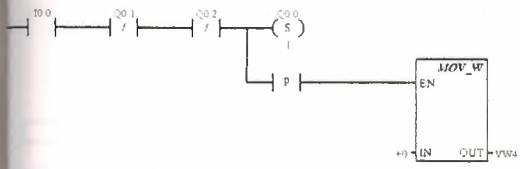
Statement List of the PLC Program

```
NETWORK
LD      I0.0
AN      Q0.1
AN      Q0.2
S       Q0.0, 1
EU
MOVW    +0, VW4

NETWORK
LD      I0.1
R       Q0.0, 1
R       Q0.1, 1
R       Q0.2, 1

NETWORK
LDW=    C0, +8
LPS
EU
+I      +1, VW4
LPP
EU
MOVW    +0, C0

NETWORK
LD      Q0.0
LD      I0.0
EU
CTU     C0, +0
```

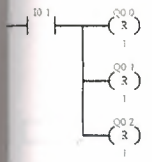
```
NETWORK
LDW>=  +0, C48
MOVW   +20, C48

NETWORK
LD     I0.2
LD     I0.3
LD     I0.7
EU
CTUD   C48, +20

NETWORK
LD     I0.5
CALL   0

NETWORK
LD     I0.6
CALL   1

NETWORK
MEND

NETWORK
SBR    0

NETWORK
LD     Q0.0
TON    T33, C48

NETWORK
LD     T33
S      Q0.2, 1
R      Q0.0, 1
R      T33, 1

NETWORK
LD     Q0.2
TON    T34, C48

NETWORK
LD     T34
R      Q0.2, 1
R      T34, 1
S      Q0.1, 1

NETWORK
LD     Q0.1
TON    T35, C48

NETWORK
LD     T35
R      Q0.1, 1
R      T35, 1
S      Q0.0, 1
```

```
NETWORK
RET

NETWORK
SBR    1

NETWORK
LD     Q0.0
TON    T33, C48

NETWORK
LD     T33
S      Q0.1, 1
R      Q0.0, 1
R      T33, 1

NETWORK
LD     Q0.1
TON    T34, C48

NETWORK
LD     T34
R      Q0.1, 1
R      T34, 1
S      Q0.2, 1

NETWORK
LD     Q0.2
TON    T35, C48

NETWORK
LD     T35
R      Q0.2, 1
R      T35, 1
S      Q0.0, 1

NETWORK
RET
```

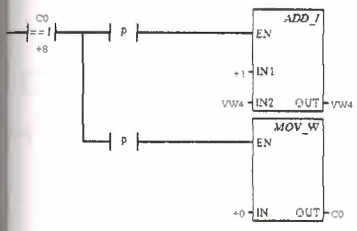
NETWORK 1



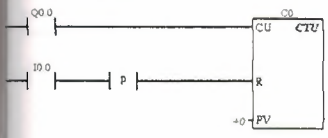
NETWORK 2



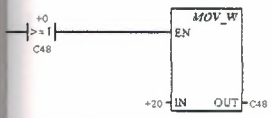
NETWORK 3



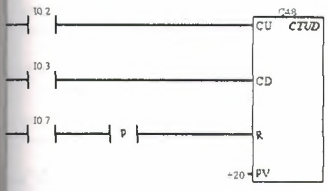
NETWORK 4



NETWORK 5

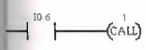


NETWORK 6



NETWORK 7

NETWORK 8

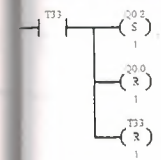


NETWORK 9

NETWORK 10

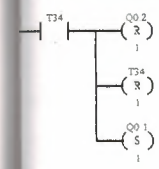
NETWORK 11

NETWORK 12



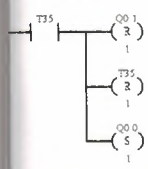
NETWORK 13

NETWORK 14



NETWORK 15

NETWORK 16

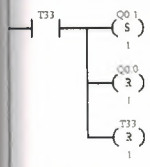


NETWORK 17

NETWORK 18

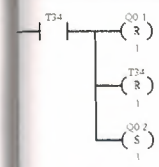
NETWORK 19

NETWORK 20



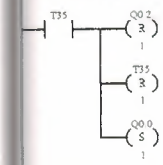
NETWORK 21

NETWORK 22



NETWORK 23

NETWORK 24



NETWORK 25

(RET)

Description of the PLC program

This program does three main job

1) Step motor start & stop and direction of rotation control
2) Step motor speed control
3) Calculating number of turns

1) Step motor start & stop and direction of rotation control

When input (I0.0) is on at first scan the output firs coil is set(Q0.0) when outputs second and third coils are off. It is necessary for no intersection between outputs. After first coil set timer of first coil begins to counting and when the timer value is reached the selected timer value then timer resets first output coil and sets second output(Q0.1) and own counter value for next use. This routine is done when rotation selector input (I0.5) is ON.

If input (I0.5) is OFF and other rotation selector (I0.6) is ON, another routine is done. This is reverse rotation routine. It is similar as first routine except timer sets output reverse direction (Q0.2_Q0.1_Q0.0). If any rotation selector is not ON , there is no rotation on the step motor and step motor takes break position.

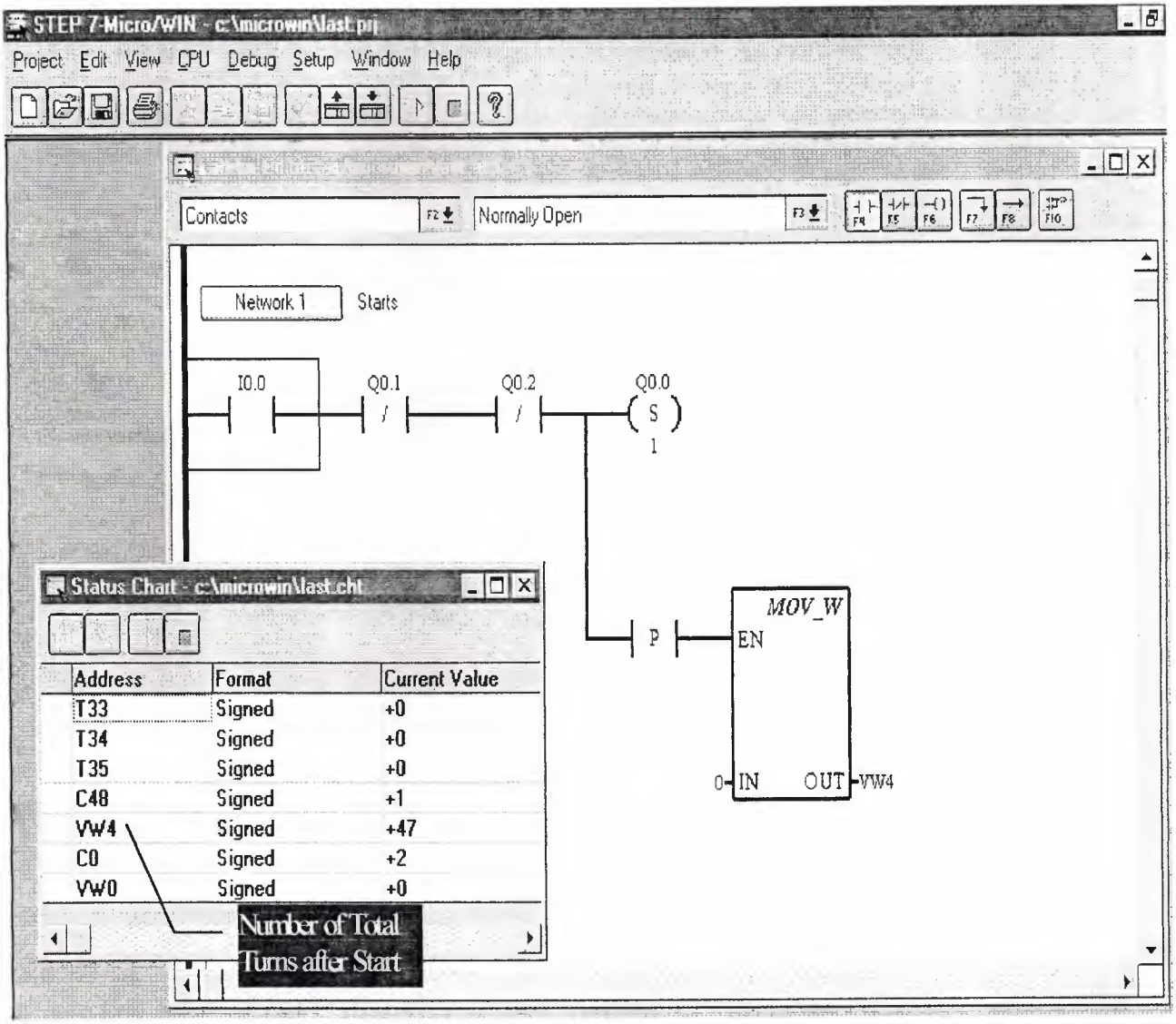


Figure shows ;using status chart total turns value in memory location VW4

2) Step motor speed control

Speed control is made by a Up/Down Counter (C48). This counter has three inputs (I0.2 I0.3 I0.7).

When input I(0.2) is gives pulses to counter, counter increments own value by one.

When input I(0.3) is gives pulses to counter, counter decrements own value by one.

When input I(0.7) is gives pulses to counter, counter resets own value.

All timers selection value is this counter value(C48). This value is decimal number. Our timer are selected between T33-T37. That means timers minimum steps 10ms. All timer set values multiplied by 10, that's results inputs of timers set value is 10 times more then counter(C48) value.

In this procedures shows how can we control speed.

3) Calculating number of turns

The Number of turns is stored in the memory of PLC, it can be shown by PLC programmer program (Micro/Win_Step 7).

As we described before step motor has three inputs and resolution is 24. For each turn we must be send 8 cycle. Cycle means sending outputs with beginning first coil(Q0.0), then second coil(Q0.1), and third coil(Q0.2).

In this program there is a Count Up counter (C0) is selected for sensing 8 cycle. Input of this counter is Q0.0,When output of Q0.0 is sets eight times counter value is eight. Then counter increases selected memory byte (VW4) by one with add instruction. Other input (counter reset) of counter is connect with I0.0. That means counter counts the number of turns at beginning of the starting of step motor.

Conclusion

In industrial control systems, positioning and variable controlling are very popular problems. When solving this problem we will phase some problems. Which system is gives best accuracy and reliability.

In position controlling step motors gives us many advantages. In step motor applications there is no need feedback to control. So, it eliminates most of the problems. For positioning if our system is related with computer, that gives power in complex calculations.

In our application PLC's is used as a bridge to computer. When inputs are given by a computer many complex movement can be done by PLC. PLC generates needed output signal form for step motor.

All this thinks shows us, there is a need for engineers and technicians to be familiar with PLCs and to be able to programme, service and repair PLC based systems. Although the appearance and size of the PLC may vary from model to model, between manufacturers and in the style of programming in general, they all have the same requirement; the need to be connected to a physical system and be programmed correctly if they are to provide the quality of monitoring and control required.