Modicon Compact 984 Ladder Logic Manual

043503387

Modicon Compact 984 Ladder Logic Manual

043503387

April, 1993

MODICON, Inc., Industrial Automation Systems
One High Street
North Andover, Massachusetts 01845

Preface

The data and illustrations in this book are not binding. We reserve the right to modify our products in line with our policy of continuous product improvement. Information in this document is subject to change without notice and should not be construed as a commitment by Modicon, Industrial Automation Systems. Modicon, Inc. assumes no responsibility for any errors that may appear in this document.

Contents

Chapter 1
Compact Controllers

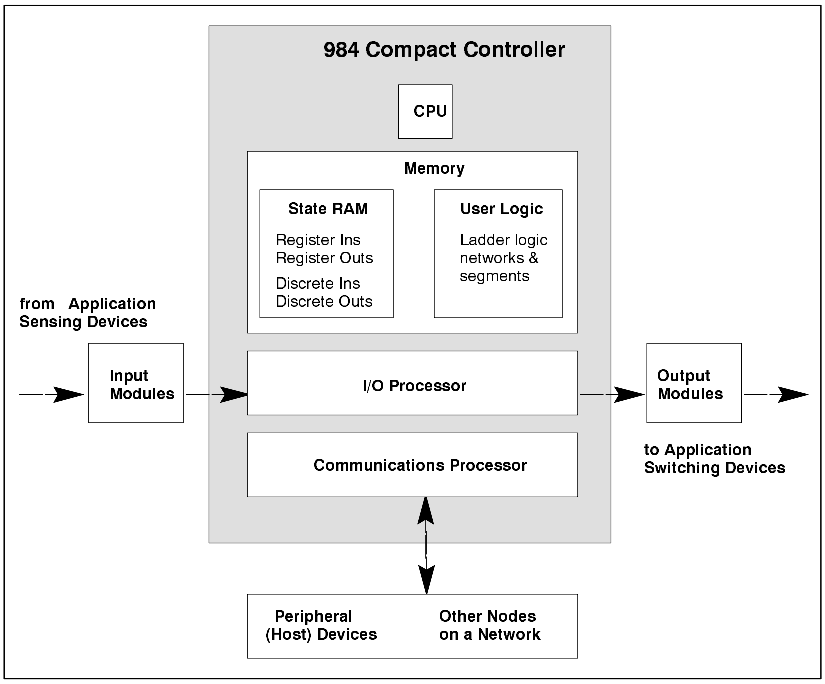
The Compact Controllers

Modicon's Compact Programmable Logic Controllers bring the high performance, application flexibility, and programming compatibility of the 984 family to the small controller market. Like other controllers in the 984 family, the Compacts implement a common instruction set for developing user logic, along with Modbus and optional Modbus Plus communication capabilities.

Common 984 Architecture

The Compact Controllers share the following processing architecture with all other controllers in the 984 family:

- ❐ A *memory* section that stores user logic, I/O tables, and system overhead in battery-backed CMOS RAM and holds the system's Executive firmware in nonvolatile EPROM
- ❐ A *CPU* section that solves the user logic program based on the current input values in state RAM, then updates the output values in state RAM
- ❐ An *I/O processing* section that directs the flow of signals from input modules to state RAM and provides a path over which output signals from the CPU's logic solve are sent to the output modules
- ❐ A *communications* section that provides one or more port interfaces. These interfaces allow the controller to communicate with programming panels, host computers, hand-held diagnostic tools and other master devices, as well as with additional controllers and other nodes on a Modbus (or Modbus Plus) network



This architectural consistency allows the Compact Controllers to achieve machine compatibility with the other controllers in the family. This means that user logic created on a midrange or high performance controller—such as a 984-685 or a 984B Controller—can be relocated to a Compact if the specifications of the Compact are not exceeded. Also, user logic you generate for the small controller is upwardly compatible to a larger 984. It also means that a Compact can be easily integrated into a multi-controller network.

Ladder Logic Programming

All 984 Controllers can be programmed via ladder logic, a powerful and highly graphical language for control operations. A database of standard ladder logic instructions is stored in the system Executive.

A120 I/O Support

The Compact Controllers work with Modicon's low-cost series of A120 I/O modules. A120 modules are available in various densities of discrete I/O points and various numbers of analog I/O channels. For detailed descriptions of available A120 modules, see the ***A120 Series I/O Modules User Guide*** (GM-A984-IOS).

Each module uses a standardized pair of screw-type terminal blocks that facilitate easy access and easy field wiring. Because the terminal blocks are standardized and removeable, they allow you to make module changes without disturbing connections.

A tool (AS-0TBP-000) to facilitate the removal of terminal blocks is shipped with the Compact.

Power Supplies

The Compact Controllers have a built-in 5 VDC power supply that provides up to 2.5 A across the I/O bus to all I/O modules in the system.

An external 24 VDC source (-15% to +20% range, 1 A minimum) must be connected to the Compact to power the system. If you are operating in an all-AC environment, you can use the AS-P120-000 Power Supply to convert AC source power to 24 VDC.

Some A120 I/O modules require an external 24 VDC supply, and others require an external 115 or 230 VAC supply.

Auxiliary Memory Upload-Download Capabilities

All Compact Controllers contain an auxiliary memory socket for a credit card-sized EEPROM card. You can write the current system configuration and user logic program to an EEPROM card while the controller is stopped and read the data back to the controller from the EEPROM card as part of the power-up sequence. This utility allows you to record, store, and reload applications using an easily accessible medium.

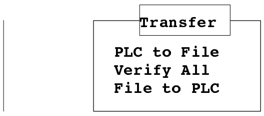
Relocating Logic from One 984 to Another

The only constraints on logic relocation are that the program in the source controller must generate logic that implements only instructions acceptable to the target controller, and that the size of the source logic program must not exceed the memory limits of the target controller.

Relocating 984 Logic

Ladder logic from one 984 controller can be easily downloaded to another 984 using your panel software—e.g., MODSOFT Lite. First you upload the source program to your programming panel by selecting `PLC` on the main menu, then selecting `Transfer` from the top level menu line.

The select `PLC to File` command from the pulldown menu saves the contents of the target controller in a file. The `File to PLC` command from the pulldown loads the contents of the file to a target controller.



Compact CPU and User Memory Choices

Several Compact models are currently available with different user memory sizes and various comm port offerings:

- The 984-120 CPU with 1.5K words of user memory and one Modbus communication port
- The 984-130 CPU with 4K words of user memory and one Modbus communication port
- The 984-131 CPU with 4K words of user memory and two Modbus communication ports
- The 984-141 CPU with 8K words of user memory and two Modbus communication ports
- The 984-145 CPU with 8K words of user memory, one Modbus port, and one Modbus Plus network interface

User Memory

User memory is the amount of memory space (one word comprises 16 bits) provided for your user logic program and for the system overhead. Approximately 1K of user memory is used for system overhead, and the remaining words are available for application logic.

An additional 2048 (16-bit) words are provided for *state RAM*—up to 1920 words can be used for register/analog inputs, outputs, and internal data storage while the remainder is dedicated to discrete I/O. Up to 2048 bits can be used for discrete inputs, outputs, and internal coils.

All Compact models provide up to 256 points of I/O under local control.

All Compact models solve logic at the rate of 4.25 ... 6 ms/K nodes of standard ladder logic.

Reference Numbering

For ladder logic programming, the Compact Controllers use a reference numbering system to handle input/output information and internal logic. Each reference number has a leading digit that identifies the I/O data type; the leading digit is followed by a string of four digits that defines that I/O point's unique location in user data memory.

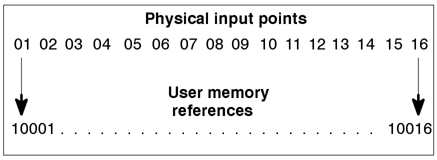
There are four reference types:

I/O Reference Numbering System

Reference Number	Description
0*xxxx*	A discrete output (or coil). A 0*x* reference can be used to drive real output data through an output unit in the control system or it can be used to set one or more coils in state RAM. A specific 0*x* reference may be used only once as a coil in a logic program, but that coil status may be used multiple times to drive contacts in the program
1*xxxx*	A discrete input. The ON/OFF status of a 1*x* reference is controlled by field data sent to the CPU from an input unit. It can be used to drive contacts in a logic program
3*xxxx*	An input register. A 3*x* register holds information represented by A 16-bit number and received from an external source—e.g., a thumbwheel, an analog signal, data from a high speed counter. A 3*x* register can also hold 16 consecutive discrete input signals, which may be entered into the register in binary or binary coded decimal (BCD) format.
4*xxxx*	An output or holding register. A 4*x* register may be used to store numerical data (binary or decimal) in state RAM or to send the data from the CPU to an output unit in the control system.

Note: The *x* following the leading character in each reference type represents a four-digit address location in user data memory—e.g., the reference 40201 indicates that the reference is a 16-bit output or holding register located at address 201 in state RAM.

Each word in user memory is 16 bits long. The (ON/OFF) state of each discrete I/O point is represented by the 1 or 0 value assigned to an individual bit in a word (16 0*x* or 1*x* references per word).



Discrete outputs are traffic copped to 0*x* registers in a similar way

In the case of analog I/O, each input channel and each output channel is traffic copped to a full word in user data memory (3*x* registers for inputs and 4*x* registers for outputs).

Logic Elements and Instructions

Standard Ladder Logic Programming Elements

Symbol	Meaning
-| |-	A normally open contact
-|\\\|-	A normally closed contact
-|↑|-	A positive transitional contact
-|↓|-	A negative transitional contact
-( )-	A normal coil
-(L)-	A latched coil

Compact Instruction Set

Counter and Timer Instructions (Two-Node Functions)

Instruction	Meaning
UCTR	Counts up from 0 to a preset value
DCTR	Counts down from a preset value to 0
T1.0	Timer that measures in seconds
T0.1	Timer that measures in tenths of a second
T.01	Timer that measures in hundredths of a second

Calculation Instructions (Three-Node Functions)

Instruction	Meaning
ADD	Adds top node value to middle node value
SUB	Subtracts middle node value from top node value
MUL	Multiplies top node value by middle node value
DIV	Divides top node value by middle node value

DX Move Instructions (Three-Node Functions)

Instruction	Meaning
R→T	Moves register values to a table
T→R	Moves table values to a register
T→T	Moves a specified set of values from one table to another table
BLKM	Moves a specified block of data
TBLK	Moves a block of data from a table to another specified block area
BLKT	Moves a block of registers to specified locations in a table
FIN	First-in operation to a queue
FOUT	First-out operation from a queue
SRCH	Performs a table search for a value
STAT	Displays system status from locations in the controller's memory

Compact Instruction Set (concluded)

DX Matrix Instructions (Three-Node Functions)

Instruction	Meaning
AND	Logically ANDs two matrices
OR	Does logical inclusive OR of two matrices
XOR	Does logical exclusive OR of two matrices
COMP	Performs the logical complement of values in a matrix
CMPR	Logically compares the values in two matrices
MBIT	Logical bit modify
SENS	Logical bit sense
BROT	Logical bit rotate
CKSM	Performs one of four possible checksum operations (*This function is not available on the 984-145 Controller.*)

Skip-Node Instruction (One-Node Function)

Instruction	Meaning
SKP	Skips a specified number of networks in ladder logic

Ladder Logic Subroutine Instructions (One- and Two-Node Functions)

Instruction	Meaning
JSR	Jumps from scheduled logic scan to a ladder logic subroutine
LAB	Labels the entry point of a ladder logic subroutine
RET	Returns from the subroutine to scheduled logic

PID Instruction (Three-Node Function)

Instruction	Meaning
PID2	Performs a specified proportional-integral-derivative function

Enhanced Math (Three-Node Function)

Instruction	Meaning
EMTH	Performs 38 math operations, including floating point math operations and extra integer math operations such as square root

Modbus Plus Networking Instruction (Three-Node Function)

Instruction	Meaning
MSTR	Specifies a function from a menu of networking operations (*This function is available only on the 984-145 Controller, which supports Modbus Plus communications.*)

The following chapters of this book provide more details on the usage of these standard ladder logic elements and instructions.

Loadable Instructions

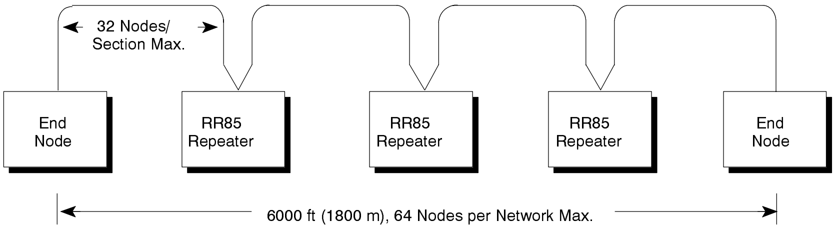
The Compact Controllers also support various loadable instructions, including:

- EARS, a tool for developing an early alarm reporting system (see ***Event Alarm Reporting System User Guide***, GM-EARS-001)
- EUCA, an engineering unit conversion algorithm (see ***EUCA Loadable Function Block User Guide***, GM-EUCA-001)
- FN*xx*, user-designed loadable instructions created with our custom loadable tool (see ***Custom Loadable Support Software Programming Manual***, GM-CLSS-001)
- DRUM and ICMP, which can be used to create control logic for tenor drum sequencing applications (see ***Drum Sequencer Demo S/W User Guide***, GI-0984-SAS)
- HLTH, which creates history and status matrices that can be programmed to alert a user to changes in a PLC system (see ***Health Loadable User Guide***, GM-HLTH-001)

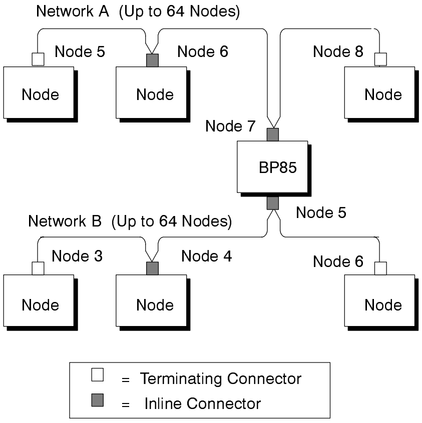
Chapter 2
Modbus Plus

Modbus Plus Capability for the Compact-984 Controller

Modbus Plus is a local area network designed for industrial control applications. It enables the 984-145 Controller to become a node on the network and to communicate with other 984 controllers, host computers, and special bridge and multiplexer devices. A network may comprise one or more communication sections—one section may support up to 32 *nodes*. Up to 64 nodes may exist on a network.



Multiple Modbus Plus networks may be interconnected using a BP85 Bridge Plus device.



Each node within a network must have a unique address number in the range 1 ... 64. The node address of a 984 chassis mount controller is specified using a set of DIP switches provided on the top front of the 984-145 module.

Modbus Plus uses a proprietary protocol that delivers high performance intercommunication capabilities at a data transfer rate of 1 Mbit/s. The network medium is twisted-pair shielded cable, laid out in a sequential multidrop path directly between successive nodes. Taps and splitters are not used.

Modbus Plus Token Rotation

Each node on a Modbus Plus network functions as a peer on a logical ring, gaining access to the network upon receipt of a token frame. The token is a bit grouping that is passed in a rotating address sequence from one node to the next. While an individual node holds the token, it may initiate data read/write and statistical transactions with other nodes; when the node passes its token, it may write to a global database that is maintained by all nodes on the network. Use of this global database allows rapid updating of alarms, setpoints, and other data.

How the 984-145 Initiates Modbus Plus Transactions

A 984-145 (or any programmable controller with Modbus Plus capability) may initiate network communication using a ladder logic function called MSTR. MSTR allows you to specify the type of communications transaction you want to carry out and to define the routing path over which you wish the transaction to take place.

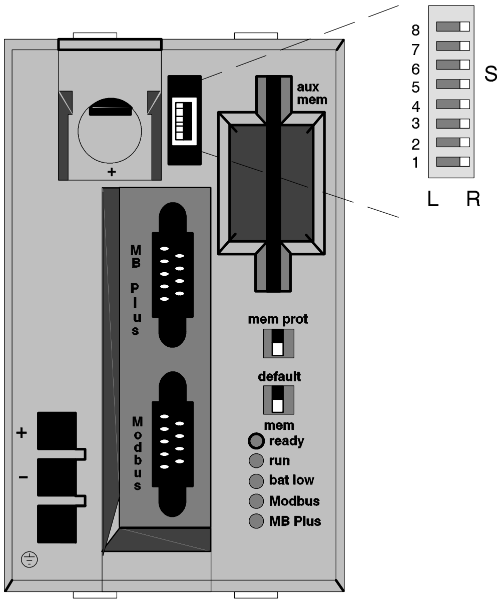
The MSTR block is part of the standard 984-145 instruction set, contained in the system executive.

> **Note** In order to thoroughly understand Modbus Plus theory of operations, to be able to plan the layout of the total network, and to meet all the requirements of the network cable installation, refer to ***Modicon Modbus Plus Network Planning and Installation Guide*** (GM-MBPL-001).

For a full description of the MSTR function block, see Chapter 8 of this book.

Modbus Plus Node Addressing

Each node on a Modbus Plus network must be assigned a unique address in the range 1 ... 64 using switches 1 ... 6 on the addressing DIP switch on the top front of the 984-145 bezel.



Switches Shown in OFF Position

R = OFF
L = ON

Location of the Modbus Plus Addressing Switches

Modbus Plus Node Address Settings for the 984-145 Controller

	Switch Position									Switch Position						
Address	1	2	3	4	5	6	7	8	Address	1	2	3	4	5	6	7	8
1	R	R	R	R	R	R	–	–	33	R	R	R	R	R	L	–	–
2	L	R	R	R	R	R	–	–	34	L	R	R	R	R	L	–	–
3	R	L	R	R	R	R	–	–	35	R	L	R	R	R	L	–	–
4	L	L	R	R	R	R	–	–	36	L	L	R	R	R	L	–	–
5	R	R	L	R	R	R	–	–	37	R	R	L	R	R	L	–	–
6	L	R	L	R	R	R	–	–	38	L	R	L	R	R	L	–	–
7	R	L	L	R	R	R	–	–	39	R	L	L	R	R	L	–	–
8	L	L	L	R	R	R	–	–	40	L	L	L	R	R	L	–	–
9	R	R	R	L	R	R	–	–	41	R	R	R	L	R	L	–	–
10	L	R	R	L	R	R	–	–	42	L	R	R	L	R	L	–	–
11	R	L	R	L	R	R	–	–	43	R	L	R	L	R	L	–	–
12	L	L	R	L	R	R	–	–	44	L	L	R	L	R	L	–	–
13	R	R	L	L	R	R	–	–	45	R	R	L	L	R	L	–	–
14	L	R	L	L	R	R	–	–	46	L	R	L	L	R	L	–	–
15	R	L	L	L	R	R	–	–	47	R	L	L	L	R	L	–	–
16	L	L	L	L	R	R	–	–	48	L	L	L	L	R	L	–	–
17	R	R	R	R	L	R	–	–	49	R	R	R	R	L	L	–	–
18	L	R	R	R	L	R	–	–	50	L	R	R	R	L	L	–	–
19	R	L	R	R	L	R	–	–	51	R	L	R	R	L	L	–	–
20	L	L	R	R	L	R	–	–	52	L	L	R	R	L	L	–	–
21	R	R	L	R	L	R	–	–	53	R	R	L	R	L	L	–	–
22	L	R	L	R	L	R	–	–	54	L	R	L	R	L	L	–	–
23	R	L	L	R	L	R	–	–	55	R	L	L	R	L	L	–	–
24	L	L	L	R	L	R	–	–	56	L	L	L	R	L	L	–	–
25	R	R	R	L	L	R	–	–	57	R	R	R	L	L	L	–	–
26	L	R	R	L	L	R	–	–	58	L	R	R	L	L	L	–	–
27	R	L	R	L	L	R	–	–	59	R	L	R	L	L	L	–	–
28	L	L	R	L	L	R	–	–	60	L	L	R	L	L	L	–	–
29	R	R	L	L	L	R	–	–	61	R	R	L	L	L	L	–	–
30	L	R	L	L	L	R	–	–	62	L	R	L	L	L	L	–	–
31	R	L	L	L	L	R	–	–	63	R	L	L	L	L	L	–	–
32	L	L	L	L	L	R	–	–	64	L	L	L	L	L	L	–	–

Bridge Mode Between Modbus and Modbus Plus

The standard Modbus port on the 984-145 Controller can be used in either of two ways: as a slave port to a Modbus master device or as a *bridge* between a Modbus master device and the Modbus Plus network nodes. Make this selection by setting the comm parameter slide switch (the bottom slide switch) on the 984-145 Controller.

The Standard Modbus Setting

If you want the standard Modbus mode, set the switch to the **mem** (↓) position. You must set the desired Modbus port parameters in software—using the configurator editor.

The Modbus Plus Bridge Mode Setting

Bridge mode allows you to access nodes on a Modbus Plus network from a Modbus master device (connected to the standard Modbus port). To set the Modbus Plus bridge mode, set the slide switch to **default** position—the controller's bridge mode is automatically enabled.

The Modbus port parameters are set to 9600 baud, RTU mode (8 data bits and 1 stop bit), and EVEN parity, the same **default** conditions as the -120 and -130 Controllers. *Unique to the 984-145, however, is the default port address*. Instead of defaulting to Modbus port address 1, it defaults to the Modbus Plus port address set by the DIP switch at the top of the 984-145 Controller.

When a Modbus master device is connected to the Modbus port while the 984-145 is in bridge mode, the master device can be attached to the local controller or to any other node on Modbus Plus. When you attach to the local controller, messages from the Modbus master are sent directly to the local 984-145 without being routed over a Modbus Plus communication path. When you attach to any other node on the network, the message is routed through the Modbus Plus port to the destination device.

When you are connecting a Modbus master device to a node on Modbus Plus, always use the desired node's Modbus Plus address. If you are attaching to the local 984-145 in bridge mode, the master automatically attaches to the Modbus Plus node address set by the DIP switches on the local controller; however, if you want to attach to any other Modbus Plus node, the Modbus master device must specify that node by Modbus Plus address.

Caution If you are accustomed to using Modbus master devices (such as programming panels) with Modicon programmable controllers in unnetworked environments, you may be used to attaching to the local controller by addressing it as device #1—the default device address in the configurator editor. Be aware that in a Modbus Plus network environment you must know the Modbus Plus address of the controller (or any other nodal device) with which you want to communicate and you must specify that address correctly in the attach procedure.

If you want to attach to a node on Modbus Plus but do not know its network address, get this information from your network supervisor before proceeding.

Note When a Modbus port is used in bridge mode, it must be connected to a single Modbus master device—the bridge cannot be used as a connection for a Modbus device network.

Addressing Ranges on Modbus Plus

A single Modbus Plus network can have up to 64 addressable nodes, each with a unique address in the range 1 ... 64). The Modbus master device connected to the Modbus port can attach to any one of these nodes using *direct attach address* routing, simply by specifying the correct address in the range 1 ... 64.

Multiple networks can be joined via BP85 Bridge Plus devices, and nodes across multiple networks can be addressed. In cases such as this, you will require an addressing capability outside the 1 ... 64 range. Two address routing strategies—*explicit* and *implicit attach address* routing—are available in Modbus Plus. These routing techniques are described in the following sections.

Modbus Plus Address Routing Schemes

Modbus devices use addresses of one byte in the range 1 ... 255. Modbus Plus devices are addressed in the range 1 ... 64, with five consecutive routing bytes contained in each message. When a Modbus message is received at the Modbus port on the 984-145 Controller, the single-byte address contained in the message is converted into a five-byte routing path for Modbus Plus. The five bytes of routing are imbedded in a Modbus Plus message frame as it is sent from the originating node.

Destination Device Requirements

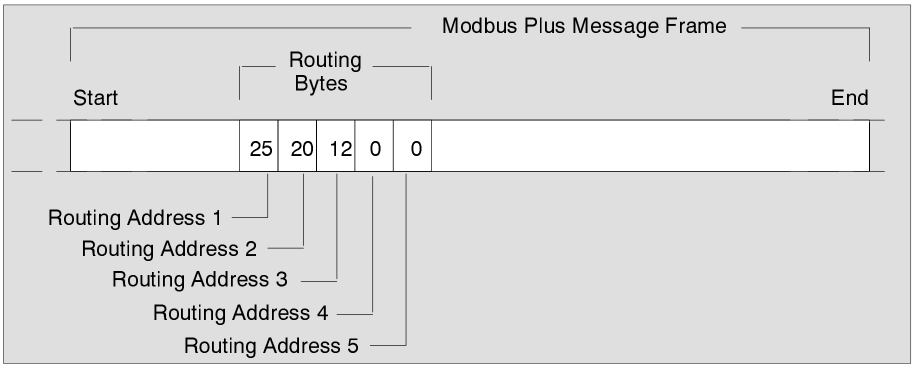
The structure of the Modbus Plus routing address is determined by the type of device at the destination node:

- If you are initiating a transaction with *another 984 controller*, the last (*rightmost*) nonzero byte in the routing scheme is the destination node address
- If you are initiating a transaction with a *network adapter in a non-controller node*—e.g., an SA85—the next to the last nonzero byte is the destination node address, and the last nonzero byte is the task # (range: 1 ... 8)
- If you are initiating a transaction with a *single slave on a Bridge MUX port*, the next to the last nonzero byte is the Bridge MUX node address, and the last nonzero byte is the desired MUX port # (range: 1 ... 4)
- If you are initiating a transaction with a *slave device on a Modbus network connected to a Bridge MUX*, the second from the last nonzero byte is the node address of the MUX, the next to the last nonzero byte is the desired MUX port # (range: 1 ... 4), and the last nonzero byte is the desired Modbus slave address (range: 1 ... 247)

Any leading nonzero bytes ahead of the address bytes described above are Bridge Plus node addresses.

Assume, for example, that your routing path is to a controller two networks removed from the originating 984. The message is routed first to a BP85 Bridge Plus at node address 25. The bridge forwards the message to node 20, a BP85 Bridge Plus device on the second network. Node 20 forwards the message to the destination controller node address 12 on the third network. The zero-content bytes in the fourth and fifth routing bytes specify that no further routing is required beyond the third byte:

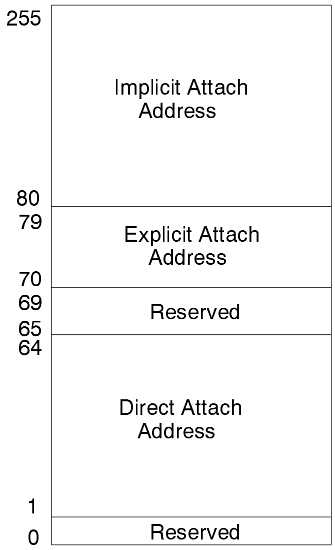
☞ **Note** The routing address scheme must be developed as part of an overall network planning process—for details, see ***Modbus Plus Network Planning and Installation Guide*** (GM-MBPL-001).



A Message Frame Routing Path

Direct, Explicit, and Implicit Attaches

The manner in which Modbus Plus converts a Modbus message using bridge mode is determined by the range of the Modbus address (1 ... 255):



Modbus-to-Modbus Plus Address Conversion

If the address range in the Modbus message is between 1 ... 64, the message is routed to the corresponding Modbus Plus node address on the local network. This routing procedure is called *direct attach address*. Direct attach address routing implies that a nonzero value exists in only routing address 1 in the Modbus Plus message frame; it does not allow you to send the incoming Modbus message beyond the local network.

If the address range in the Modbus message is between 70 ... 79, the controller initiates an *explicit attach address* procedure which compares the Modbus address to an address table stored in the controller. Up to 10 addresses in the range 70 ... 79 become pointers to the table, which contains up to 10 stored routing paths for Modbus Plus. (This table starts with the register that immediately follows the register selected for the free-running timer in the controller.)

Each routing path is five bytes in length. The routing path pointed to by each address is applied to the corresponding message.

Explicit attach address routing implies that nonzero values may exist in any or all routing addresses in the Modbus Plus message frame; it allows you to send incoming Modbus messages through as many as four BP85 Bridge Plus devices across as many as five Modbus Plus networks.

If the address range in the Modbus message is between 80 ... 255, the controller initiates an *implicit attach address* procedure which divides the address by 10 and uses the quotient and remainder as the first and second bytes, respectively, in a routing path. Implicit attach address routing implies that there may be nonzero values in routing addresses 1 and 2 in the Modbus Plus message frame; it allows you to send incoming Modbus messages through one BP85 Bridge Plus device across up to two Modbus Plus networks.

Modbus Plus Communication Paths

With multiple devices processing messages asynchronously on a Modbus Plus network, it becomes possible for an individual device to have several concurrent transactions in process. The 984-145 Controller opens a communication path when a transaction begins, keeps it open during the transaction, and closes it when the transaction terminates. When the path is closed, it becomes available for another transaction.

Four Types of Communication Paths

A 984-145 Controller maintains four types of communication paths

- *Data master paths*—For *read* and *write data* or *get* and *clear remote statistics* operations originated by a MSTR block in the 984-145 Controller going to a destination device on the network. A 984-145 supports up to *five* data master paths—paths DM01 ... DM04 for processing up to four concurrent MSTR blocks, and path DM05 that may be used for data master transactions via the Modbus port. Design your application to use a maximum of four MSTR data master paths at any one time.

- *Data slave paths*—For data reads and writes received over the network. The 984-145 supports up to *four* data slave (DS) paths handling up to four concurrent network transactions.

- *Program master paths*—For sending programming commands from the local controller to the Modbus Plus network. Program master paths can handle all Modbus commands—i.e., function codes. When a Modbus master is connected to the Modbus port on the 984-145, it may be used for either programming or monitoring functions. A 984-145 supports *one* program master (PM) path.

- *Program slave paths*—For accepting programming commands received over the network. A 984-145 supports *one* program slave (PS) path.

Both the originating and destination devices open paths and maintain them until the transaction completes. If the transaction passes through one or more Bridge Plus devices to access a destination across multiple networks, each bridge opens and maintains a path at each of its two network ports. Thus a logical path is maintained between the originating and destination devices until the transaction is finished.

All paths are independent of one another, and activity on one path does not affect the performance of the other paths.

Chapter 3
Essentials of Ladder Logic Programming

- ❐ Segments and Networks
- ❐ Standard Ladder Logic Elements
- ❐ Application Example: A Motor Start/Stop Circuit
- ❐ Standard PLC Instructions
- ❐ Instructions Available on Select Compact Models

Segments and Networks

Ladder Logic Segments

All the ladder logic required to control your application is stored in a logic *segment* in user memory. If you are calling subroutines as part of your application, the subroutine logic must be placed in a separate segment.

The Compact Controllers allow you to configure up to 32 logic segments. The last segment is where all subroutine logic is stored. Subroutines logic is scanned only when it is called, either from the ladder logic or from an external event that triggers an interrupt.

Ladder Logic Networks

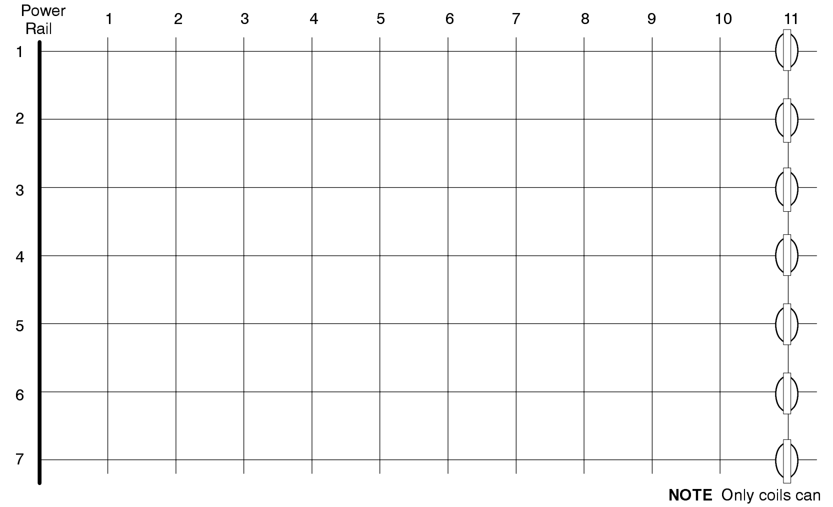
Each segment is composed of a group of contiguous *networks*. Each network is a small, clearly defined ladder diagram bounded on the left by a power rail and on the right by a rail that, by convention, is not displayed. The ladder is seven rungs high by eleven columns wide.

The intersection of each rung and column in the network is called a node—each network contains 77 nodes.

The number of networks in a segment is limited by the amount of *user program memory* available in the CPU and by the time it takes for the CPU to scan the ladder logic program.

Placing Relay Logic and Instructions in a Network

Each time you use an *relay logic element*—e.g., a contact, a coil, a horizontal short—in ladder logic, the element consumes one node in the logic network.



Ladder Logic Network Structure

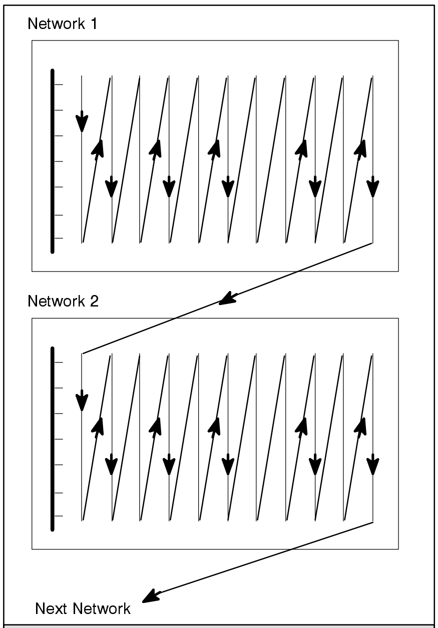
NOTE Only coils can be shown in column 11

An *instruction* in ladder logic may consume one, two, or three nodes in a network, depending on the instruction type. A counter instruction, for example, is a two-high nodal instruction—it consumes two contiguous nodes that must be one over the other. An ADD instruction, on the other hand, is a three-high nodal instruction consuming three contiguous nodes stacked over each other.

How Ladder Logic Is Solved

A Compact Controller scans the ladder logic program sequentially in the following order:

- ❐ Segment by segment
- ❐ Network 1 through network *n* sequentially within each segment
- ❐ Node by node within each network, starting in the upper left node of the ladder and moving top to bottom, then left to right



Power Flow in and between Ladder Logic Networks

Relay Logic Elements

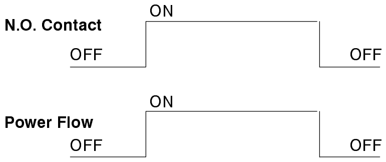
There are three general types of relay logic elements used in ladder logic programming—contacts, coils, and shorts.

Each relay logic element consumes one node in a ladder network.

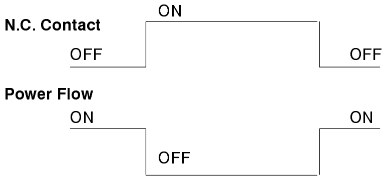
Relay Contacts

Contacts are used to pass or inhibit power flow in a ladder logic program. Four kinds of contacts may be used:

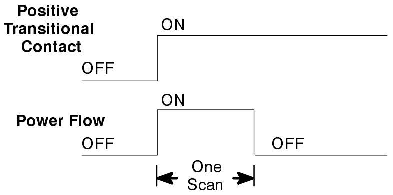
- The normally open (N.O.) contact, which passes power when its referenced coil or input is ON:



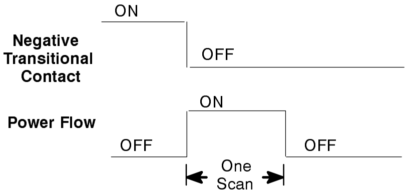
- The normally closed (N.C.) contact, which passes power when its referenced coil or input is OFF:



- The positive transitional contact, which passes power for only one scan as the contact or coil transitions from OFF to ON:



- The negative transitional contact, which passes power for only one scan as the contact or coil transitions from ON to OFF:



The symbols used in ladder logic to represent contact types are shown in the table below.

Element	Symbol	Function	Memory Utilization
N.O. Contact	–| |–	Passes power when its referenced coil or input is ON	Can be referenced to a logic coil in a 0*x* register or to a discrete input in a 1*x* register
N.C. Contact	–|/|–	Passes power when its referenced coil or input is OFF	Can be referenced to a logic coil in a 0*x* register or to a discrete input in a 1*x* register
Positive Transitional Contact	–|↑|–	Passes power for one scan as the contact or coil transitions from OFF to ON	Can be referenced to a logic coil in a 0*x* register or to a discrete input in a 1*x* register
Negative Transitional Contact	–|↓|–	Passes power for one scan as the contact or coil transitions from ON to OFF	Can be referenced to a logic coil in a 0*x* register or to a discrete input in a 1*x* register

Normal and Memory-retentive Coils

Element	Symbol	Function	Memory Utilization
Normal Coil	—( )—	Turns OFF when power is removed	A discrete output value represented by a 0*x* reference number; may be used internally in the logic program or externally to a discrete output
Memory-retentive Coil	—(M)—	Coil comes back in the same state when power is restored for one scan	A discrete output value represented by a 0*x* reference number; may be used internally in the logic program or externally to a discrete output

A coil is a discrete output value represented by a 0*x* reference bit. Because output values are updated in state RAM by the CPU, a coil may be used internally in the logic program or externally via the I/O map to a discrete output unit in the control system.

A coil is either ON or OFF, depending on power flow. When a coil is ON, it either passes power to a discrete output circuit or changes the state of the associated internal relay contact in state RAM.

There are two types of coils—*normal* coils and *memory-retentive* coils. When power is applied or restored to a normal coil, any value previously held by the coil is cleared prior to the first logic scan of the PLC. With a memory-retentive coil, the value previously held by the coil is retained for one scan, then the logic takes control.

Displaying Coils in a Network

A ladder network can contain a maximum of seven coils. No logic elements except coils are allowed in the eleventh column. If a coil appears on a rung in a column other than 11, no other logic element can be placed to the right of the coil on that rung.

Vertical and Horizontal Shorts

Shorts are simply straight-line connections between instruction blocks and/or contacts in a ladder logic network.

A *vertical* short connects contacts or instruction blocks one above the other in a network column. Vertical shorts can also be used to connect inputs or outputs to create either/or conditions such as the one illustrated on the following page. When two contacts are connected by a vertical short, power is passed when one or both contact(s) receive power. A vertical short does not consume any user memory.

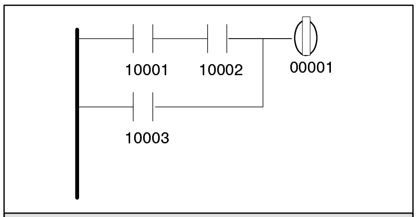
Horizontal shorts are used to expand a rung in a ladder logic network without breaking the power flow. Each horizontal short used in a program consumes one word of user logic memory.

On the following page are two examples of how horizontal and vertical shorts can be used together with relay contacts to create ladder logic.

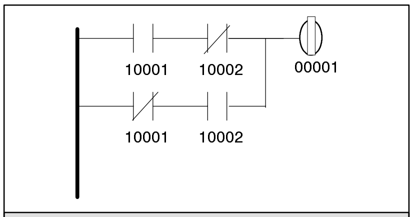
The first example is a simple either/or condition—the top rung of ladder contains two N.O. contacts (10001 and 10002), and the lower rung contains a single contact (10003) followed by a horizontal short. A vertical short connects the two rungs after the second column. Power can pass through the network to energize coil 00001 when *either* contacts 10001 and 10002 are energized *or* when contact 10003 is energized.

The second example shows an Exclusive-OR circuit built with similar contacts and shorts. This circuit can be used to prevent coil 00001 from energizing when two conditions, represented by contact 10001 and contact 10002, are activated simultaneously.

In both examples, the vertical shorts, which do not consume any user program memory, are treated as part of the node in which contact 10002 is programmed.

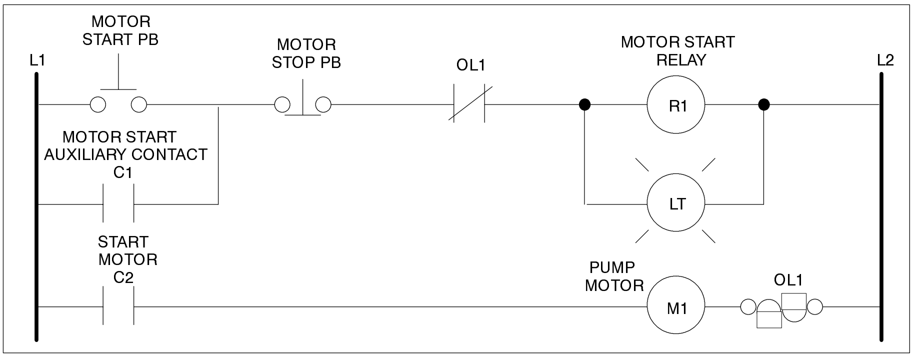


Example 1: Either/Or Relay Logic



Example 2: Exclusive-OR Relay Logic

Application Example: A Motor Start/Stop Circuit



Above is an example of a standard across-the-line electrical diagram for a pushbutton-activated motor start/stop circuit.

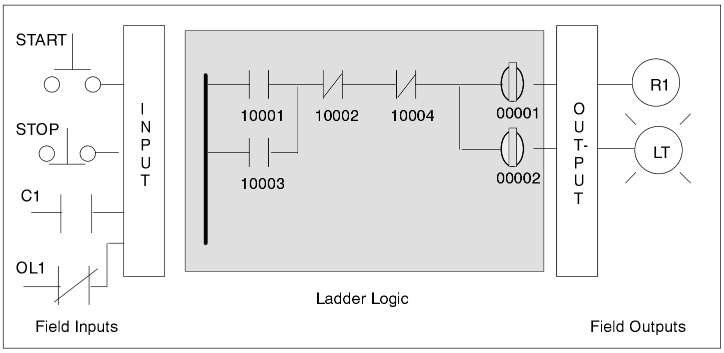
Pushing the *motor start* pb energizes motor control relay R1 and closes contact C2 to start motor M1. The auxiliary contacts on motor control relay C1 also close, allowing the motor start/stop circuit to be latched ON. Two things can cause relay R1 to drop out:

- An overload (OL1) on motor M1
- The *motor stop* pb is pushed

Now let's look at an implementation of the same circuit using contacts, coils, and shorts in a ladder logic network.

We see in the illustration below that the sequence of operation remains essentially the same when the motor start/stop circuit is designed for the controller. The big difference is that all the I/O points are wired directly to input/output modules contained in the programmable control system, and the actual control is programmed in ladder logic.

The ladder logic implementation allows greater flexibility of control and decreased development time, since all the *hard-wiring* between points of control is done electronically.



Chapter 4 Counters and Timers

- ❐ Counter Instructions
- ❐ Timer Instructions
- ❐ Application Example: Logic for a Real-time Clock

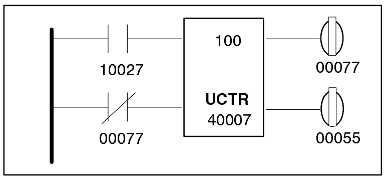
Counter Instructions

Two counter instructions are provided. The up-counter (UCTR) counts up from 0 to a preset value, and the down-counter (DCTR) counts down from a preset value to 0. Both are two-high nodal instructions.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Up-counter	I — 3*x*, 4*x*, or K* — O I — UCTR 4*x* — O	*Top:* ON initiates counter *Bottom:* 0 = reset 1 = enabled	*Top:* counter preset *Bottom:* accumulated count	*Top:* count = preset *Bottom:* count < preset	Counts up from 0 to a preset value
Down-counter	I — 3*x*, 4*x*, or K* — O I — DCTR 4*x* — O	*Top:* ON initiates counter *Bottom:* 0 = reset 1 = enabled	*Top:* counter preset *Bottom:* accumulated count	*Top:* count = 0 *Bottom:* count > preset	Counts down from a preset value to 0
*K is an integer constant in the range 1 ... 999.					

A Simple Up-counter Example

When contact 10027 is energized, the top input to UCTR receives power; since contact 00077 also passes power, the instruction is enabled. Each time contact 10027 transitions from OFF to ON, the accumulated count increments by 1. When the value reaches 100, the top output passes power—coil 00077 is energized, and coil 00055 is de-energized. Contact 00077 opens when coil 00077 is energized, and the accumulated count is reset to 0 on the next scan. On the next scan, coil 00077 is de-energized; contact 00077 closes and the UCTR is enabled.



Timer Instructions

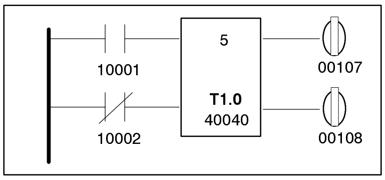
The three timer instructions can be used to time events or create delays in an application. They are two-high nodal instructions.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
One-second timer	I — 3*x*, 4*x*, or K* — O I — **T1.0** 4*x* — O	*Top:* ON when bottom input = 1 *Bottom:* 0 = reset 1 = enabled	*Top:* timer preset *Bottom:* accumulated time	*Top:* time = preset *Bottom:* time < preset	Timer increments at intervals of one second
Tenth-of-a second timer	I — 3*x*, 4*x*, or K* — O I — **T0.1** 4*x* — O	*Top:* ON when bottom input = 1 *Bottom:* 0 = reset 1 = enabled	*Top:* timer preset *Bottom:* accumulated time	*Top:* time = preset *Bottom:* time < preset	Timer increments at intervals of 0.1 s
Hundredth-of a-second timer	I — 3*x*, 4*x*, or K* — O I — **T.01** 4*x* — O	*Top:* ON when bottom input = 1 *Bottom:* 0 = reset 1 = enabled	*Top:* timer preset *Bottom:* accumulated time	*Top:* time = preset *Bottom:* time < preset	Timer increments at intervals of 0.01 s

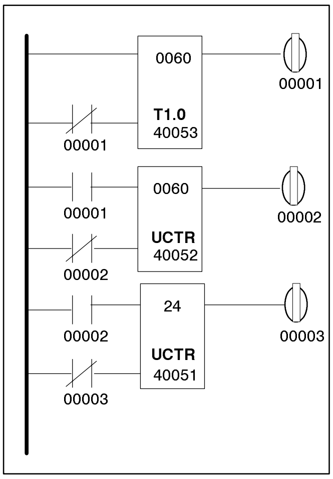
*K is an integer constant in the range 1 ... 999.

A One-second Timer Example

When contact 10002 is closed—i.e., the timer is enabled—the value contained in register 40040 is 0. Coil 00108 is ON and 00107 is OFF. When contact 10001 is closed, the count accumulates in register 40040 at one-second intervals until 5 is reached; coil 00107 goes ON and 00108 goes OFF. When contact 10002 is opened, the value in register 40040 is reset to 0, coil 00107 goes OFF, and 00108 goes ON.



Application Example: Logic for a Real-time Clock



This example shows the ladder logic for a real-time clock with one-second accuracy. The T1.0 instruction is programmed to pass power at 1 min intervals. When logic solving begins, coil 00001 is OFF, and both the top and bottom inputs of the timer instruction receive power.

Register 40053 in the bottom node of the T1.0 instruction starts incrementing time in seconds. After 60 increments, the top output passes power to energize coil 00001 and opens N.C. contact 00001 to reset register 40053 to 0.

N.C. contact 00001 closes and passes power to the count input of the first UCTR. The accumulated count value in register 40052 increments by 1, indicating that one minute has passed.

Because the accumulated time count in T1.0 no longer equals the timer preset, coil 00001 loses power, N.C. contact 00001 closes, and the timer begins to re-accumulate time in seconds and continues to increment the first UCTR. When the accumulated count in register 40052 of the first UCTR instruction increments to 60, the top output passes power and energizes coil 00002.

N.C. contact 00002 opens, and the value in register 40052 resets to 0. N.C. contact 00002 closes, and the accumulated count in register 40051 of the second UCTR instruction increments by 1. This indicates that an hour has passed.

The time of day can be read in registers 40051 (indicating the hour count), 40052 (indicating the minute count), and 40053 (indicating the second count).

Chapter 5
Basic Math Instructions

Integer Math Instructions

Standard addition, subtraction, multiplication, and division instructions are provided for calculating integer math operations. Each of the four instructions is a three-high nodal instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Integer Addition	I — 3*x*, 4*x*, or K* — O 3*x*, 4*x*, or K* **ADD** 4*x*	*Top:* ON enables a (val 1) + (val 2) operation	*Top:* value 1 *Middle:* value 2 *Bottom:* sum	*Top:* sum > 9999	Adds the values in the top and middle nodes, then stores the result in a 4*x* register in the bottom node
Absolute (no signs in the values) Integer Subtraction	I — 3*x*, 4*x*, or K* — O 3*x*, 4*x*, or K* — O **SUB** 4*x* — O	*Top:* ON enables a (val 1) – (val 2) operation	*Top:* value 1 *Middle:* value 2 *Bottom:* difference	*Top:* val 1 > val 2 *Middle:* val 1 = val 2 *Bottom:* val 1 < val 2	Subtracts the middle node value from the top node value and stores the difference in a 4*x* register in the bottom node
Integer Multiplication	I — 3*x*, 4*x*, or K* — O 3*x*, 4*x*, or K* **MUL** 4*x*	*Top:* ON enables a (val 1) x (val 2) operation	*Top:* value 1 *Middle:* value 2 *Bottom:* product (high order digits)	*Top:* echos the top input	Multiplies the values in the top and middle nodes, then stores the product in two contiguous 4*x* registers
Integer Division with remainder	I — 3*x*, 4*x*, or K* — O I — 3*x*, 4*x*, or K* — O **DIV** 4*x* — O	*Top:* ON enables a (val 1) / (val 2) operation Middle: 0 = fractional remainder 1 = decimal remainder	*Top:* value 1** *Middle:* value 2 *Bottom:* result (remainder in reg 4*x* + *1*)	*Top:* division successful *Middle:* if result > 9999 a value of 0 is returned *Bottom:* value 2 = 0	Divides the top node value by the middle node value, then stores the result in the 4*x* register in the bottom node and the remainder in register 4*x* + 1

*K is an integer constant in the range 1 ... 999.

** If value 1 of the DIV instruction is stored 3*x* or 4*x* registers, then the register shown in the top node is the first of two contiguous registers. The high-order half of value 1 is stored in the displayed register (3*x* or 4*x*) and the low-order half of value 1 is stored in the next contiguous register (3*x* + 1 or 4*x* + 1).

The MUL and DIV blocks require that two contiguous registers be used in the bottom node. The first of the two registers is seen in the block, and the presence of the second register is implicit.

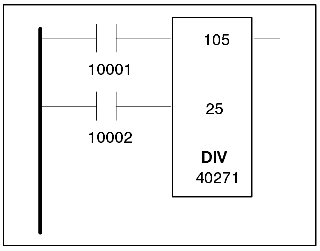
In the MUL instruction block, the high-order portion of the calculated product is stored in the first bottom-node register and the low-order portion of the product is stored in the second bottom-node register.

In the DIV instruction block, the quotient is stored in the first bottom-node register and the remainder is stored in the second bottom-node register. If you do not use a constant as the top-node value in a DIV instruction, then it the value must be placed in two contiguous 3*x* or 4*x* registers. The high-order half of the value is stored in the displayed register, and the low-order half of the value is stored in the implied register.

For example, if the top-node value is 105 and it were to be placed in two contiguous registers, 40025 and 40026, instead of being given as a constant, then register 40025 would contain all zeros and register 40026 would contain the value 105.

A DIV Example

Here is an example of a DIV operation where the top-node value, 105, is divided by the middle-node value, 25. The quotient (4) is stored in register 40271, and the remainder (5) is stored in register 40272.



When the middle input—contact 10002—is open, the remainder is expressed as a fraction (0005); when contact 10002 is closed, the remainder is expressed as a decimal (2000).

Application Example: Fahrenheit-to-Centigrade Conversion

This example implements the formula

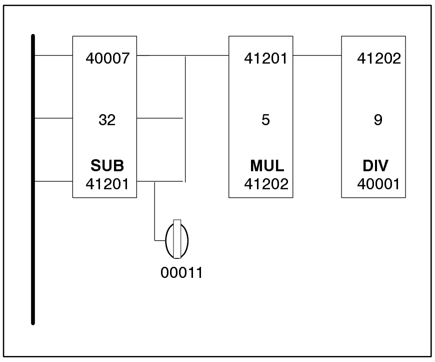
$$°C = (°F - 32) \times {}^{5}/_{9}$$

When the top input to the SUB instruction block receives power, the value in the middle node, 32, is subtracted from the value stored in register 40007, some number of degrees Fahrenheit. The difference is placed in register 41201.

The top input to the MUL instruction block then receives power, regardless of whether the subtraction result is positive, negative, or 0. In the case where the subtraction result is negative, coil 00011 is energized to indicate a negative value.

The value in the top-node register of the MUL block—register 41201—is then multiplied by 5 and the product is placed in register 41202 and implicit register 41203.

The top node in the DIV instruction block is then energized, and the value in registers 41202 and 41203 is divided by 9. The quotient, which is the temperature conversion in degrees Centigrade, is stored in register 40001 (and the remainder in implicit register 40002).



Note: The vertical short to coil 00011 (indicating a negative value) must be placed to the left of the vertical shorts that link the three SUB block output.

Chapter 6 Data Management Instructions

- ❐ Moving Register and Table Data
- ❐ Building a FIFO Stack
- ❐ Searching a Table
- ❐ Moving a Block of Data

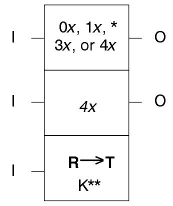
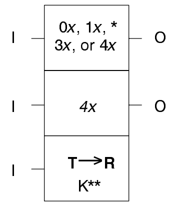
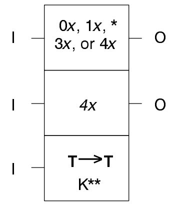
Moving Register and Table Data

Three standard instruction blocks are provided for moving the data stored in registers and in tables of registers:

- ❐ A register-to-table (R→T) DX move
- ❐ A table-to-register (T→R) DX move
- ❐ A table-to-table (T→T) DX move

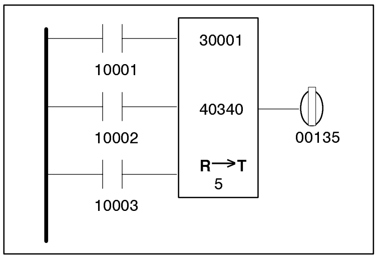
A Compact Controller can accommodate the transfer of one register per scan for each instruction in a ladder logic program.

Each is a three-high nodal instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Register-to-table move	 I — 0*x*, 1*x*, * 3*x*, or 4*x* — O I — 4*x* — O I — R→T K**	*Top:* ON moves data and increments pointer *Middle:* ON freezes the pointer *Bottom:* ON resets the pointer to 0	*Top:* source register *Middle:* pointer to the target register (4*x* + 1) in the destination table *Bottom:* Table length*	*Top:* echos the top input *Middle:* pointer = table length	Copies a 16-bit pattern in a source register to a register in the destination table; the destination register is pointed to by the 4*x* register in the middle node
Table-to-register move	 I — 0*x*, 1*x*, * 3*x*, or 4*x* — O I — 4*x* — O I — T→R K**	*Top:* ON moves data and increments pointer *Middle:* ON freezes the pointer *Bottom:* ON resets the pointer to 0	*Top:* source table *Middle:* pointer to the destination register (4*x* + 1) *Bottom:* Table length*	*Top:* echos the top input *Middle:* pointer = table length	Copies the bit pattern of a register in the source table to a destination register (register 4*x* + 1 in the middle node)
Table-to-table move	 I — 0*x*, 1*x*, * 3*x*, or 4*x* — O I — 4*x* — O I — T→T K**	*Top:* ON moves data and increments pointer *Middle:* ON freezes the pointer *Bottom:* ON resets the pointer to 0	*Top:* source table *Middle:* pointer to the target register (4*x* + 1) in the destination table *Bottom:* Table length*	*Top:* echos the top input *Middle:* pointer = table length	Copies the bit pattern of a register in the source table to a register in the same position in a destination table; the destination register is pointed to by the 4*x* register in the middle node

* If you use a 0*x* or 1*x* reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** K is an integer constant in the range 1 ... 255.



The ladder logic example shown above moves the value stored in register 30001 into a destination table of five holding registers, 40341 ... 40345. One 30001 register value is moved into one of the table registers in every scan.

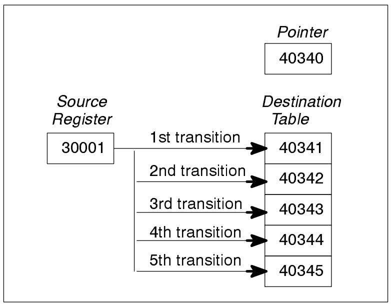
The pointer to the destination table—register 40340—is specified in the middle node of the register-to-table instruction block, and the number of holding registers in the table, 5, is specified in the bottom node.

When contact 10001 transitions ON for the first time, the current contents of register 30001 are copied to register 40341, the first of five contiguous registers in the destination table. The first table in the destination register is always the next contiguous register after the pointer reference number given in the middle node of the instruction block. When this DX move takes place, the value in the pointer register increments from 0 to 1.

In the next scan of contact 10001, the contents of register 30001 are copied into register 40432, the second register in the destination table; the value in the pointer register increments from 1 to 2.

This process continues until the contents of register 30001 are copied into register 40345 in the table and the pointer value has incremented to 5. At this point, the middle output from the block passes power and energizes coil 00135.

No further register-to-table moves are possible while the value of the pointer equals the table length specified in the bottom node of the block.



If, after the second transition of contact 10001, contact 10002 were to become energized, the pointer value would be frozen-i.e., it could not be incremented or decremented—and subsequent transitions of contact 10001 would cause the current value in register 30001 to be copied to register 40343.

If contact 10003 is energized, the value of the pointer is reset to 0.

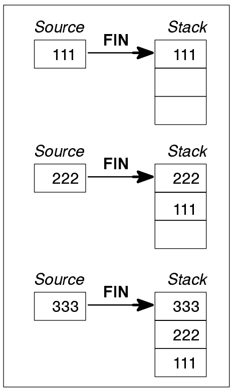
Building a FIFO Stack

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
First-in to a queue stack	I — 0*x*, 1*x*, * 3*x*, or 4*x* — O 4*x* — O **FIN** K** — O	*Top:* ON inserts a bit pattern in the top of the stack	*Top:* The source register in the stack *Middle:* pointer to the register in the stack where the source bits will be inserted *Bottom:* stack length*	*Top:* echos the top input *Middle:* stack is full *Bottom:* stack is empty	Copies a 16-bit pattern into a register at the top of a stack; the table begins at register 4*x* + 1 of the middle node
First-out of a queue stack	I — 4*x* — O 0*x* or 4*x* — O **FOUT** K** — O	*Top:* ON removes the bit pattern from the bottom of the stack	*Top:* pointer to the source register in the stack *Middle:* destination register where source bits will be moved *Bottom:* stack length*	*Top:* echos the top input *Middle:* stack is full *Bottom:* stack is empty	Moves the bit pattern in the bottom register of the stack to a destination register out of the stack

* If you use a 0*x* or 1*x* reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** K is an integer constant in the range 1 ... 255.

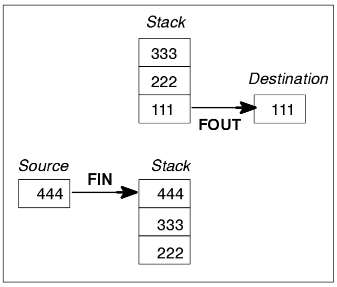
The two instructions above let you queue data into a first-in/first-out stack. The FIN instruction copies the bit pattern of a register or of 16 discretes into a register at the top of a table (or stack) of holding registers.



The FOUT instruction moves the bit pattern down through the stack, then out of the stack and into a destination table.

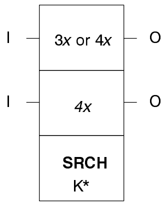
STOP **Warning FOUT will override any disabled coils in a destination table without enabling them. If a coil has been disabled for repair or maintenance, there is the potential for injury, since that coil's state can change as a result of the FOUT operation.**

When you are running a FIFO stack in ladder logic, the FOUT instruction should be executed in each scan before the FIN instruction so that the oldest data in the stack can be cleared to the destination table before the newest data is queued into the stack. If the FIN block is executed first, an attempt to enter data into a filled stack is ignored.



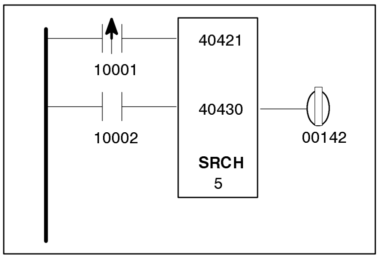
Searching a Table

The SRCH instruction allows you to search a table of registers for a specific bit pattern contained in one of the table registers. SRCH is a three-high nodal instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Table search		*Top:* ON initiates a search *Middle:* 0 = search from the beginning 1 = search from last match	*Top:* first register in the source table *Middle:* 4*x* pointer to the location in the table of the register holding the value searched for; the next register, 4*x* + 1, contains the value being searched for *Bottom:* Table length*	*Top:* echos the top input *Middle:* match found	Searches a table of registers for the bit pattern specified in the register immediately following the pointer in the middle node

* K is an integer constant in the range 1 ... 255.

An Example of a SRCH Operation



The source table to be searched is five registers long starting at holding register 40421, and the content of the table registers is as follows:

Source Table Registers		*Register Content*
40421	=	1111
40422	=	2222
40423	=	3333
40424	=	4444
40425	=	5555

The bit pattern to be searched for is 3333, which is the value that gets entered into register 40431 (the register immediately following the pointer register in the middle node).

When contact 10001 transitions from OFF to ON, the logic searches the source table for the register that contains 3333. When that value is found (in register 40423), the pointer value in register 40430 is set to 3, indicating that the third register in the source table contains the searched-for value; coil 00142 is also energized for one scan.

Moving a Block of Data

The block move (BLKM) instruction copies the entire contents of a source table of registers to a destination table in one logic scan. BLKM is a three-high nodal instruction.

Warning BLKM will override any disabled coils in a destination table without enabling them. If a coil has been disabled for repair or maintenance, there is the potential for injury, since that coil's state can change as a result of the BLKM operation.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Block move	I — 0*x*, 1*x*, * 3*x*, or 4*x* — O 0*x*** or 4*x* **BLKM** K***	*Top:* ON initiates a block move	*Top:* source table *Middle:* destination table *Bottom:* Table length*	*Top:* echos the top input	Copies the entire contents of one table to another table of outputs or holding registers

* If you use a 0*x* or 1*x* reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** If 0*x* references are used as the destination, they cannot be programmed as coils, only as contacts referencing those coil numbers

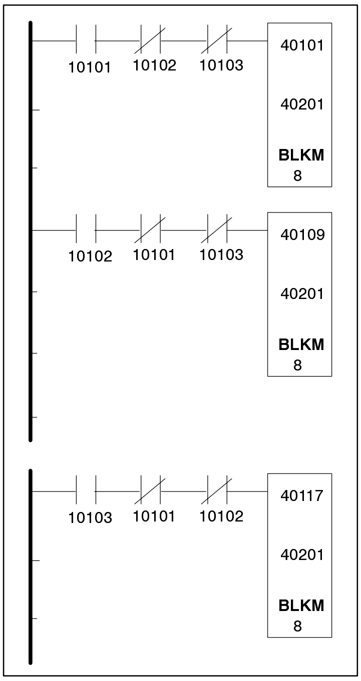
*** K is an integer constant in the range 1 ... 100.

Application Example: A Recipe Loading Routine Using Block Moves

A ladder logic program can store a collection of specific process recipes, each in a unique storage table and loadable on demand to a working table where a generic process is being run. The recipes must be structured with similar types of information in corresponding registers—if heating temperature information is kept in the third register of one recipe, similar information should be kept in the third register of all the other recipes as well.

Specific recipes can be loaded to and removed from the generic process via BLKM instructions.

The logic example shown on the next page contains an eight-register working table (registers 40201 ... 40208) in which three different recipes can be run. Recipe selection is handled by three input switches, contacts 10101, 10102, and 10103.



To run process A, for example, turn contact 10101 ON and leave contacts 10102 and 10103 OFF. When input 10101 is energized, it passes power through N.C. contacts 10102 and 10103, and the first BLKM block moves the recipe for process A from registers 40101 ... 40108 to registers 40201 ... 40208.

Chapter 7
Data Manipulation Instructions

Boolean Logic Instructions

Three instructions are available to perform ANDing, ORing, and XORing logic operations.

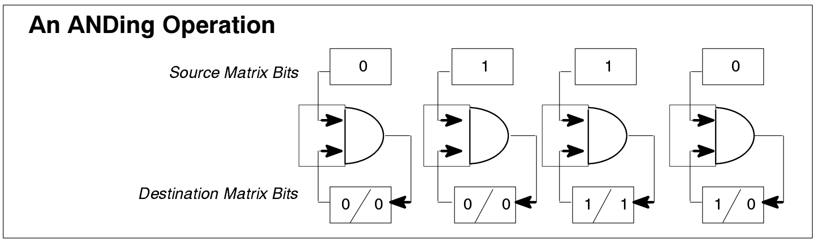
Warning These Boolean instructions will override any disabled coils in the destination matrix without enabling them. If a coil has been disabled for repair or maintenance, there is the potential for injury, since that coil's state can change as a result of the logic operation.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Boolean AND	I — 0x, 1x, * 3x, or 4x — O 0x** or 4x **AND** K****	*Top:* Initiates a logical AND operation	*Top:* source matrix *Middle:* destination matrix *Bottom:* matrix length*	*Top:* echos the top input	ANDs the bits in the source matrix with the equivalently positioned bits in the destination matrix, then places the results in the destination matrix, overwriting the original bit pattern
Boolean OR	I — 0x, 1x, * 3x, or 4x — O 0x** or 4x **OR** K***	*Top:* Initiates a logical OR operation	*Top:* source matrix *Middle:* destination matrix *Bottom:* matrix length*	*Top:* echos the top input	ORs the bits in the source matrix with the equivalently positioned bits in the destination matrix, then places the results in the destination matrix, overwriting the original bit pattern
Boolean exclusive OR	I — 0x, 1x, * 3x, or 4x — O 0x** or 4x **XOR** K***	*Top:* Initiates a logical XOR operation	*Top:* source matrix *Middle:* destination matrix *Bottom:* matrix length*	*Top:* echos the top input	XORs the bits in the source matrix with the equivalently positioned bits in the destination matrix, then places the results in the destination matrix, overwriting the original bit pattern

* If you use a 0x or 1x reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

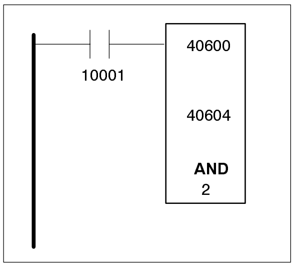
** If 0x references are used as the destination, they cannot be programmed as coils, only as contacts referencing those coil numbers

*** K is an integer constant in the range 1 ... 100


An ANDing Operation

An AND instruction logically ANDs each bit in a source matrix with the corresponding bits in a destination matrix, then posts the results in the destination matrix—overwriting the previous bit pattern in the destination matrix.

For example, when contact 10001 passes power in the network below, the bit matrix comprising registers 40600 and 40601 are ANDed with the bit matrix comprising registers 40604 and 40605.



The result is then copied into registers 40604 and 40605, overwriting the previous bit pattern.

Source Matrix

40600	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
40601	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

Original Destination Matrix

40604	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
40605	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

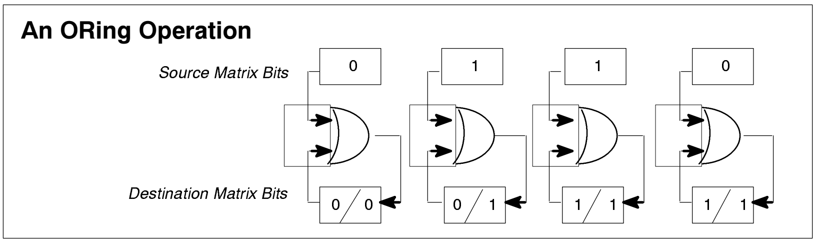
ANDed Destination Matrix

40604	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
40605	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

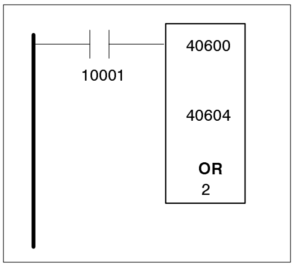
OR

Likewise, an OR instruction logically ORs the bits in a source matrix with the corresponding bits in a destination matrix, then overwrites the destination matrix with the results of the operation.

Note Outputs and coils cannot be turned OFF with the OR instruction.


An ORing Operation

For example, if we were to OR the same two matrixes as in the example shown above:



the result would be:

	Source Matrix
40600	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
40601	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

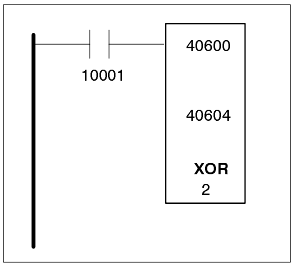
	Original Destination Matrix
40604	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
40605	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

	ORed Destination Matrix
40604	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
40605	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

XOR

The exclusive OR instruction logically XORs the bits in a source matrix with the corresponding bits in a destination matrix, then overwrites the destination matrix with the results of the operation.

For example, if we were to XOR the same two matrixes as in the example shown above:



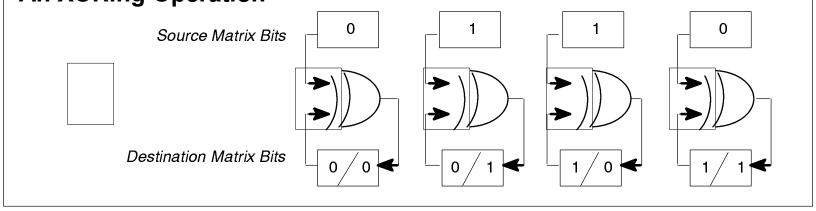
the result would be:

	Source Matrix
40600	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
40601	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

	Original Destination Matrix
40604	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
40605	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

	XORed Destination Matrix
40604	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
40605	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

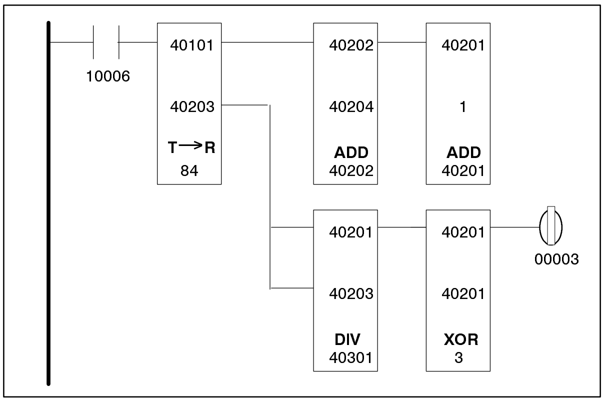
An XORing Operation



Archiving the Original Destination Matrix Values

If you want to save the original bit pattern from the registers in the destination matrix, use the BLKM instruction to copy the information into another table before running the Boolean logic operation.

An Application Example: Simple Table Averaging



Here is an application routine that combines three integer math calculations with a data transfer and an XOR instruction. It calculates the average value of the 84 values stored in the table of registers 40101 ... 40184.

When contact 10006 closes, the top node in the table-to-register instruction receives power, initiating the data transfer. The value in the first register of the table is copied into the middle node of the first ADD instruction, and the table pointer value increments register 40203 in the middle node of both the table-to-register instruction and the DIV instruction. Because the top output from the table-to-register instruction passes power, the first ADD block receives power and adds the value in register 40204 to the value in register 40202 (which is initially 0); then the sum of this addition overwrites the previous value in register 40202.

The routine continues to run this way until all the values in the table of 84 registers have been added together. At this point, the pointer value in the middle node of the table-to-register instruction is 84, and the middle output from that instruction passes power and enables the DIV instruction.

The values in registers 40201 (all 0s, representing the high-order portion of the sum of all the register values in the table) and 40202 (the low-order portion of the sum) are divided by 84. The result is placed in register 40301, and the remainder is placed in register 40302. (Because there is power to the middle input of the DIV instruction, the remainder is expressed as a decimal.) The result of the DIV operation is the average value of the current values stored in all 84 registers in the table.

When the top output from the DIV instruction passes power, the XOR instruction becomes empowered. It exclusively ORs the values in registers 40201 ... 40203 with themselves, clearing the matrix to 0s and indicating that the current table averaging operation is complete and that a new one should start.

Bit Complementing in a Data Matrix

The COMP instruction complements the bit pattern in a matrix—i.e., changes all the 0s to 1s and all the 1s to 0s—then copies the result in a second matrix. A matrix can be complemented in one scan.

COMP is a three-high nodal instruction.

Warning COMP will override any disabled coils in a destination matrix without enabling them. If a coil has been disabled for repair or maintenance, there is the potential for injury, since that coil's state can change as a result of the COMP operation.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Bit complement	I — 0*x*, 1*x*, * 3*x*, or 4*x* — O 0*x*** or 4*x* **COMP** K***	*Top:* ON initiates the bit complement operation	*Top:* source matrix *Middle:* destination matrix *Bottom:* matrix length*	*Top:* echos the top input	Complements the bit values in the source matrix and places the results in the destination matrix

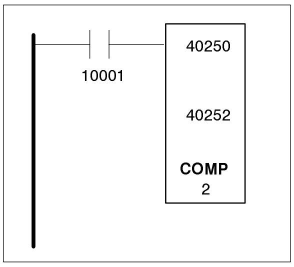
* If you use a 0*x* or 1*x* reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** If 0*x* references are used as the destination, they cannot be programmed as coils, only as contacts referencing those coil numbers

*** K is an integer constant in the range 1 ... 100

A Bit Complement Example

The ladder logic below shows a COMP block with a source matrix composed of two registers—40250 and 40251—and a destination matrix composed of registers 40252 and 40253.



When contact 10001 passes power the block complements the bit values in the source register and places the results in the destination register.

Source Matrix

40250	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
40251	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

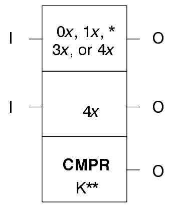
Complemented Destination Matrix

40252	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
40253	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

All values stored in the destination register before the COMP instruction is enabled will be overwritten by the complemented source values as a result of the COMP operation.

Bit Comparison in a Data Matrix

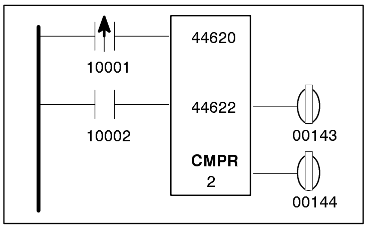
The CMPR instruction compares the bit pattern in one register matrix with the bit pattern in another matrix. When a bit value in one matrix miscompares with the correspondingly positioned bit value in the other matrix, a value indicating that matrix location is posted in the middle node.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Bit compare	 I 0x, 1x, * 3x, or 4x O I 4x O CMPR K** O	*Top:* ON initiates the bit compare *Middle:* 0 = restart at last miscompare 1 = restart at the beginning	*Top:* matrix a *Middle:* posts the bit position of the currently detected miscompared bit and points to matrix b, which begins at 4x + 1 *Bottom:* matrix length*	*Top:* echos the top input *Middle:* miscompare detected *Bottom:* state of miscompared bit in matrix a	Compares bit patterns in matrixes a and b, and reports miscompares

* If you use a 0x or 1x reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** K is an integer constant in the range 1 ... 100

A Bit Comparison Example



This example shows a bit comparison between two two-register matrixes. Matrix a comprises registers 44620 and 44621; matrix b comprises registers 44623 and 44624:

	Matrix a
40600	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
40601	1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
	Matrix b
40604	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
40605	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Matrix a is compared against matrix b bit by bit on every scan that contact 10001 transitions from OFF to ON until one miscompare is found.

In the first transition of contact 10001, the matrix bits are compared until bit 17, where the value in matrix a = 1 and the value in matrix b = 0. At this point, a value of 17 is posted in register 44622, the comparison stops, and coils 00143 and 00144 energize for one scan.

If contact 10002 is energized, the function will begin to compare at matrix position 1 in the next transition of 10001 and stop again when the value in register 44622 = 17. If contact 10002 is not energized, the function will begin to compare at matrix position 18 in the next transition of 10001 and stop when the value in register 44622 = 25.

Sensing and Manipulating Bits in a Data Matrix

Three instructions are provided to let you examine and manipulate the bit patterns in a data matrix:

- The bit-sense (SENS) instruction examines and reports the sense—1 or 0—of specific bits in the matrix
- The bit-modify (MBIT) instruction modifies the sense of a specific bit in a matrix—i.e., changes a 0 bit to 1 and clears a 1 bit to 0
- The bit-rotate (BROT) instruction shifts the bit pattern in a matrix to the left or right, forcing the exiting bit to either fall out of the matrix or wrap onto the other end of the register

One bit per scan may be sensed, modified, or rotated via these instructions. Each is a three-high nodal instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Bit rotation	I — 0*x*, 1*x*, * 3*x*, or 4*x* — O I — 0*x*** or 4*x* — O I — **BROT** K***	*Top:* ON initiates the bit rotation *Middle:* 0 = start left 1 = start right *Bottom:* 0 = bit falls out of the register 1 = bit wraps to start of register	*Top:* source matrix *Middle:* destination matrix *Bottom:* matrix length*	*Top:* echos the top input *Middle:* sense of the bit rotating out of the matrix	Rotates or shifts the bit pattern in a matrix, shifting the bits one position per scan
Bit sensing	I — 3*x*, 4*x*, or K_1*** — O I — 0*x*** or 4*x* — O I — **SENS** K_1** — O	*Top:* ON reports the sense of the matrix bits *Middle:* increments the pointer after a bit sense *Bottom:* resets the pointer to 1	*Top:* pointer to the matrix *Middle:* address of first register in the matrix *Bottom:* matrix length**	*Top:* echos the top input *Middle:* copies the sensed bit *Bottom:* pointer > matrix length	Examines and reports the sense of specific bits—i.e., 1 or 0—in a matrix; one bit per scan can be sensed
Bit modification	I — 3*x*, 4*x*, or K_1*** — O I — 0*x*** or 4*x* — O I — **MBIT** K_1*** — O	*Top:* ON changes the sense of the matrix bits *Middle:* 0 = clear bit 1 = set bit *Bottom:* increments the pointer after bit modification	*Top:* pointer to the matrix *Middle:* address of first register in the matrix *Bottom:* matrix length**	*Top:* echos the top input *Middle:* echos the middle input *Bottom:* pointer > matrix length	Changes the value of a bit in the matrix from 0 to 1 or from 1 to 0; one bit per scan can be modified

* If you use a 0*x* or 1*x* reference, it must be given as a multiple of 16 + 1 (1, 17, 33, etc.), and it implies the use of 16 discrete bits (1 ... 16, 17 ... 32, 33 ... 48, etc.).

** If 0*x* references are used as the destination, they cannot be programmed as coils, only as contacts referencing those coil numbers

*** K is an integer constant in the range 1 ... 100 ; K_1 is an integer constant in the range 1 ... 255

Warning MBIT and BROT will override any disabled coils in the matrix without enabling them. If a coil has been disabled for repair or maintenance, there is the potential for injury, since that coil’s state can change as a result of bit manipulation.

Chapter 8
The MSTR Instruction

- ❐ Overview
- ❐ MSTR Function Error Codes
- ❐ *Read* and *Write* MSTR Functions
- ❐ *Get Local Statistics*
- ❐ *Clear Local Statistics*
- ❐ *Write Global Data*
- ❐ *Read Global Data*
- ❐ *Get Remote Statistics*
- ❐ *Clear Remote Statistics*
- ❐ *Read Peer Cop Communication Health*
- ❐ Network Statistics

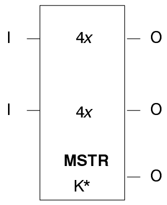
Overview

The 984-145 Compact Controller supports Modbus Plus communications. A special instruction called MSTR is provided with this controller to allow it to initiate Modbus Plus message transactions via ladder logic. An MSTR instruction allows you to initiate one of eight possible operations:

MSTR Function	Code
Write data	1
Read data	2
Get local statistics	3
Clear local statistics	4
Write global database	5
Read global database	6
Get remote statistics	7
Clear remote statistics	8
Read peer cop health	9

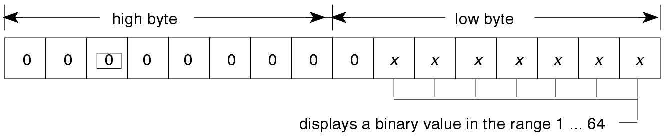
Up to four MSTR instructions may be simultaneously active in a ladder logic program. More than four MSTRs may be programmed to be enabled by the logic scan—i.e., as one active MSTR releases the resources it has been using and becomes inactive, the next MSTR encountered by the logic scan may be activated.

MSTR is a three-high nodal instruction:

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Modbus Plus master function		*Top:* ON activates the selected MSTR function *Middle:* Terminates an activates MSTR operation	*Top:* First of nine registers in the MSTR control block *Middle:* The data area** *Bottom:* Maximum number of registers in the data area	*Top:* Selected function is active *Middle:* Operation has terminated unsuccessfully *Bottom:* Operation has been completed successfully	Initiates a Modbus Plus communication function from ladder logic

* K is an integer constant in the range 1 ... 100

** For operations that provide the communications processor with data—e.g., write functions—the *data area* is the source of the data. For operations that acquire data from the communications processor—e.g., read functions—the *data area* is the destination of the data

MSTR Control Block (pointed to by the register in the top node)	
Register	**Function**
4*x*	Identifies one of the nine MSTR functions
4*x* + 1	Displays the error status in hex format (see error codes on the next page)
4*x* + 2	Displays the length (see descriptions of individual functions for specifics)
4*x* + 3	Displays function-dependent information (see descriptions of individual functions for specifics)
4*x* + 4	The *Routing 1* register, which uses the bit value of the low byte to designate the address of the destination device:  high byte low byte 0 0 0 0 0 0 0 0 0 *x* *x* *x* *x* *x* *x* *x* displays a binary value in the range 1 ... 64
4*x* + 5	The *Routing 2* register
4*x* + 6	The *Routing 3* register
4*x* + 7	The *Routing 4* register
4*x* + 8	The *Routing 5* register

MSTR Function Error Codes

If an error occurs during the execution of an MSTR function, a hexadecimal error code is displayed in register 4*x* + 1 of the MSTR control block.

The form of the code is *Mmss*, where:

- *M* is the major code
- *m* is the minor code
- *ss* is a subcode

Hex Error Code	Meaning
1001	User-initiated abort
2001	Invalid operation type
2002	User parameter changed
2003	Invalid length
2004	Invalid offset
2005	Invalid length + offset
2006	Invalid slave device data area
2007	Invalid slave device network area
2008	Invalid slave device network routing
2009	Route = your own address
200A	Attempting to acquire more global data words than are available
200B	Peer cop change on read/write global data
200C	Bad pattern for change of address request
200D	Bad address for change of address request
30*ss*	Modbus slave exception response where *ss* = 01 = Slave device does not support the requested function *ss* = 02 = Nonexistent slave device registers requested *ss* = 03 = Invalid data value requested *ss* = 04 = Unassigned *ss* = 05 = Slave has accepted long-duration program command *ss* = 06 = Function cannot be performed now—a long-duration command is in effect *ss* = 07 ... 255 = Unassigned
4001	Inconsistent Modbus slave response
5001	Inconsistent network response
6*mss*	Routing failure where the *m* subfield is an index into the routing information, indicating where where an error has been detected. A value of 0 indicates the local node, a value of 2 indicates the second device on the route, etc. And where *ss* = 01 = No response received *ss* = 02 = Program access denied *ss* = 03 = Node offline and unable to communicate *ss* = 04 = Exception response received *ss* = 05 = Router node data path busy *ss* = 06 = Slave device down *ss* = 07 = Bad destination address *ss* = 08 = Invalid node type in the routing path *ss* = 10 = Slave has rejected the command *ss* = 20 = Initiated transaction forgotten by the slave device *ss* = 40 = Unexpected master output path received *ss* = 80 = Unexpected response received
0007	Slave has rejected long-duration program command
F001	Selected option is not present

Read and *Write* MSTR Functions

An MSTR *write* function transfers data from a master source device to a specified slave destination device on the Modbus Plus network.

An MSTR *read* function transfers data from a specified slave source device on the network to the master destination device.

Read and write functions use one data master transaction path and may be completed over multiple scans.

The nine registers in the top node of the MSTR instruction contain the following information when you implement a read/ write function.

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	1 = write, 2 = read
4*x* + 1	Error status	A hex value representing an MSTR error where relevant, as shown on previous page
4*x* + 2	Length	Write = # of registers to be sent to a slave Read = # of registers to be read from a slave
4*x* + 3	Slave device data area	Specifies first register in the slave to be read or written (1 = 40001, 49 = 40049, etc.)
4*x* + 4, + 5, +6, +7, +8	Routing 1, 2, 3, 4, 5, respectively	Specifies the first through the fifth routing path addresses, respectively. The last nonzero byte in the routing path is the destination device

☞ **Note** If you attempt to program an MSTR instruction to read or write its own address, an error will be generated in the second register of the control block.

☞ **Note** It is possible to attempt a read/write operation with a nonexistent register in a slave device. The slave will detect this condition and report it as an error, but it may take multiple scans to detect it.

☞ **Note** For a full discussion of Modbus Plus routing path structures, refer to ***Modbus Plus Planning and Installation Guide*** (GM-MBPL-001).

Get Local Statistics

The MSTR *get local statistics* function obtains operational information related to the local node—i.e., the controller where the MSTR instruction has been programmed.

This function does not require a data master transaction path, and it takes one scan to complete.

The first four registers in the MSTR control block are used with this function.

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	3
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 2	Length	Starting from an offset, the # of words of statistics from the local processor's statistics table. Must be > 0 ≤ K as specified in the bottom node of the instruction
4*x* + 3	Offset	A value relative to the first available word in the local processor's statistics table—if the offset = 1, the function obtains statistics starting with the second word of the table

☞ **Note** The network statistics are given at the end of this chapter.

Clear Local Statistics

The MSTR *clear local statistics* function clears operational information related to the local node—i.e., the controller where the MSTR instruction has been programmed.

This function does not require a data master transaction path, and it takes one scan to complete.

The first two registers in the MSTR control block are used with this function.

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	4
4*x* + 1	Error status	A hex value representing an MSTR error where relevant

☞ **Note** The network statistics are given at the end of this chapter.

Write Global Data

The MSTR *write global data* function transfers data to the comm processor in the current node so that it can be sent over the network when the nodes gets the token. All nodes on the network can receive this data.

This function does not require a data master transaction path, and it takes one scan to complete.

The first three registers in the MSTR control block are used with this function.

Control Block Utilization		
Register	**MSTR Function**	**Register Content**
4*x*	Operation type	5
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 2	Length	Specifies the # of registers from the data area to be sent to the comm processor. Must be ≤ 32 and must not exceed K as specified in the bottom node of the instruction

Read Global Data

The MSTR *read global data* function gets data from the comm processor in any node node on the local network link that is providing global data.

This function does not require a data master transaction path, and it may take multiple scans to complete.

The first four registers in the MSTR control block are used with this function.

Control Block Utilization		
Register	**MSTR Function**	**Register Content**
4*x*	Operation type	6
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 2	Length	Specifies the # of words of global data to be requested from the comm processor designated by the routing path 1 parameter. Must be > 0 ≤ 32 and must not exceed K as specified in the bottom node of the instruction
4*x* + 3	Available words	Contains the # of words available from the requested node. automatically updated by the internal software.

Get Remote Statistics

The MSTR *get remote statistics* function obtains operational information related to remote nodes on the network.

This function does not require a data master transaction path, and it may take multiple scans to complete.

The nine registers in the MSTR control block are used as shown below for this function.

The remote comm processor always returns it complete statistics table when a request is made, even if the request is for less than the full table. The MSTR instruction then copies only the amount of words you have requested to the designated registers.

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	7
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 2	Length	Starting from an offset, the # of words of statistics from the remote node. Must be > 0 ≤ the total number of statistics available (54) and must not exceed the number of statistic words available
4*x* + 3	Offset	A value relative to the first available word in the statistics table—the value must not exceed the number of statistic words available
4*x* + 4, + 5, + 6, + 7, + 8	Routing 1, 2, 3, 4, 5, respectively	Specifies the first through the fifth routing path addresses, respectively. The last nonzero byte in the routing path is the destination device

Clear Remote Statistics

The MSTR *clear remote statistics* function clears operational statistics related to a remote node from the data area of the local node.

This function uses one data master transaction path, and it may take multiple scans to complete.

Seven of the registers in the MSTR control block are used for this function:

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	8
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 4, + 5, + 6, + 7, + 8	Routing 1, 2, 3, 4, 5, respectively	Specifies the first through the fifth routing path addresses, respectively. The last nonzero byte in the routing path is the destination device

☞ **Note** For a full discussion of Modbus Plus routing path structures, refer to ***Modbus Plus Planning and Installation Guide*** (GM-MBPL-001).

Read Peer Cop Communication Health

The MSTR *read peer cop communication health* function loads a specified subset of the peer cop communication health table into 4*x* registers in the controller's state RAM. This table comprises 12 words.

The first four registers in the MSTR control block are used with this function.

Control Block Utilization

Register	MSTR Function	Register Content
4*x*	Operation type	9
4*x* + 1	Error status	A hex value representing an MSTR error where relevant
4*x* + 2	# of words requested	The range is 1 ... 12
4*x* + 3	Starting word index	The range is 0 ... 11

The Peer Cop Communication Health Table

The peer cop communication health table contains 12 words, word 0 ... word 11, as shown below.

Word	Type of Health Status	for Nodes
0	Global inputs	1 ... 16
1	Global inputs	17 ... 32
2	Global inputs	33 ... 48
3	Global inputs	49 ... 64
4	Specific outputs	1 ... 16
5	Specific outputs	17 ... 32
6	Specific outputs	33 ... 48
7	Specific outputs	49 ... 64
8	Specific inputs	1 ... 16
9	Specific inputs	17 ... 32
10	Specific inputs	33 ... 48
11	Specific inputs	49 ... 64

The most significant bit of word 0 gives the health of the global input communication expected from node 16. The least significant bit on word 0 gives the health of the global input communication expected from node 1. All words in the table use this format.

The associated health bit is 0 for every null peer cop entry. A health bit is set when the node accepts inputs for the associated peer copped input data group or when it hears that another node has accepted specific output data from the associated peer copped output data group at this node. A peer cop health bit is cleared when no communication has occurred for the associated data group within the configured peer cop health timeout period.

All health bits are cleared when a START PLC command is executed. The specific input and global input health words are not valid until at least one full token rotation cycle has completed. The peer cop health bits are always valid when this peer node is not in the normal token operation state.

During the first few scans that the specific output health bits are declared invalid, the controller sets all specific output health bits. Upon initial start-up of the controller, all nodes peer copped with specific outputs have their associated health bits set to 1, meaning *healthy*. This start-up condition enables you to create ladder logic that compares the health bits without having to create special conditions during start-up, which would be the case if the values of the health bits were unknown.

Network Statistics

You can acquire the following network statistics by using the appropriate MSTR function or by using Modbus function code 8.

☞ **Note** When you use a *clear local statistics* or *clear remote statistics* function, only words 13 ... 22 are cleared.

Modbus Plus Network Statistics

Word	Byte	Meaning
00		Node type I.D. :
	0	Unknown node type
	1	Standard programmable controller node
	2	Bridge MUX
	3	Host
	4	Bridge Plus
	5	Peer I/O
01		Comm processor version (the first release was 1.00 and was displayed as 0100 hex)
02		Network address for this station
03		MAC state variable :
	0	Power-up state
	1	Monitor offline state
	2	Duplicate offline state
	3	Idle state
	4	Use token state
	5	Work response state
	6	Pass token state
	7	Solicit response state
	8	Check pass state
	9	Claim token state
	10	Claim response state
04		Peer status (LED code); provides the status of the unit relative to the network :
	0	Monitor link operation
	32	Normal link operation
	64	Never getting token
	96	Sole station
	128	Duplicate station
05		Token pass counter; increments each time the station gets the token
06		Token rotation time in ms
07	LO HI	Data master failed during token ownership bit map Program master failed during token ownership bit map
08	LO HI	Data master token owner work bit map Program master token owner work bit map
09	LO HI	Data slave token owner work bit map Program slave token owner work bit map
10	LO HI	Data master/get master response transfer request bit map Data slave/get slave command transfer request bit map
11	LO HI	Program master/get master response transfer request bit map Program slave/get slave command transfer request bit map
12	LO HI	Program master connect status bit map Program slave automatic logout request bit map

Modbus Plus Network Statistics (continued)

Word	Byte	Meaning
13	LO HI	Pretransmit deferral error counter Receive buffer DMA overrun error counter
14	LO HI	Repeated command received counter No try counter (nonexistent station)
15	LO HI	Cable A framing error Cable B framing error
16	LO HI	UART error Bad packet-length error counter
17	LO HI	Bad link address error counter Transmit buffer DMA-underrun error counter
18	LO HI	Bad internal packet length error counter Bad MAC function code error counter
19	LO HI	Communication retry counter Communication failed error counter
20	LO HI	Good receive packet success counter No response received error counter
21	LO HI	Exception response received error counter Unexpected path error counter
22	LO HI	Unexpected response error counter Forgotten transaction error counter
23	LO HI	Active station table bit map, nodes 1 ... 8 Active station table bit map, nodes 9 ... 16
24	LO HI	Active station table bit map, nodes 17 ... 24 Active station table bit map, nodes 25 ... 32
25	LO HI	Active station table bit map, nodes 33 ... 40 Active station table bit map, nodes 41 ... 48
26	LO HI	Active station table bit map, nodes 49 ... 56 Active station table bit map, nodes 57 ... 64
27	LO HI	Token station table bit map, nodes 1 ... 8 Token station table bit map, nodes 9 ... 16
28	LO HI	Token station table bit map, nodes 17 ... 24 Token station table bit map, nodes 25 ... 32
29	LO HI	Token station table bit map, nodes 33 ... 40 Token station table bit map, nodes 41 ... 48
30	LO HI	Token station table bit map, nodes 49 ... 56 Token station table bit map, nodes 57 ... 64
31	LO HI	Global data present table bit map, nodes 1 ... 8 Global data present table bit map, nodes 9 ... 16
32	LO HI	Global data present table bit map, nodes 17 ... 24 Global data present table bit map, nodes 25 ... 32
33	LO HI	Global data present table bit map, nodes 33 ... 40 Global data present table bit map, nodes 41 ... 48
34	LO HI	Global data present table bit map, nodes 49 ... 56 Global data present table bit map, nodes 57 ... 64
35	LO HI	Receive buffer in use bit map, nodes 1 ... 8 Receive buffer in use bit map, nodes 9 ... 16
36	LO HI	Receive buffer in use bit map, nodes 17 ... 24 Receive buffer in use bit map, nodes 25 ... 32
37	LO HI	Receive buffer in use bit map, nodes 33 ... 40 Station management command-processed initiation counter

Modbus Plus Network Statistics (concluded)

Word	Byte	Meaning
38	LO HI	Data master output path 1 command initiation counter Data master output path 2 command initiation counter
39	LO HI	Data master output path 3 command initiation counter Data master output path 4 command initiation counter
40	LO HI	Data master output path 5 command initiation counter Data master output path 6 command initiation counter
41	LO HI	Data master output path 7 command initiation counter Data master output path 8 command initiation counter
42	LO HI	Data slave input path 41 command processed counter Data slave input path 42 command processed counter
43	LO HI	Data slave input path 43 command processed counter Data slave input path 44 command processed counter
44	LO HI	Data slave input path 45 command processed counter Data slave input path 46 command processed counter
45	LO HI	Data slave input path 47 command processed counter Data slave input path 48 command processed counter
46	LO HI	Program master output path 81 command initiation counter Program master output path 82 command initiation counter
47	LO HI	Program master output path 83 command initiation counter Program master output path 84 command initiation counter
48	LO HI	Program master output path 85 command initiation counter Program master output path 86 command initiation counter
49	LO HI	Program master output path 87 command initiation counter Program master output path 88 command initiation counter
50	LO HI	Program slave input path C1 command processed counter Program slave input path C2 command processed counter
51	LO HI	Program slave input path C3 command processed counter Program slave input path C4 command processed counter
52	LO HI	Program slave input path C5 command processed counter Program slave input path C6 command processed counter
53	LO HI	Program slave input path C7 command processed counter Program slave input path C8 command processed counter

Chapter 9
Other Standard Instructions

- ❐ Skipping Networks
- ❐ Checking the Controller’s Health Status
- ❐ The Subroutine Instructions
- ❐ Sweep Instructions

Skipping Networks

The SKP instruction allows you to skip a specified number of networks in a ladder logic program.

When it is powered, the SKP operation is performed on every scan. The remainder of the network in which the instruction appears counts as the first of the specified number of networks to be skipped; the CPU continues to skip networks until the total number of networks skipped equals the number specified in the instruction block or until a segment boundary is reached. A SKP operation cannot cross a segment boundary.

A SKP instruction can be activated only if you specify in the controller set-up editor that skips are allowed.

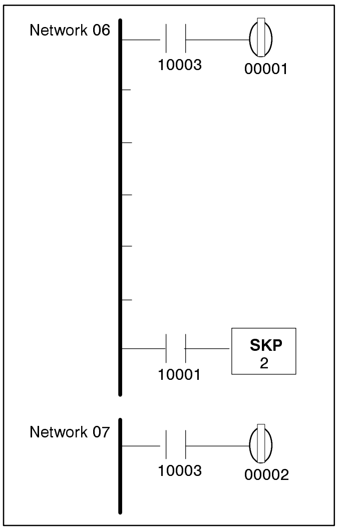
Warning If inputs and outputs that normally effect control are unintentionally skipped (or not skipped), the result can create hazardous conditions for personnel and application equipment.

SKP is a one-high nodal instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Skip logic networks	I — **SKP** 3*x*, 4*x*, *or* K*	*Top:* ON activates the skip function	*Top:* Specifies the number of logic networks to be skipped		Bypasses networks of ladder logic in the program and does not solve skipped logic
*K is an integer constant in the range 1 ... 255					

A Simple SKP Example

When contact 10001 is closed, the remainder of network 06 and all of network 07 are skipped. Power flow in the skipped networks is invalid. Coil 00001 is still controlled by contact 10003 because it is solved before the SKP.



Checking Compact Health Status

The Compact Controllers maintain a table in memory that contains vital system diagnostic information regarding the CPU, I/O, and communications. This table is 56 words long, and its contents are structured as follows:

Status Word	Content of Status Register
1 ... 11	Controller status information
12 ... 15	Health of A120 I/O modules
16 ... 181	Not used
182 ... 184	Global health and communications retry status

Each status word is 16 bits long, and the status information is conveyed by the sense of the bits in each word. The illustrations on the following pages show how the status information is presented in the status table.

The words in the status table can be accessed in ladder logic using the STAT instruction. The STAT block displays the bit patterns of the status words in a table of contiguous 4*x* registers, the values of which can then be seen in the panel software.

☞ **Note** Although you are allowed to specify either a 0*x* or 4*x* register in the top node, we recommend that you specify a 4*x* because of the excessive number of 0*x* registers that would be required to manage the status information.

The register you specify in the top node of the block is loaded with the current *word 1* bit values, and as many registers as you specify in the bottom node will be loaded with bit values from the corresponding words in the status table.

For example, if you are interested only in accessing controller status information, you could specify a register address of, say, 40701 in the top node of the block and a value of 11 in the bottom node—the bit values of the first 11 words in the status table will be loaded into registers 40701 ... 40711, respectively.

If you want to load the whole status table, specify 184 in the bottom node of the instruction. If you are not using expanded I/O, you need only specify 40 in the bottom node to get all the relevant status information.

STAT is a two-high nodal instruction.

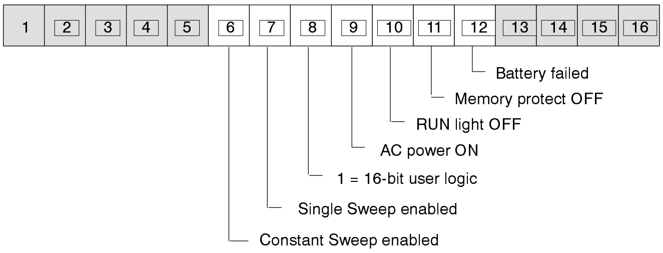
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Check CPU/ I/O Status	I — 0*x* or 4*x* — O STAT K*	*Top:* ON accesses the status table	*Top:* First word in the system status table *Bottom:* size of the status table	*Top:* operation completed	Gets status data from the status table in system memory and displays it in user registers

*K is an integer constant in the range 1 ... 184

The Compact Controller Status Table

Word 1 CPU Status

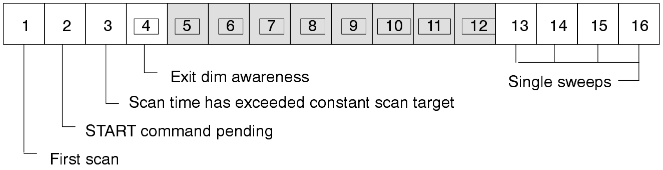
If the bit is set to 1, the condition is TRUE



Word 2 is not used

Word 3 Controller Status

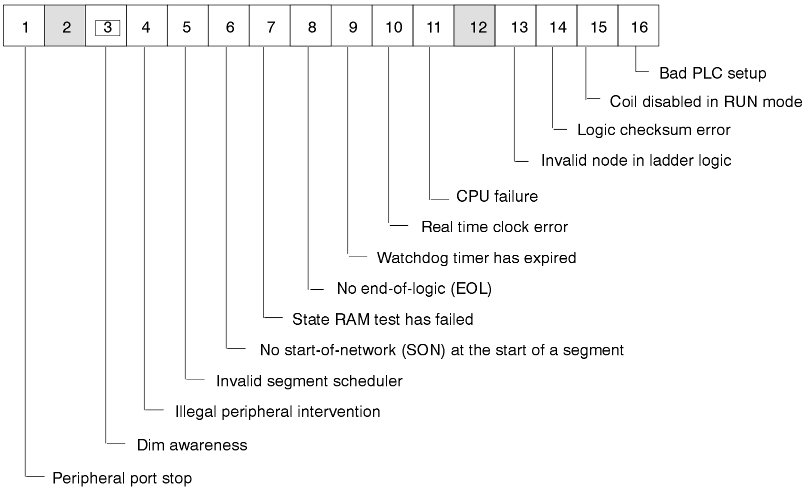
If the bit is set to 1, the condition is TRUE



Word 4 is not used

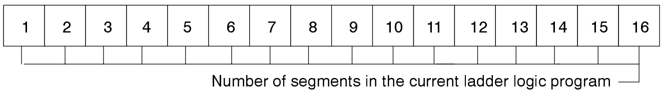
Word 5 CPU Stop State Conditions

If the bit is set to 1, the condition is TRUE

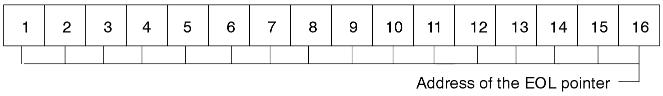

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Bad PLC setup
Coil disabled in RUN mode
Logic checksum error
Invalid node in ladder logic
CPU failure
Real time clock error
Watchdog timer has expired
No end-of-logic (EOL)
State RAM test has failed
No start-of-network (SON) at the start of a segment
Invalid segment scheduler
Illegal peripheral intervention
Dim awareness
Peripheral port stop

The Compact Controller Status Table (continued)

Word 6 Segments in Program



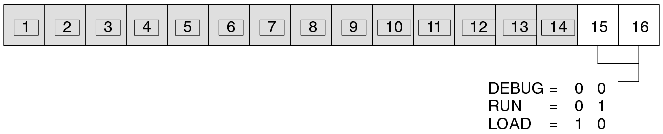
Word 7 End-0f-Logic Pointer



Word 8 is used only with the 984-145; it provides memory sizing information to the programming panel.

Word 9 is not used

Word 10 RUN/LOAD/DEBUG Status

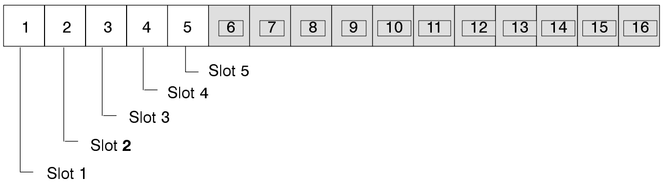


Word 11 is not used

Words 12 ... 15 are used to display the health of the A120 I/O modules in the four racks:

Word 12	Rack 1
Word 13	Rack 2
Word 14	Rack 3
Word 15	Rack 4

Each word contains the health status of up to five A120 I/O modules. The most significant (leftmost) bit represents the health of the module in slot 1 of the rack:



If an I/O module is traffic copped and ACTIVE, the bit will have a value of 1. If the module is inactive or not traffic copped, the bit will have a value of 0.

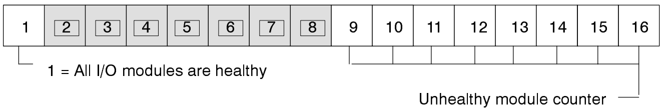
Slots 1 and 2 in rack 1 (word 12) are not used because the controller itself uses those two slots.

Words 16 ... 181 are not used

The Compact Controller Status Table (concluded)

The last three words describe health and communications on the installed A120 I/O modules

Word 182 Systemwide I/O Health Status

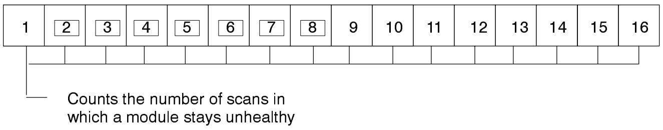


1 = All I/O modules are healthy

Unhealthy module counter

Bits 9 ... 16 are used as a counter that increments each time an unhealthy module is encountered. The counter rolls over at a count of 255.

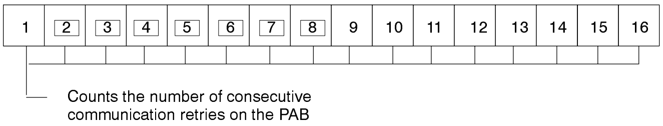
Word 183 I/O Error Count



Counts the number of scans in which a module stays unhealthy

Bits 1 ... 16 are used as a counter that increments once on each logic scan while an I/O module is unhealthy.

Word 184 PAB Bus Retry Count



Counts the number of consecutive communication retries on the PAB

Normally, all bits in this word should be 0s. Bits 1 ... 16 are used as a counter that increments once each time a comm retry occurs. If after five retries a bus error is still detected, the controller stops and displays error code 10 on the programming panel.

The Subroutine Instructions

Subroutine logic can be initiated by a program-based instruction (JSR) in the control logic. When a subroutine is initiated, the logic scan jumps to an instruction in the last segment called LAB. This instruction labels the beginning of that subroutine's logic. When the logic scan reaches an instruction in the subroutine called RET, it jumps out of that subroutine and returns to its previous position in the control logic.

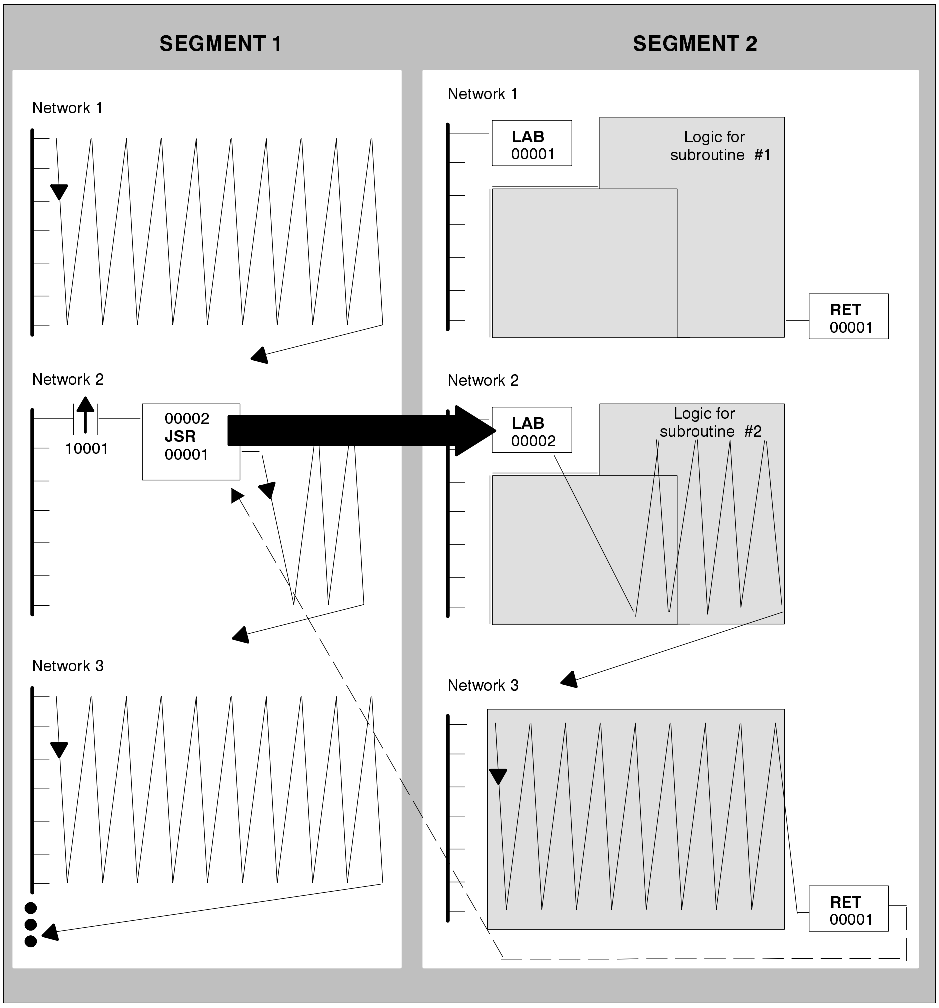
Subroutine logic is always kept in the last segment of the ladder logic program. No other logic except the subroutine logic is stored there.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Jump to a subroutine	I — 4*x* or K* — O **JSR** 00001 — O	*Top:* ON enables the source subroutine	*Top:* A constant or register value that indicates the desired subroutine *Bottom:* Always a constant value of 1	*Top:* echos the top input *Bottom:* ON if an error is detected	Causes the logic scan to jump to a specified subroutine in the last (unscheduled) segment of user logic
Label the subroutine	I — **LAB** K* — O	*Top:* ON activates the specified subroutine	*Top:* A unique constant value that identifies the selected subroutine	*Top:* ON if an error is detected	Marks the starting point of the subroutine in the user logic segment
Return to ladder logic	I — **RET** 00001 — O	*Top:* ON initiates the return out of the subfunction	*Top:* Always a constant value of 1	*Top:* ON if an error is detected	Returns the logic scan to the node immediately following the place where the subroutine was entered

*K is an integer constant in the range 1 ... 255

Below is a conceptual illustration of how a subroutine is called from ladder logic. When the logic scan in segment 1 encounters an enabled JSR instruction, it jumps to the indicated subroutine in segment 2. Only the logic associated with the called subroutine is scanned in segment 2—all other subroutine logic is ignored.

When the logic scan encounters a RET instruction in the subroutine logic, it jumps back to the node immediately following the JSR instruction in segment 1.



Sweep Instructions

Sweep functions allow you to scan logic at fixed intervals—they do not make the controller solve logic faster or terminate scans prematurely. Sweeps may be *constant* or predetermined over some fixed number of scans—i.e., *single sweeps*.

Constant sweep allows you to target your scan times from 10 ... 200 ms (in multiples of 10 ms). A target scan time is the time that elapses between the start of one scan and the start of the next. If a constant sweep is invoked with a time lapse smaller than the actual scan time, the sweep time is ignored and the system uses its normal scan rate.

The target scan time in a constant sweep encompasses logic solve time, I/O and Modbus port servicing, and system diagnostics. If you set a constant sweep target scan at 40 ms and the actual logic solve, port servicing, and diagnostics require only 30 ms, the controller will wait for 10 ms at the end of each scan before continuing to the next.

Single sweep functions allow your controller to execute a fixed number of scans—from 1 ... 15—and then to stop solving logic but continue servicing I/O. This function is useful for diagnostic work. It allows solved logic, moved data, and completed calculations to be examined for errors.

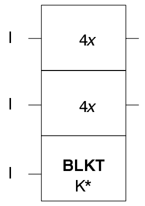
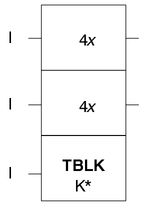
STOP **Warning Single sweeps should not be used to debug controls on machine tools, processes, or material handling systems once they have become active. Once the specified number of scans has been solved, all the outputs are frozen in their last state; since no logic solving takes place, the controller ignores all input information. This can result in unsafe, hazardous, and destructive operation of the tools or processes connected to the controller.**

Consult your programming documentation for procedures to invoke sweep instructions.

Chapter 10
Enhanced Instructions

- ❐ Block↔Table Move Instructions
- ❐ The Checksum Instruction
- ❐ The Proportional-Integral-Derivative Instruction
- ❐ Extended Math Instructions

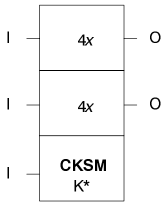
Block↔Table Move Instructions

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Block-to-table move		*Top:* ON initiates the move *Middle:* ON freezes the pointer *Bottom:* ON resets the pointer to 0	*Top:* First register in the source block *Middle:* pointer to the first register (4*x* + 1) in the destination table *Bottom:* size of the destination table	*Top:* ON when operation is completed *Middle:* Error detected—Move not possible	Moves large quantities of 4*x* registers from a fixed source block to a destination in a table
Table-to-block move		*Top:* ON initiates the move *Middle:* ON freezes the pointer *Bottom:* ON resets the pointer to 0	*Top:* First register in the source table *Middle:* pointer to the first register (4*x* + 1) in the destination block *Bottom:* size of the destination block	*Top:* ON when operation is completed *Middle:* Error detected—Move not possible	Moves a large number of contiguous registers in a table to a fixed-destination block

*K is an integer constant in the range 1 ... 100

The Checksum Instruction

The CKSM instruction is not offered as part of the standard instruction set for the 984-145 Controller. Instead, the -145 contains MSTR, which is specific to the Modbus Plus functionality of that controller.

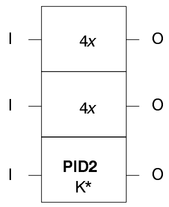
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Checksum		*Top:* ON calculates the source table cksm *Middle:* Used with bottom input to determine cksm type *Bottom:* Used with middle input to determine cksm type	*Top:* First register in the source table *Middle:* First of two registers containing the result and the implied register count *Bottom:* size of the source table	*Top:* ON when calculation is completed *Middle:* Error detected: register count = 0 or register count > size of the source table	Performs straight check, binary addition check, CRC-16 check, or LRC check, depending on state of the middle and bottom inputs (see table below)

*K is an integer constant in the range 1 ... 255

CKSM Input Usage

CKSM Calculation	Middle Input	Bottom Input
Straight check	OFF	ON
Binary addition	ON	ON
CRC-16	ON	OFF
LRC	OFF	OFF

The Proportional-Integral-Derivative Instruction

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Proportional-Integral-Deriviative		*Top:* 0 = Manual Mode 1 = Auto Mode *Middle:* 0 = Tracking ON 1 = Tracking OFF *Bottom:* 0 = output increases as E** increases 1 = output decreases as E** increases	*Top:* First of 21 registers in the source table *Middle:* First of 9 registers used by the block for calculations *Bottom:* constant representing the interval at which the calculation is performed in tenths of a second	*Top:* invalid parameter or loop active but not being solved *Middle:* PV ≥ low alarm limit*** *Bottom:* PV ≥ low alarm limit***	Implements an algorithm that performs the specified P, PI, or PID operation, as defined in registers 4x *+ 5, 4x +* 6*,* 4x *+ 7, and 4*x + 8 of the source table

* K is an integer constant in the range 1 ... 255

** E is error expressed in raw analog units

*** PV is the process variable

Block Function	Source Table Register Value			
	4*x* + 5	**4*x* + 6**	**4*x* + 7**	**4*x* + 8**
P	non-zero	zero	zero	non-zero
PI	non-zero	non-zero	zero	zero
PI	non-zero	non-zero	non-zero	zero

PID2 Source Table (Top Node)

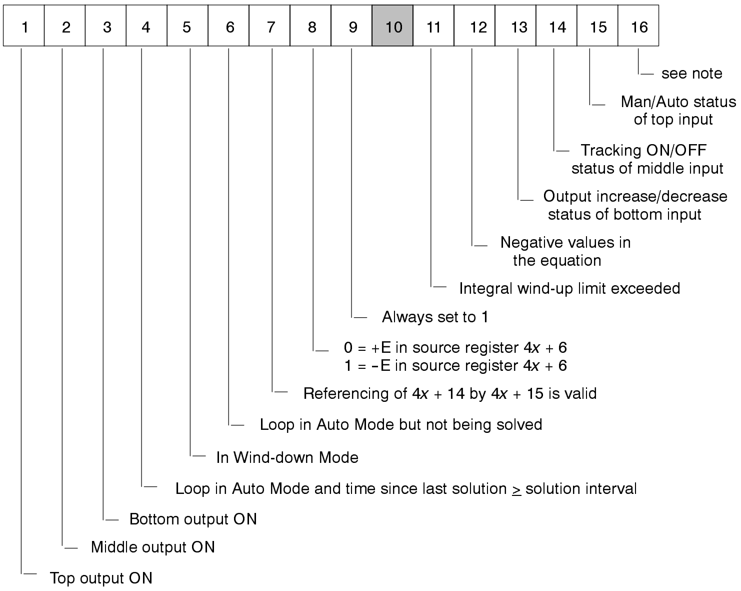
Register Number	Register Content
4*x*	**Scaled PV**: loaded by the block each time it is scanned; a linear scaling is done on register 4*x* + 13 using the high and low ranges in 4*x* + 11 and 4*x* +12: $$\text{scaled PV} = \frac{\text{reg } 4x + 13}{4095} \times (\text{reg } 4x + 11 - \text{reg } 4x + 12) + \text{reg } 4x + 12$$ Truncate the result at the decimal point and discard all digits to the right of the decimal point—do not round off the result.
4x + 1	**SP**: the set point specified in engineering units; its value must be > 4*x* + 11 > 4*x* + 12
4x + 2	**M$_V$**: loaded by the block every time the loop is solved; it is clamped to the range 0 ... 4095, making the output compatible with an analog output; the manipulated variable register may be used for furhter CPU calculations such as cascaded loops
4x + 3	**High alarm limit**: load a value into this register to specify a high alarm for PV (at or above SP); enter the value in engineering units within the range specified in registers 4*x* + 11 and 4*x* + 12
4x + 4	**Low alarm limit**: load a value into this register to specify a low alarm for PV (at or below SP); enter the value in engineering units within the range specified in registers 4*x* + 11 and 4*x* + 12
4x + 5	**Proportional band**: load this register with the desired proportional constant in the range 5 ... 500; the smaller the number, the larger the proportional contribution; a valid number is required in this register for PID2 to operate

Proportional-Integral-Derivative Instruction (continued)

PID2 Source Table (Top Node)

Register Number	Register Content
4*x* + 6	**Reset time constant**: load this register to add integral action to the calculation; the value is an integer constant in the range 0000 ... 9999, representing a range of 00.00 ... 99.99 repetitions per minute—values <9999 or >0000 stop the PID2 calculation; the larger the number, the larger the integral contribution
4*x* + 7	**Rate time constant**: load this register to add derivative action to the calculation; the value is an integer constant in the range 0000 ... 9999, representing a range of 00.00 ... 99.99 repetitions per minute—values <9999 or >0000 stop the PID2 calculation; the larger the number, the larger the derivative contribution
4*x* + 8	**Bias**: load this register to add a bias to the output—the value, which is added directly to M_V, must be between 0000 ... 4095
4*x* + 9	**High integral wind-up limit**: load this register with the upper limit of the output value (between 0 ... 4095) where the anti-reset wind-up takes place; if the specified value (normally 4095) is exceeded, the integral sum is no longer updated
4*x* + 10	**Low integral wind-up limit**: load this register with the lower limit of the output value (between 0 ... 4095) where the anti-reset wind-up takes place—the specified value is normally 0
4*x* + 11	**High engineering range**: load this register with the highest value for which the measurement device is spanned—e.g., if a resistance temperature device ranges from 0 ... 500 degrees C, the high engineering range value is 500; the high range value must be specified as a positive integer between 0001 ... 9999, corresponding to a raw analog input value of 4095
4*x* + 12	**Low engineering range**: load this register with the lowest value for which the measurement device is spanned; the low range value must be specified as a positive integer between 0001 ... 9998, corresponding to a raw analog input value of 0—it must be less than the value specified in register 4*x* + 11
4*x* + 13	**Raw analog measurement**: the logic program loads this register with PV; the measurement must be scaled and linear in the range 0 ... 4095
4*x* + 14	**Pointer to loop counter register**: the value you load in this register points to the register that counts the number of loops solved in each scan; the value entered in the register is the reference number of the register where the loop count is kept—e.g., if register 41236 keeps the count, enter the value 1236 in register 4*x* + 14 of the PID2 source table; the same value must be loaded to the 4*x* + 14 register in the source table of every PID2 block in a logic program
4*x* + 15	**Maximum number of loops/scan**: if register 4*x* = 14 contains a non-zero value, you may load a value into this register to specify the limit on the number of loops to be solved in a single scan
4*x* + 16	**Pointer to reset feedback input**: the value you load in this register points to the holding register that contains the feedback value (F); integration calculations rely on the F value being connected to M_V—as the PID2 output varies from 0 ... 4095, so should F vary from 0 ... 4095; the value entered in the register is the feedback register reference number—e.g., if the feedback register is 42250, enter the value 2250 in register 4*x* + 16 of the PID2 source table
4*x* + 17	**Output clamp high**: the value entered in this register determines the upper limit of M_V (normally 4095)
4*x* + 18	**Output clamp low**: the value entered in this register determines the lower limit of M_V (normally 0)
4*x* + 19	**RGL constant**: the *rate gain limit* value entered in this register determines the effective degree of derivative filtering; the range for this value is from 2 ... 30; the smaller the value, the more filtering takes place
4*x* + 20	**Pointer to track input**: the value entered in this register points to the holding register containing the track input (T) value; the T value is connected to the input of the integral lag whenever the auto bit and track bit are both TRUE; the value entered in this register is the track input register reference number—e.g., if the track input register is 40956, enter 0956 in register 4*x* + 20 in the PID2 source table

Proportional-Integral-Derivative Instruction (continued)

PID2 Calculation Block (Middle Node)	
Register Number	**Register Content**
4x	Loop status register  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 see note Man/Auto status of top input Tracking ON/OFF status of middle input Output increase/decrease status of bottom input Negative values in the equation Integral wind-up limit exceeded Always set to 1 0 = +E in source register 4*x* + 6 1 = −E in source register 4*x* + 6 Referencing of 4*x* + 14 by 4*x* + 15 is valid Loop in Auto Mode but not being solved In Wind-down Mode Loop in Auto Mode and time since last solution ≥ solution interval Bottom output ON Middle output ON Top output ON
	Note: Bit 16 is set after initial start-up or installation of the loop. If the bit is cleared, the following actions all take place in one scan: The loop status register is rest The current value in the real-time clock is stored in register 4*x* + 1 in this block Registers 4*x* + 3, 4*x* + 4, and 4*x* + 5 in this block are set to zero The value in source table register 4*x* + 13 is multiplied by 8 and stored in register 4*x* + 6 of this block Register 4*x* + 7 and 4*x* + 8 in this block are cleared

Proportional-Integral-Derivative Instruction (continued)

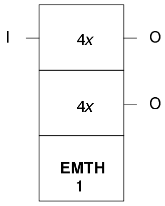
<table>
<tr><th colspan="2">PID2 Calculation Block (Middle Node)</th></tr>
<tr><th>Register Number</th><th>Register Content</th></tr>
<tr><td>4x + 1</td><td>Error (E) status
<table>
<tr><th>Bit Code</th><th>Meaning</th><th>Check This Register in the Source Table (Top Node)</th></tr>
<tr><td>0000</td><td>No errors, all validations OK</td><td></td></tr>
<tr><td>0001</td><td>Scaled SP above 9999</td><td>4x + 1</td></tr>
<tr><td>0002</td><td>High alarm above 9999</td><td>4x + 3</td></tr>
<tr><td>0003</td><td>Low alarm above 9999</td><td>4x + 4</td></tr>
<tr><td>0004</td><td>Proportional band below 5</td><td>4x + 5</td></tr>
<tr><td>0005</td><td>Proportional band above 500</td><td>4x + 5</td></tr>
<tr><td>0006</td><td>Reset above 99.99 repeats/min</td><td>4x + 6</td></tr>
<tr><td>0007</td><td>Rate above 99.99 min</td><td>4x + 7</td></tr>
<tr><td>0008</td><td>Bias above 4095</td><td>4x + 8</td></tr>
<tr><td>0009</td><td>High integral limit above 4095</td><td>4x + 9</td></tr>
<tr><td>0010</td><td>Low integral limit above 4095</td><td>4x + 10</td></tr>
<tr><td>0011</td><td>High engineering unit scale above 9999</td><td>4x + 11</td></tr>
<tr><td>0012</td><td>Low engineering unit scale above 9999</td><td>4x + 12</td></tr>
<tr><td>0013</td><td>High engineering unit scale below low engineering unit</td><td>4x + 11 and 4x + 12</td></tr>
<tr><td>0014</td><td>Scaled SP above high engineering unit</td><td>4x + 1 and 4x + 11</td></tr>
<tr><td>0015</td><td>Scaled SP below low engineering unit</td><td>4x + 1 and 4x + 11</td></tr>
<tr><td>0016</td><td>Loops/scan > 9999</td><td>(4x + 15 = 0)</td></tr>
<tr><td>0017</td><td>Reset feedback pointer out of range</td><td>4x + 16</td></tr>
<tr><td>0018</td><td>High output clamp above 4095</td><td>4x + 17</td></tr>
<tr><td>0019</td><td>Low output clamp above 4095</td><td>4x + 18</td></tr>
<tr><td>0020</td><td>Low output clamp above high output clamp</td><td>4x + 17 and 4x + 18</td></tr>
<tr><td>0021</td><td>RGL below 2</td><td>4x + 19</td></tr>
<tr><td>0022</td><td>RGL above 30</td><td>4x + 19</td></tr>
<tr><td>0023</td><td>Track F pointer out of range</td><td>4x + 20 and middle input ON</td></tr>
<tr><td>0024</td><td>Track F pointer is zero</td><td>4x + 20 and middle input ON</td></tr>
<tr><td>0025</td><td>Node locked out (short of scan time)</td><td>see note below</td></tr>
<tr><td>0026</td><td>Loop counter pointer is zero</td><td>4x + 14 and 4x + 15</td></tr>
<tr><td>0024</td><td>Loop counter pointer out of range</td><td>4x + 14 and 4x + 15</td></tr>
</table>
<b>Note:</b> If lockout occurs often and all the parameters are valid, increase the maximum allowable number of loops/scan. Lockout may also occur if the counting registers in use are not cleared as required.</td></tr>
<tr><td>4x + 2</td><td><b>Loop timer register</b>: stores the real-time clock reading on the system clock each time the loop is solved; the difference between the current clock value and the value stored in this register is the elapsed time; if elapsed time ≥ the solution interval (10 times the value given in the bottom node of the PID2 block), the loop should be solved in the current scan</td></tr>
<tr><td>4x + 3
4x + 4
4x + 5</td><td>Reserved for internal use</td></tr>
</table>

Proportional-Integral-Derivative Instruction (concluded)

PID2 Calculation Block (Middle Node)

Register Number	Register Content
4x + 6	P_v x 8 (filtered): stores the result of the filtered analog input (from source register 4*x* + 14) multiplied by eight; this value is useful in derivative control operations
4x + 7	**Absolute value of E**: contains the absolute value of SP – PV; bit 8 in register 4*x* + 1 of this block indicates the sign of E; the value in this register is updated after each loop solution
4x + 8	Reserved for internal use

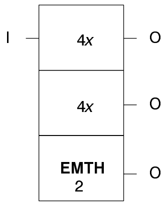
Extended Math Instructions

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Double precision (32-bit) addition		*Top:* ON initiates the double precision addition	*Top:* First of two contiguous registers containing operand 1—its value is in the range 0 ... 99,999,999 *Middle:* First of six registers in the block described below *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle:* an operand is invalid or out of range	Adds operand 1 (the value in the top node register block) and operand 2 (the value in the first two registers of the middle node block), then places the result in the fourth and fifth registers of the middle node block

Middle Node Block

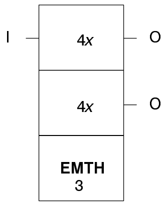
Register Number	Register Content
4*x* and 4*x* + 1	the value of operand 2, in the range 0 ... 99,999,999
4*x* + 2	a non-zero value indicates that an overflow condition exists
4*x* + 3 and 4*x* + 4	the result of the double precision addition
4*x* + 5	not used but must be configured

Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Double precision (32-bit) subtraction		*Top:* ON initiates the double precision subtraction	*Top:* First of two contiguous registers containing operand 1—its value is in the range 0 ... 99,999,999 *Middle:* First of six registers in the block described below *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle:* operand$_1$ = operand$_2$ *Bottom:* operand$_1$ < operand$_2$	Subtracts operand 2 (the value in the first and second registers in the middle node block) from operand 1 (the value in the top node block), then places the result in the third and fourth registers of the middle node block

Middle Node Block

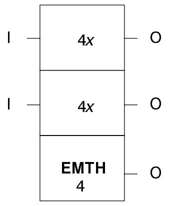
Register Number	Register Content
4*x* and 4*x* + 1	the value of operand 2, in the range 0 ... 99,999,999
4*x* + 2 and 4*x* + 3	the result of the double precision subtraction
4*x* + 4	non-zero value indicates that an out-of-range condition exists
4*x* + 5	not used but must be configured

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Double precision multiplication		*Top:* ON initiates the double precision multiplication	*Top:* First of two contiguous registers containing operand 1, whose value is in the range 0 ... 99,999,999 *Middle:* First of six registers in the block described below *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle:* an operand is out of range	Multiplies operand 1 (the value in the top node register block) by operand 2 (the value in the first two registers of the middle node block), then places the result in the third, fourth, fifth, and sixth registers of the middle node block

Middle Node Block

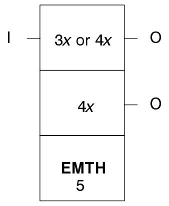
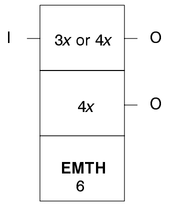
Register Number	Register Content
4*x* and 4*x* + 1	the value of operand 2, in the range 0 ... 99,999,999
4*x* + 2, 4*x* + 3, 4*x* + 4, and 4*x* + 5	the result of the double precision multiplication

Extended Math Instructions (continued)

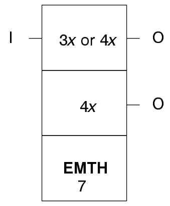
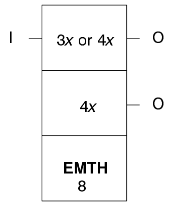
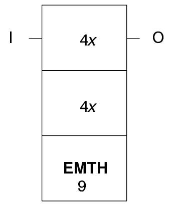
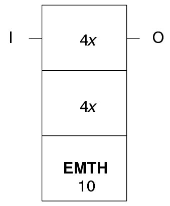
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Double precision division		*Top:* ON initiates the double precision division *Middle*: ON = remainder is stored as a fraction OFF = remainder is stored as a whole number	*Top:* First of two contiguous registers containing operand 1—its value is in the range 0 ... 99,999,999 *Middle:* First of six registers in the block described below *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle*: an operand is out of range *Bottom*: operand 2 = 0	Divides operand 1 (the value in the top node register block) by operand 2 (the first two registers in the middle node block), then places the result in the third and fourth registers of the middle node block and the remainder in the fifth and sixth registers of the middle node block

Middle Node Block

Register Number	Register Content
4*x* and 4*x* + 1	the value of operand 2, in the range 0 ... 99,999,999
4*x* + 2 and 4*x* + 3	the result (quotient) of the double precision division
4*x* + 4 and 4*x* + 5	the remainder of the double precision division

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Square root		*Top:* ON initiates the √ operation	*Top:* First of two registers containing a source value in the range 0 ... 99,999,999 *Middle:* First of two registers where the result is stored in the fixed-decimal format: 1234.5600 *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle*: source value is out of range	Calculates the square root of the source value in the top node registers and stores the result in the middle node registers
Process square root		*Top:* ON initiates the √ operation	*Top:* First of two registers containing a source value in the range 0 ... 99,999,999 *Middle:* First of two registers where the *linearized* result is stored *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle*: source value is out of range	Calculates the square root of the source value in the top node registers, linearizes it by multiplying it by 63.9922 (the square root of 4095), then stores the linearized result in the middle node registers Process square roots are often used in PID2 operations

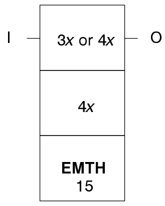
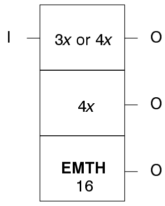
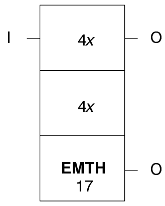
Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Logarithm		*Top:* ON initiates a logarithmic operation	*Top:* First of two contiguous registers containing a source value in the range 0 ... 99,999,999 *Middle:* A holding register where the result is stored *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle*: an error has been detected or a value is out of range	Performs a base 10 logarithmic operation on the value in the source registers in the top node, then stores the result in the middle-node register
Antilogarithm		*Top:* ON initiates a logarithmic operation	*Top:* A single register that contains a source value stored in the fixed decimal format 1.234 and in the range 0 ... 7.999 *Middle:* First of two contiguous registers where the result is stored *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle*: an error has been detected or a value is out of range	Performs a base 10 antilogarithmic operation on the value in the source register and stores the result in the middle-node registers in the fixed-decimal format: 12345678
Integer-to-floating point conversion		*Top:* ON initiates the conversion	*Top:* First of two contiguous registers containing a double-precision integer source value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Converts a double-precision integer value into a 32-bit floating point value and stores the result in the third and fourth registers of the middle-node block The first two registers in the block are not used*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 9 instruction.			
Integer + floating point addition		*Top:* ON initiates the addition	*Top:* First of two contiguous registers containing a double-precision integer value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Adds the double-precision integer value in the top- node register block and the FP value in the first two registers in the middle-node block then stores the result in the third and fourth registers of the middle-node block

Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
Integer - floating point subtraction	I — 3*x* or 4*x* — O 4*x* **EMTH** 11	*Top:* ON initiates the subtraction	*Top:* First of two contiguous registers containing a double-precision integer value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Subtracts the FP value in the first two registers of the middle-node block from the integer value in the top-node register block then stores the result in the third and fourth registers of the middle-node block
Integer x floating point multiplication	I — 3*x* or 4*x* — O 4*x* **EMTH** 12	*Top:* ON initiates the multiplication	*Top:* First of two contiguous registers containing a double-precision integer value *Middle:* First in a block of four contiguous registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Multiplies the double-precision integer value in the top-node register block by the FP value in the first two registers of the middle-node block, then stores the product in the third and fourth registers of the middle-node block
Integer/floating point division	I — 4*x* — O 4*x* **EMTH** 13	*Top:* ON initiates the division	*Top:* First of two contiguous registers containing a double-precision integer value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Divides the double-precision integer value in the top-node register block by the FP value in the first two registers of the middle-node block, then stores the quotient in the third and fourth registers of the middle-node block
floating point - integer subtraction	I — 4*x* — O 4*x* **EMTH** 14	*Top:* ON initiates the subtraction	*Top:* First of two contiguous registers containing a floating point value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Subtracts the double-precision integer value in the first two registers of the middle-node block from the FP value in the top-node register block, then stores the result in the third and fourth registers of the middle-node block

Extended Math Instructions (continued)

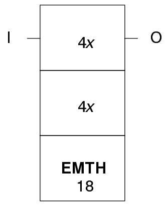
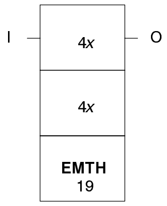
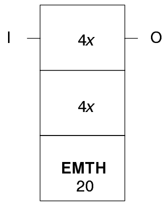
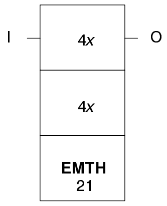
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point/ integer division		*Top:* ON initiates the division	*Top:* First of two contiguous registers containing a floating point value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Divides the double-precision integer value in the first two registers of the middle-node block by the FP value in the top-node register block, then stores the quotient in the third and fourth registers of the middle-node block
Integer-floating point comparison		*Top:* ON initiates the comparison	*Top:* First of two contiguous registers containing a double-precision integer value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle:* used with the bottom output to indicate the value relationship *Bottom:* used with the middle output to indicate the value relationship	Compares the double-precision integer value with the floating point value (in the first two registers of the middle-node block), then indicates the relationship via the middle and bottom outputs (see table below) The third and fourth registers in the middle-node block are not used but must be configured
floating point-to-integer conversion		*Top:* ON initiates the conversion	*Top:* First of two contiguous registers containing a double-precision integer *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Bottom:* 0 = + integer value 1 = – integer value	Converts the FP value stored in the third and fourth registers of the middle-node block into a double-precision integer value and stores the converted value in the top-node registers The first and second registers in the middle node are not used but must be configured*

EMTH 16 Outputs

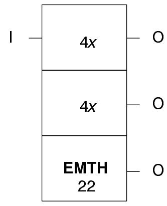
Middle Output State	Bottom Output State	Value Relationship
ON	OFF	I > FP
OFF	ON	I < FP
ON	ON	I = FP

* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 17 instruction.

Extended Math Instructions (continued)

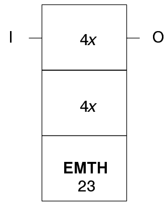
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point addition		*Top:* ON initiates the subtraction	*Top:* First of two contiguous registers containing FP value 1 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Adds FP value 1 (in the top-node register block) and FP value 2 (from the first two registers of the middle-node block), then stores the sum in the third and fourth registers of the middle-node block
floating point subtraction		*Top:* ON initiates the multiplication	*Top:* First of two contiguous registers containing FP value 1 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Subtracts FP value 2 (stored in the first and second registers of the middle-node block) from FP value 1 (in the top-node register block), then stores the difference in the third and fourth registers of the middle-node block
floating point multiplication		*Top:* ON initiates the division	*Top:* First of two contiguous registers containing FP value 1 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Multiplies FP value 1 (in the top-node register block) by FP value 2 (stored in the first and second registers of the middle-node block), then stores the product in the third and fourth registers of the middle-node block
floating point division		*Top:* ON initiates the subtraction	*Top:* First of two contiguous registers containing FP value 1 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Divides FP value 1 (in the top-node register block) by FP value 2 (stored in the first and second registers of the middle-node block), then stores the quotient in the third and fourth registers of the middle-node block

Extended Math Instructions (continued)

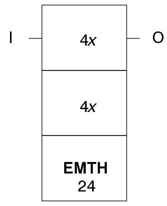
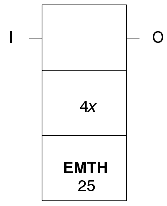
Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point comparison		*Top:* ON initiates the comparison	*Top:* First of two contiguous registers containing FP value 1 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when comparison is complete *Middle:* used with the bottom output to indicate the value relationship *Bottom:* used with the middle output to indicate the value relationship	Compares FP value 1 (in the top-node register block) and FP value 2 (in the first two registers of the middle-node block), then indicates the relationship via the middle and bottom outputs (see table below) The third and fourth registers in the middle node block are not used but must be configured

EMTH 22 Outputs

Middle Output State	Bottom Output State	Value Relationship
ON	OFF	FP value 1 > FP value 2
OFF	ON	FP value 1 < FP value 2
ON	ON	FP value 1 = FP value 2

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point square root		*Top:* ON initiates the √ operation	*Top:* First of two contiguous registers containing an FP value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Performs a square root operation on the FP value in the top-node block and stores the result in the third and fourth registers of the middle-node block. The first and second registers in the middle-node block are not used but must be configured*

*** Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 23 instruction.

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point sign change		*Top:* ON initiates the sign change operation	*Top:* First of two registers containing an FP value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when operation is completed	Changes the sign of the FP value in the top-node register block and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used
floating point π loading		*Top:* ON loads π into the middle-register block	*Top*: Not used *Middle:* First of four registers where the FP value of pi is loaded *Bottom:* appropriate EMTH function code	*Top:* ON when loading is completed	Loads the FP value of pi into the third and fourth registers of the middle-node block; the first and second registers of the middle-node block are not used

Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point sine of an angle	I — 4x — O 4x **EMTH** 26	*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing the FP value of an angle in radians; the magnitude is < 65536.0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radials the sine of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 26 instruction.			
floating point cosine of an angle	I — 4x — O 4x **EMTH** 27	*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing the FP value of an angle in radians; the magnitude is < 65536.0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radians the cosine of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 27 instruction.			
floating point tangent of an angle	I — 4x — O 4x **EMTH** 28	*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing the FP value of an angle in radians; the magnitude is < 65536.0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radians the tangent of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 28 instruction.			

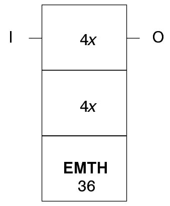
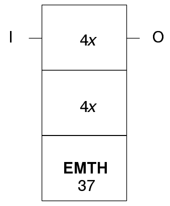
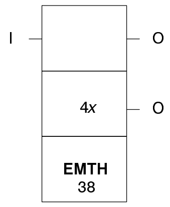
Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point arcsine of an angle	I — 4x — O 4x EMTH 29	*Top:* ON initiates the calculation	*Top:* First of two registers containing the FP value of the sine of an angle between -π / 2 ... π / 2 radians; the value must be in the range -1.0 ... +1.0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radians the arcsine of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block;. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 29 instruction.			
floating point arc cosine of an angle	I — 4x — O 4x EMTH 30	*Top:* ON initiates the calculation	*Top:* First of two registers containing the FP value of the cosine of an angle between 0 ... π radians; in the range of -1.0 ... +1.0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radians the arc cosine of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers in the middle-node block are not used but must be configured*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 30 instruction.			
floating point arctangent of an angle	I — 4x — O 4x EMTH 31	*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing the FP value of the tangent of an angle between -π/2 ... π/2 radians *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates in radians the arctangent of the floating point value in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 31 instruction.			

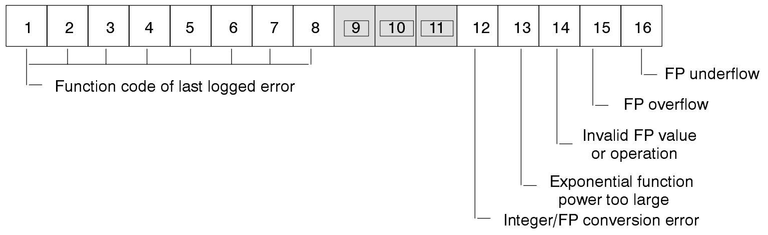
Extended Math Instructions (continued)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point radian-to-degree conversion	I — 4x — O 4x EMTH 32	*Top:* ON initiates the conversion	*Top:* First of two contiguous registers containing the FP value of an angle in radians *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when conversion is completed	Converts the FP value in the top-node registers to an FP representation of that value in radians, and stores the conversion in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		***Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 32 instruction.			
floating point degree-to-radian conversion	I — 4x — O 4x EMTH 33	*Top:* ON initiates the conversion	*Top:* First of two contiguous registers containing the FP value of an angle in degrees *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when conversion is completed	Converts the FP value in the top-node registers to an FP representation of that value in degrees, and stores the converted value in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		***Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 33 instruction.			
floating point number raised to an integer power	I — 4x — O 4x EMTH 34	*Top:* ON initiates the calculation	*Top:* First of two registers containing an FP value *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Raises the FP value in the top-node registers to the integer power specified in the second register of the middle-node block, and stores the result in the third and fourth registers of the middle-node block; the first register in the middle node must be set to zero
floating point exponential	I — 4x — O 4x EMTH 35	*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing an FP value in the range −87.34 ... +88.72 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates the exponential value of the FP number in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		***Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 35 instruction.			

Extended Math Instructions (concluded)

Instruction	Structure	Inputs (I)	Nodes	Outputs (O)	Function
floating point natural logarithm		*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing an FP value > 0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates the natural logarithm of the FP value in the top-node registers and stores the result in the third and fourth registers of the middle-node block The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 36 instruction.			
floating point common logarithm		*Top:* ON initiates the calculation	*Top:* First of two contiguous registers containing an FP value > 0 *Middle:* First in a block of four contiguous holding registers *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed	Calculates the common logarithm of the FP number in the top-node registers and stores the result in the third and fourth registers of the middle-node block. The first and second registers of the middle-node block are not used but must be configured.*
		* **Note** If you want to preserve registers, you may store the double-precision integer value in the first and second registers of the middle-node block and not configure a top-node register block in the EMTH 37 instruction.			
Error report log		*Top:* ON initiates the calculation	*Top:* Not used *Middle:* First of four registers that contain the error log data (see below) *Bottom:* appropriate EMTH function code	*Top:* ON when calculation is completed *Middle:* 1 = nonzeros in the register 0 = all bits set to zero	Error data are logged in the third register of the middle-node block, and the fourth register is always set to zero The first and second registers in the middle-node block are not used, but must be configured.

Register 4*x* + 2 in the Middle Node of EMTH 38


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
FP underflow
FP overflow
Invalid FP value or operation
Exponential function power too large
Integer/FP conversion error
Function code of last logged error