Modicon 984 Programmable Controller Systems Manual

GM-0984-SYS Rev. B

May, 1991

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

Table of Contents

Chapter 1 The 984 Programmable Controllers 1

Chapter 2 Optional and Peripheral Control Devices . . . 15

Chapter 3 984 I/O Subsystems 29

Chapter 4 984 Communications Capabilities 51

Chapter 5 984 Memory Allocation 71

Chapter 6 984 Opcode Assignments 83

Chapter 7 Ladder Logic Overview 95

Chapter 8 Contacts, Shorts, and Coils 119

Chapter 9 Counters and Timers 127

Chapter 10 Standard Calculate Functions 133

Chapter 11 DX Move Functions 141

Chapter 12 DX Matrix Functions 155

Chapter 13 ASCII READ/WRITE Functions 169

Chapter 14 Monitoring System Status 179

Chapter 15 Bypassing Networks with SKP 205

Chapter 16 Extended Memory Capabilities 209

Chapter 17 Modbus Plus Master Function 217

Preface

The following are trademarks of Modicon, Inc.:

Modbus
Modbus Plus
Modbus II
984
984A
984B
984X
984-785
984-780
984-685
984-680
984-485
984-480
984-385
984-381
984-380
Micro-984
984-120
984-130
984-145
Compact 984
AT-984
MC-984
Q984
P190
P230
BP85
SM85
SA85

MODSOFT® is a registered trademark of Modicon, Inc.

IBM® is a registered trademark of International Business Machines Corporation. IBM AT™, IBM XT™, Micro Channel™, Personal System/2™, and NetBIOS™ are trademarks of International Business Machines Corporation.

Microsoft® and MS-DOS® are registered trademarks of Microsoft Corporation.

Chapter 1
The 984 Programmable Controllers

- ❐ Modicon's Family of 984 Programmable Controllers
- ❐ 984 Controller Performance and Capacity Characteristics
- ❐ How a 984 System Provides Application Control
- ❐ P190-Style Panel Software Support
- ❐ MODSOFT Panel Software Support
- ❐ Overview of the 984 Instruction Set

1.1 Modicon's Family of Programmable Controllers

Modicon offers a wide range of compact, midsize, and high-performance CPUs with its 984 family of programmable controllers. All 984 controllers, regardless of their particular hardware implementation, use a common processing architecture; they are all programmed with ladder logic, a powerful and graphical language that emulates relay-equivalent symbology; and they share common instructions drawn from a large set of calculation, data transfer (DX), matrix, and special-application functions. Modicon also provides you with various networking strategies, allowing you to interconnect multiple controllers—and other devices—for increased application control and data exchange.

1.1.1 The 984 Family

984 controllers are available in four generic hardware classes:

- ☐ Large, rugged, high-performance *chassis mount* controllers
- ☐ Rugged, midrange-performance *slot mount* controllers, which reside in a primary housing beside 800 Series I/O modules
- ☐ *Host-based* controllers built on various industry-standard computer cards designed to reside in and execute control logic from a host computer
- ☐ Low-cost, easy-to-install *compact* controllers, for applications with less demanding environmental and performance requirements

The family approach to 984 controller design allows you to make choices based on controller *capacity* (the number of discrete and analog/register points available for application programming, the number of I/O drops it supports), *throughput* (the rate at which it solves logic and updates I/O modules), and *environmental hardness* (the design standards its hardware implementation must meet).

1.1.2 Controller Compatibility

A major advantage of the family approach to 984 controller design is *product compatibility*. Regardless of its computational capacity, performance characteristics, or hardware implementation, each 984 controller is architecturally consistent with other 984s.

The 984 instruction set (the functional capabilities of the controller, part of the system firmware stored in executive PROM) comprises logic functions common to other 984s. This means that user logic created on a midrange or high-performance unit such as a 984-685 or a 984B can be relocated to a smaller controller such as a 984-145 (assuming sufficient memory in the smaller machine) and that logic created on a smaller controller is upwardly compatible to a larger unit. As your application requirements increase, it is relatively easy to upgrade your controller hardware without having to rewrite control logic.

Also, training costs and learning curves can be reduced, since users familiar with one 984 model automatically have a strong understanding of others.

1.2 984 Controller Performance and Capacity Characteristics

The table on the following page gives you an overview of 984 programmable controller characteristics. The 984 controller models are listed by capacity in descending order, the 24 bit CPUs first, followed by the 16 bit CPUs. The capacity of a controller is a function of the number of discrete and register points available in state RAM—a discrete point uses one bit while a register/analog point requires 16 bits.

Notice that the discretes and registers are implemented in two different areas of system memory—in state RAM and in real-world I/O locations as defined by the 984 *traffic cop*. The registers and discretes available in state RAM may be used for programming I/O, internal coils, and data registers; the registers and discretes available through the traffic cop can be used only for programming local or remote I/O points. In some of the smaller-cpacity controllers, the traffic cop limits the maximum number of I/O bits and the total number of discrete I/O points to numbers below what is available in state RAM. The additional discretes and registers from state RAM may be used in the logic program for internal coils and data storage buffers, but they cannot be mapped to I/O points.

984 Programmable Controller Performance and Capacity Characteristics

984 Model	Hardware Implementation	Logic Solve (ms/Kword)	CPU Size	User Logic Size**	State RAM		Maximum I/O Bits per Drop	Maximum I/O Bits/System	Total Discrete I/O	Max. Drops per System
					Regs	Discretes				
984B	Chassis mount	0.75	24 bits	32K/64K*	9999	8192***	1024 in/1024 out 256 in/256 out	32768 in/32768 out 4096 in/4096 out	8192 in/8192 out 4096 in/4096 out	32 R (S908) 16 R (S901)
984-780/-785	Slot mount	1.5	24 bits	16K/32K	9999	8192***	512 in/512 out	16384 in/16384 out	8192 in/8192 out	1 L, 31 R
Q984	Host Based	2.0	24 bits	12K	9999	8192***	512 in/512 out	3584 in/3584 out	3584 any mix	7 R
984A	Chassis mount	0.75	16 bits	16K/32K	1920	2048 any mix	1024 in/1024 out 256 in/256 out	32768 any mix 4096 in/4096 out	2048 any mix 2048 any mix	32 R (S908) 16 R (S901)
984X	Chassis mount	0.75	16 bits	8K	1920	2048 any mix	512 in/512 out	3584 in/3584 out	2048 any mix	1 L, 6 R
984-685	Slot mount	2.0	16 bits	8K/16K	4096****	2048 any mix	512 in/512 out	16384 in/16384 out	2048 any mix	1 L, 31 R
984-680	Slot mount	3.0	16 bits	8K/16K	4096****	2048 any mix	512 in/512 out	16384 in/16384 out	2048 any mix	1 L, 31 R
AT-984	Host Based	1.5	16 bits	8K	1920	2048 any mix	512 in/512 out	3584 in/3584 out	2048 any mix	7 R
MC-984	Host Based	1.5	16 bits	8K	1920	2048 any mix	512 in/512 out	3584 in/3584 out	2048 any mix	7 R
984-485	Slot mount	3.0	16 bits	4K/8K	1920	2048 any mix	512 in/512 out	3584 in/3584 out	1024 any mix	1 L, 6 R
984-480	Slot mount	5.0	16 bits	4K/8K	1920	2048 any mix	512 in/512 out	3584 in/3584 out	1024 any mix	1 L, 6 R
984-385	Slot mount	3.0	16 bits	4K/6K	1920	2048 any mix	512 in/512 out	512 in/512 out	512 any mix	1 L
984-381	Slot mount	5.0	16 bits	1.5K/4K/6K	1920	2048 any mix	512 in/512 out	512 in/512 out	512 any mix	1 L
984-380	Slot mount	5.0	16 bits	1.5K/4K/6K	1920	2048 any mix	512 in/512 out	512 in/512 out	256 any mix	1 L
984-145	Compact	4.25	16 bits	8K	1920	2048 any mix	512 in/512 out	512 in/512 out	256 any mix	1 L
984-130	Compact	4.25	16 bits	4K	1920	2048 any mix	512 in/512 out	512 in/512 out	256 any mix	1 L
984-120	Compact	4.25	16 bits	1.5K	1920	2048 any mix	512 in/512 out	512 in/512 out	256 any mix	1 L
Micro-984	Micro	5.0	16 bits	4K	1920	2048 any mix	64 in/64 out (112 total)	64 in/64 out (112 total)	112 any mix	1 L

R = Remote, L = Local

* The 984B offers extended memory (XMEM) in 32K, 64K, and 96K sizes; total memory can be up to 128K, with up to 64K devoted to user logic (UL):

- 32K = 32K UL
- 64K = 64K UL or 32K UL/32K XMEM
- 96K = 32K UL/64K XMEM or 64K UL/32K XMEM
- 128K = 32K UL/96K XMEM or 64K UL/64K XMEM

** Approximately 1K words of user logic are used for system overhead; utilizes one word/node for user logic—e.g., a normally open contact uses one word of user logic memory.

*** State RAM in these 24 bit CPUs may be allocated as 8192 discrete I/O + 9999 registers or as (8192 discrete in/8192 discrete out + 8500 registers).

**** 4096 registers are available if you use an Extended Register cartridge (AS-E685-914 or AS-E680-914); otherwise, 1920 registers are available.

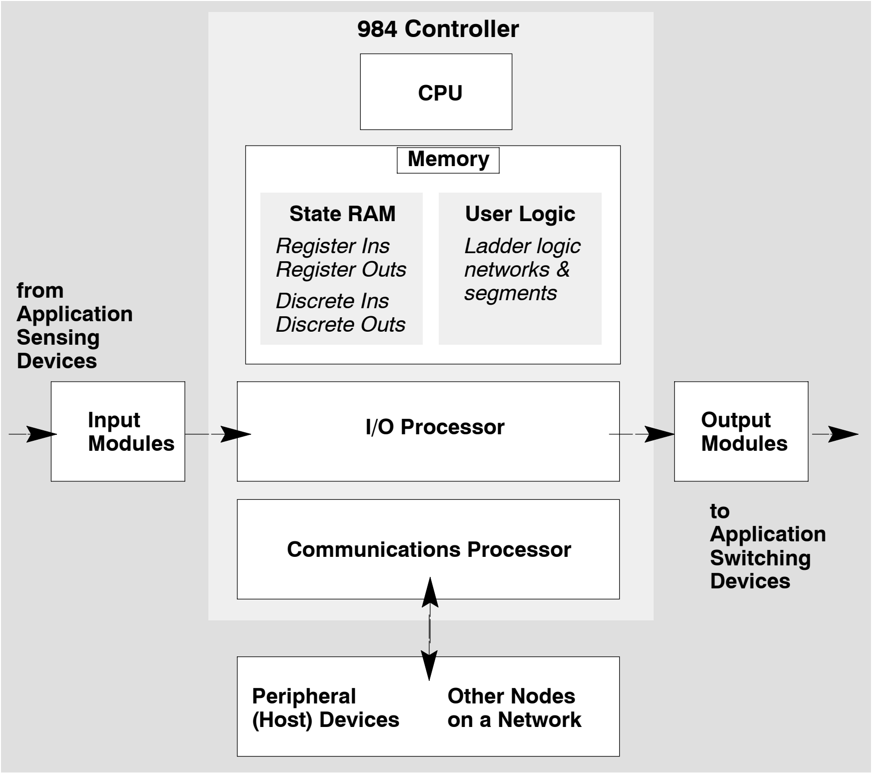
1.3 How a 984 System Provides Application Control

A 984 programmable controller is a special-purpose computer with digital processing capabilities, designed for real time control in industrial and manufacturing applications. In essence, a programmable controller *monitors* the state of field devices by receiving signals from its input modules, *solves* a user logic program via its CPU component, and *directs* further field device activity by sending control signals to its output modules.

1.3.1 The 984 Control Architecture: An Overview

All controllers in the 984 family share a common processing architecture, which comprises:

- A *memory* section that stores user logic, state RAM, and system overhead in battery-backed CMOS RAM and holds the system's Executive firmware in nonvolatile ROM
- 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 peripheral (master) devices as well as with additional controllers and other nodes on a communications network



1.3.2 Reliability and Maintainability

Modicon designs fault protection and isolation features into all 984 controllers. Orderly system startup and shutdown procedures help protect system memory, state RAM, and system hardware from damage due to external power failures.

Long-life lithium batteries back up system memory and state RAM in the event of an unexpected power failure. When power has been restored, a series of internal controller checksum diagnostics validate that RAM data are consistent with the values that were active at the time of power-down.

1.4 P190-Style Panel Software Support

Modicon provides P190 panel software (SW-CS9T-0TB) on specially constructed cassette tapes, and P190 emulation software on 5.25 in (SW-CS9D-5DA) and 3.5 in (SW-CS9D-3DA) diskettes for the P230 Programming Panel or for IBM-XT, -AT, or compatible Personal Computers.

1.4.1 Standard Panel Software Editors

Standard panel software packages contain the following editors:

Software Editor	3.5 in Diskette	5.25 in Diskette	P190 Tape	Editor Description
Configurator	✓	✓	✓	Defines control and communication parameters, allocates memory, accesses controller operations
Traffic Cop	✓	✓	✓	Links discrete and register reference numbers to locations in the I/O subsystems
Programmer	✓	✓	✓	Generates, edits, monitors ladder logic, and accesses controller operations
ASCII Programmer	✓	✓	✓	Generates and edits ASCII-formatted messages
LRV	✓	✓		Loads programs from disk to controller, records 984 memory to disk, compares programs on disk and in memory
Tape Loader			✓	Records user logic on tape, loads programs to 984 memory, compares programs on tape and in memory
Ladder Lister	✓	✓		Generates hard copy of user logic program
Annotated Ladder Lister*	✓	✓		Prints user comments along with hard copy of the user logic program
Utility			✓	Accesses controller memory, prints ladder listing, accesses controller operations
Executive	✓	✓		Overview menu for PC programming software

* There is no editor feature comparable to the Annotated Ladder Lister in the P190 Panel Software package.

1.4.2 Special Loadable Software

Additional loadable software is available to support optional controller hardware and special purpose applications:

Software Loadable	3.5 in Diskette	5.25 in Diskette	P190 Tape	Program Description
HSBY	✓	✓	✓	Enables switchover of controller functions to a back-up controller without downtime
CALL	✓	✓	✓	Expands controller's processing capabilities by calling C functions from a Coprocessor library
MBUS/PEER	✓	✓	✓	Enables peer-to-peer communications via Modbus II
PID2**	✓	✓	✓	Enables configuring, tuning, and monitoring of closed loop control system
MSTR* **	✓	✓		Provides Modbus Plus capabilities via the S985 option module
DRUM/ICMP	✓	✓	✓	Simplifies implementation of sequential step oriented logic
Advanced Math/DX*	✓	✓	✓	Provides enhanced math and data transfer capabilities
EARS	✓	✓		Provides an event/alarm reporting system that detects and time-stamps changes in events, and places the data in a controller buffer where it can be accessed by a host computer or high speed network

---

* Advanced math functions include log, antilog, square root, process square root, and double precision math; advanced DX functions include table-to-block and block-to-table moves and checksum.

** PID2, MSTR, and the advanced math DX functions are provided as loadables for the chassis mount controllers only; comparable functionality is provided as standard in other controllers (see Section 1.6).

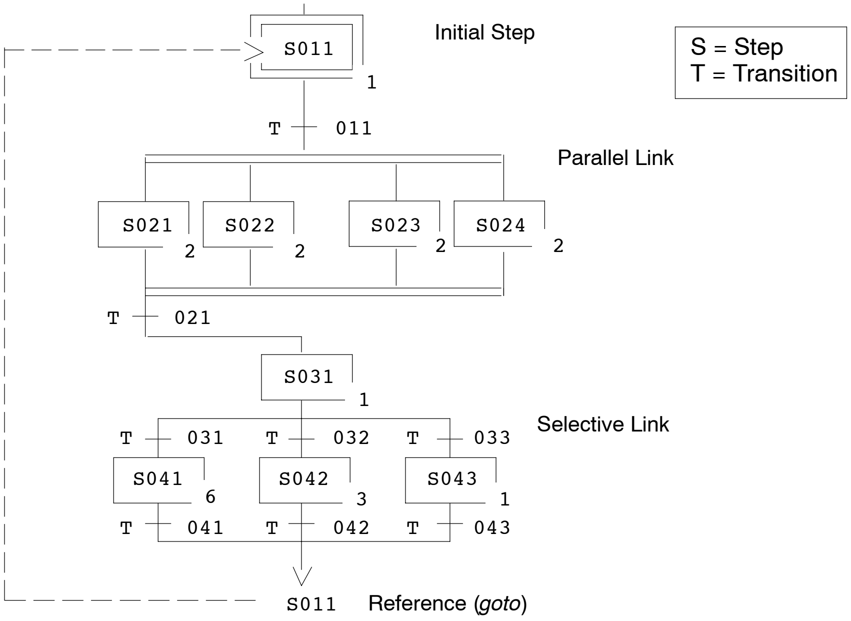
For more details on the loadable software packages, see Chapter 21.

1.5 MODSOFT Panel Software Support

MODSOFT is an integrated software tool for programming, testing, and documenting application logic for 984 controllers that may be used on a P230 Programming Panel or on an IBM-XT, -AT, or compatible Personal Computer. All the editor functions available in the P190 and P190 emulation packages are combined in MODSOFT along with enhanced features. MODSOFT comprises a set of source code editors for programs and for symbolic information. The source programs are subdivided into SFC language and ladder logic.

1.5.1 Sequential Function Charts

SFC is an optional feature that allows you to generate new programs arranged in blocks rather than the linear sequence of straight ladder logic. A sequential function chart can solve multiple networks in a *parallel link* block or one in a choice of several networks in a *selective link* block.



Logic is solved within a block until a specified transition event informs the CPU to move to the next step. SFC allows application software to be created in a format that more closely emulates an actual machining procedure or process flow; it can help improve system throughput by solving only those networks specified by transition events rather than moving linearly through each network in the program on every scan.

1.5.2 MODSOFT Macros

MODSOFT provides a macro feature that can simplify the task of generating and updating large number of repetitive network structures. Using the macro feature, you can create the repeating structure once, then specify the node values using *macro parameters* rather than standard 984 reference numbers. Each macro can contain up to 66 macro parameters—by using * wild card characters in your naming scheme, you can actually create thousands of parameters/macro.

1.5.3 MODSOFT Operating Modes

You may operate in three modes in MODSOFT:

- **Offline**, where programming and programming modification can be done without using a 984 controller linked to the programming device
- **Online**, where the application is communicating with the controller and any changes made to the program are reflected in the controller
- **Debug**, where any changes made to the logic program are saved simultaneously in the 984 controller and in the offline program file and where SFC can be monitored for power flow

1.6 Overview of the 984 Instruction Set

The following instructions are *standard in all 984 System Executives*:

Instruction	Meaning
─┤ ├─	Normally open contact
─┤/├─	Normally closed contact
─┤↑├─	Positive transitional contact
─┤↓├─	Negative transitional contact
─( )─	Coil
─(L)─	Latch coil
Calculations Functions	
ADD	Addition
SUB	Subtraction, greater than, less than, and equal to
MUL	Multiplication
DIV	Division
Counting & Timing Functions	
UCTR	Up counter from 0 to a preset
DCTR	Down counter from a preset to 0
T1.0	Timer that increments in seconds
T0.1	Timer that increments in tenths of a second
T.01	Timer that increments in hundredths of a second
Data Transfer (DX) Move Functions	
R→T	Register-to-table move
T→R	Table-to-register move
T→T	Table-to-table move
BLKM	Block move
FIN	First-in operation to a queue
FOUT	First-out operation from a queue
SRCH	Table search
STAT	Programmable controller health status
DX Matrix Functions	
AND	Logical AND of two matrices
OR	Logical inclusive OR of two matrices
XOR	Logical exclusive OR of two matrices
COMP	Logical complement of one matrix
CMPR	Logical compare of two matrices
MBIT	Logical bit modify
SENS	Logical bit sense
BROT	Logical bit rotate
SKP	A skip function

The following instructions may be available in *standard executive*, *loadable*, or *executive upgrade* form, depending on controller type:

Instruction	Meaning
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
PID2	Performs proportional-integral-derivative control functions

The following are *standard in some Executives* and *unavailable* in others:

Instruction	Meaning
Available with 984s that Support Remote I/O	
READ	Reads data from an ASCII device to 984 memory
WRIT	Sends data from a 984 to an ASCII device
Available in 984s with Extended Memory	
XMRD	Reads function for 984s with Extended Memory
XMWT	Writes Extended Memory data
Available in 984s with Modbus Plus Capabilities	
MSTR	Reads, writes, and gets status of MB+ network operations
Available in 984s with Subroutines Capabilities	
JSR	Jumps the CPU from scheduled logic to a ladder logic subroutine
LAB	Labels the entry point for a ladder logic subroutine
RET	Returns the CPU from a subroutine to scheduled ladder logic
Unavailable in Chassis Mount Controllers	
EMTH	Performs extended math functions—square root, process square root, log, antilog, and floating point functions
Unavailable in Controllers that Support Modbus Plus	
CKSM	Performs CRC-16, LRC, straight, or binary add checksum functions

The following are available as loadables in some controllers:

Instruction	Meaning
HSBY	Supports a Hot Standby control system
MBUS, PEER	Supports Modbus II read/write/status capabilities
CALL	Supports C986/C996 Coprocessor capabilities
DRUM, ICMP	Support drum sequencer applications
MATH, DMTH	Perform some extende math functions in 984s that don't use EMTH
FN*xx*	Supports a user-developed library of custom loadable functions
EARS	Supports an event/alarm reporting system

For more details regarding loadable instructions, see Chapter 21.

Chapter 2
Optional and Peripheral Control Devices

- ❐ Programming Panels
- ❐ The P965 Data Access Panel
- ❐ The Hot Standby Option Modules
- ❐ The Coprocessing Option Modules
- ❐ Optional Communication Modules

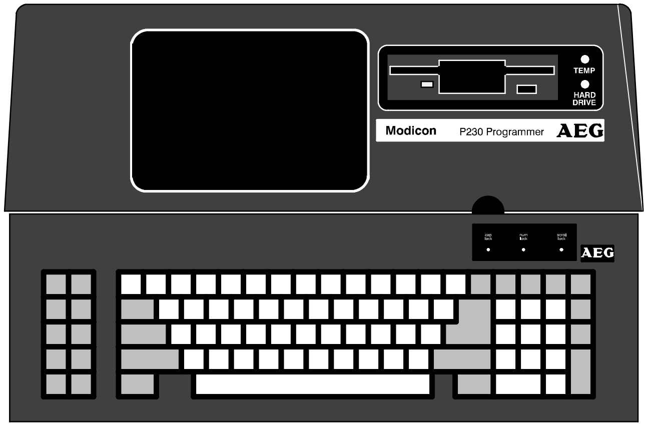
2.1 Programming Panels

Modicon offers two kinds of industrially hardened programming panels—the P230 and the P190. These panels may be used to:

- Start and stop the controller
- Enter, modify, and archive ladder logic programs
- Monitor the register and discrete values in user memory and state RAM
- Enable, disable, and force discrete inputs and coils
- Display and modify the contents of holding registers
- Display and set communication parameters for the communication ports
- Provide on-line monitoring of power flow

2.1.1 The P230

The AS-P230-000 is a portable programming panel with a 40 Mbyte hard disk formatted and installed with MS-DOS and GW-BASIC interpreter software. It supports both MODSOFT and P190 emulation software, either of which may be loaded from the unit's a 3.5 in disk drive. The P230 power supply is 115/230 VAC user-selectable.



2.1.2 The P190 Panels

The P190 is a Modicon-proprietary portable programming panel software with a set of specially designed digital tapes (see section 1.4) for use specifically in this panel. The P190 does not support the MODSOFT. There are two types of P190 Panels available—the AS-P190-212, which operates on 115 VAC, and the AS-P190-222, which operates on 220 VDC.

2.1.3 Using Industry-standard PCs as Programming Panels

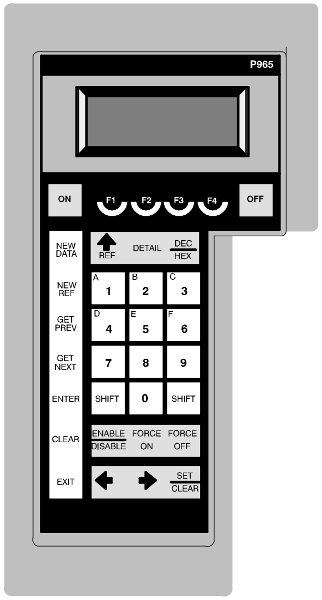
A set of 5.25 in and 3.5 in disks is available to emulate the P190 software on a standard DOS-based PC, and the integrated MODSOFT package is also available on both 5.25 in and 3.5 in distribution disks. These software packages can be run on any IBM-AT or true AT-compatible PC.

2.2 The P965 Data Access Panel

The AS-P965-000 Data Access Panel (DAP) is a hand-held troubleshooting device. It connects to a Modbus port (or ASCII/DAP port on a 984A or 984B) on any Modicon controller that supports Modbus communication.

2.2.1 Physical Design

The P965 DAP is a lightweight device with a 64-character liquid crystal display (LCD) screen and a keypad with alphanumeric and function keys.



2.2.2 How the P965 Can Be Used

A P965 DAP is a very effective tool for monitoring and troubleshooting the controller. With it, you can

- Start and stop the controller
- Monitor the register and discrete values in user memory and state RAM
- Enable, disable, and force discrete inputs and coils
- Display and modify the contents of holding registers
- Display and set communication parameters for the Modbus ports

The P965 can be used on the shop floor to monitor the status of a 984 programmable controller by accessing the STAT block. (Procedures for accessing the STAT block are described in Sections 14.4 and 14.10; the types of statistics available from the STAT block are described in detail in Section 14.5 ... 14.7 for an S901 RIO network and Sections 14.11 ... 14.13 for other 984 I/O networks.

2.3 The Hot Standby Option Modules

The Hot Standby capability has been designed for applications that demand fault-tolerant, high-availability performance. Two identically configured 984 controllers communicate with each other through two Hot Standby option modules, one in each controller. Each controller has the HSBY loadable software function block installed in the first segment of ladder logic (*described in Chapter 21*).

2.3.1 How a Hot Standby System Functions

AM-R911-000 Hot Standby option modules are designed for use in a system involving two identically configured *chassis mount* controllers. AS-S911-800 Hot Standby option modules are designed for use in a system involving two identically configured 984-680, -685, -780, or -785 *slot mount* controllers.

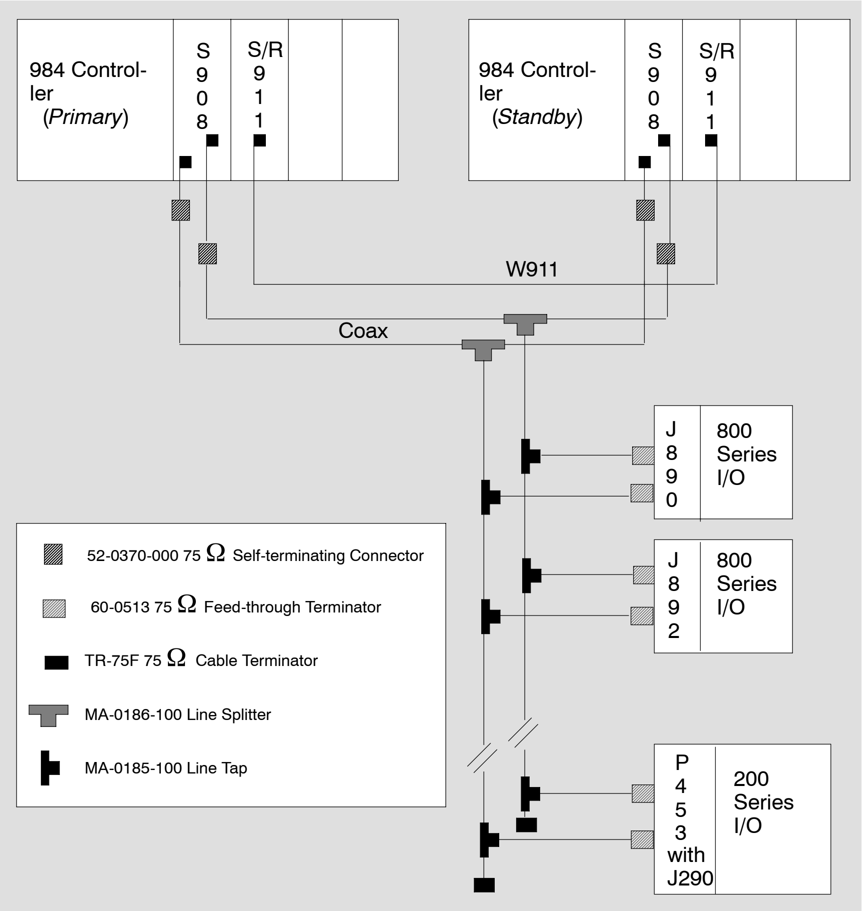
Upon powering up a 984 Hot Standby system, one of the two identically configured 984 controllers acts as the *primary* controller—it reads input data from remote I/O drops, executes the stored user programs from memory, and sends appropriate output commands to the drops. The primary controller updates the standby controller with current system and state RAM status information at the end of each scan.

The *standby* controller only reads this information—it *does not* execute control functions and does not interfere with primary control operations. It will assume primary system control in 13 ... 48 ms if the primary controller fails.

2.3.2 Controller Compatibilities

The S911 and R911 Hot Standby modules are devices designed to be installed in option slots with their host controllers. They work in conjunction with 984 controllers that use S908 Remote I/O Processor modules. The R911 modules work with the 984A, 984B, and 984X *chassis mount* Controllers; the S911 modules work with 984-68*x* and 984-78*x slot mount* Controllers. All hardware and firmware in the primary and standby controllers must be identical.

The two Hot Standby modules in a system are interconnected by a AS-W911-0*xx* cable, and the coaxial cables running from the two S908 RIO Processors pass through self-terminating connectors before being joined by an MA-0186-100 line splitter.

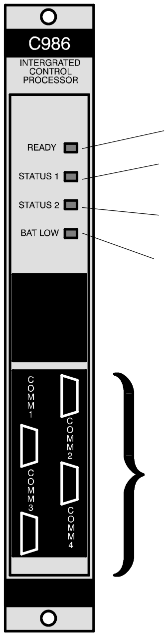


2.4 The Coprocessing Option Modules

Modicon offers two types of integrated control processors (Copros)—the C986 for use with *chassis mount* 984 controllers and the C996 for use with *slot mount* 984 controllers that support option modules. These option modules extend the processing capabilities of your controller, providing alternative programming solutions for problems that are difficult or inefficient to handle via ladder logic.

2.4.1 The C986 Copro for Chassis Mount 984s

The AM-C986-004 Copro resides in a single option slot in a 984A, 984B, or 984X chassis. It uses the flexible, multitasking VRTX Operating System, which allows it to perform parallel application processing, immediate DX processing, and deferred DX processing (see Section 2.5). Programs developed in Microsoft C, either by you or by Modicon, can be downloaded to the Copro and run in parallel with the 984 CPU.



Green READY LED indicates the system is scanning

Green STATUS 1 LED goes ON when the 984 is communicating with the C986

Green STATUS 2 LED goes ON when the C986 is under user software control

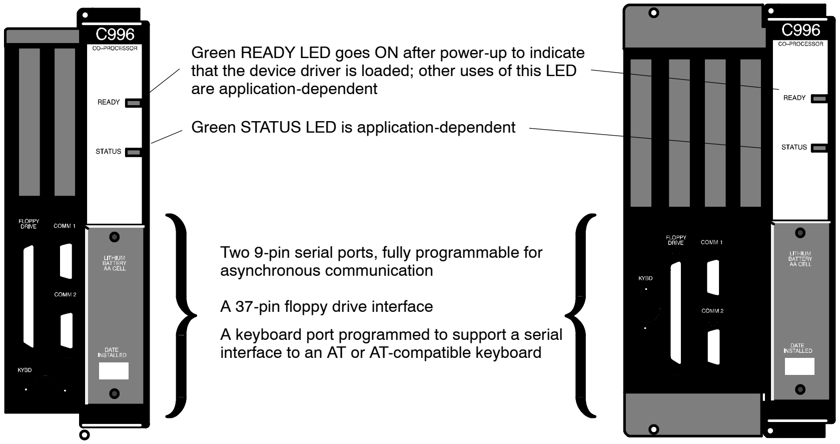
Red BATTERY LOW LED indicates that the battery needs replacing

Nine-pin D-shel subminiature receptacles that can be configured for RS-232C or RS-422

2.4.2 The C996 Copros for Slot Mount 984s

Two coprocessor models are available for use with slot mount controllers—the AM-C996-802 Copro with two expansion slots and the AM-C996-804 Copro with four expansion slots. These copros are DOS-based computer systems with a proprietary high speed interface to 984 controller memory. The C996 Copros can perform parallel application processing and immediate DX processing, but not deferred DX processing (see Section 2.5).

The AM-C996-802 consumes one and a half slots in a slot mount controller housing, and the AM-C996-804 consumes two and a half slots in the housing.



Green READY LED goes ON after power-up to indicate that the device driver is loaded; other uses of this LED are application-dependent

Green STATUS LED is application-dependent

Two 9-pin serial ports, fully programmable for asynchronous communication

A 37-pin floppy drive interface

A keyboard port programmed to support a serial interface to an AT or AT-compatible keyboard

AM-C996-802 AM-C996-804

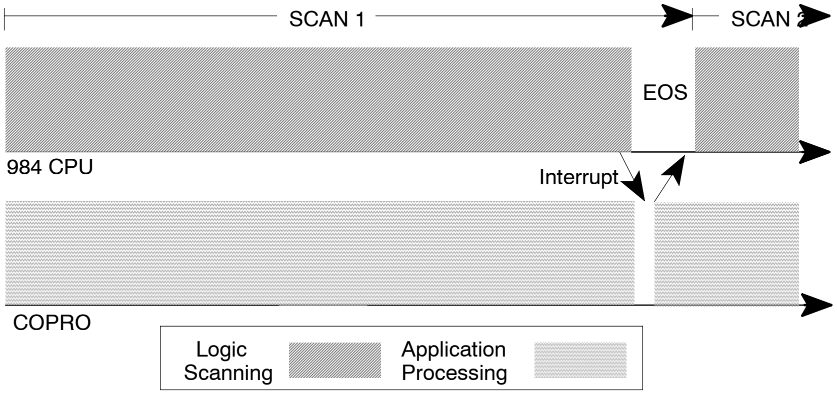
The expansion slots can support various commercially available option cards. The depth dimension of the C996 expansion slots limits your choice of option cards to half-size IBM-XT cards.

2.5 Enhancing Your Processing Environment with a Copro

Both the VRTX-based C986 Copro and the DOS-based C996 Copros can communicate with the controller in two different modes—application mode and immediate DX mode. Only the C986 Copro can communicate with the controller in deferred DX mode.

2.5.1 Application Mode

The C986 and C996 Copros can run programs in *application mode* in parallel with the 984 CPU, exchanging data with the controller at the end of scan (EOS):

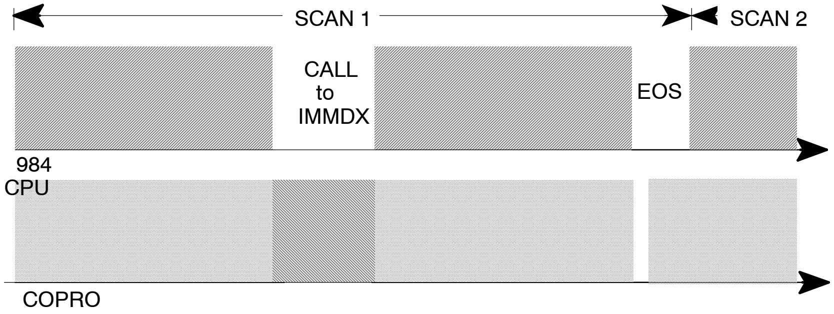


How a Copro Handles Application Processing in Parallel with the 984 CPU

2.5.2 Immediate DX Processing

The C986 and C996 Copros can run standard and customized C routines that are initiated, or *called*, by ladder logic—a loadable CALL function block (*described in Chapter 21*) is provided for this purpose.

When a Copro suspends application processing for a short interval and dedicates itself to the solution of a CALL function, it is performing in *immediate DX mode*. A typical immediate DX function might be a floating point math calculation.

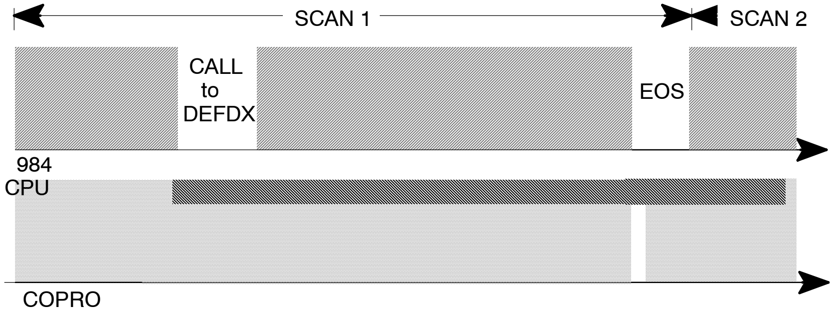


How a Copro Handles Immediate DX Processing

2.5.3 Deferred DX Processing

Because of the multitasking capability inherent of the VRTX Operating System, the C986 can also call deferred DX functions simultaneously with application and immediate DX processing. Up to ten tasks can be supported.

In *deferred DX mode*, DX processing begins with a call and continues until it is finished, even if its processing runs longer than one scan. A typical deferred DX function might be reading bar code input to a serial port.



How the C986 Copro Handles Deferred DX Processing

2.6 Optional Communication Modules

984 Controllers may be interconnected in various kinds of local area (and in some cases long distance) networks. The following 984 controller option modules that allow you to establish the network connections are described here; overall networking capabilities are described in more detail in Chapter 4.

2.6.1 Modbus Modems

The AM-S978-000 Dual Modbus Modem is an option module that allows a *chassis mount* 984 controller to be used as a slave processor in a Modbus network. The AS-J878-000 is an option module that provides similar capability in a *slot mount* 984 controller. These Modbus modems allow you to create Modbus networks up to 15,000 ft (4572 m) long and comprising up to 247 slave nodes.

These modems are electrically compatible with all Modbus products and are sized to fit in one slot (in a 984 chassis in the case of the S978 and in an 800 Series I/O primary housing in the case of the J878). The S978 module contains two modems, which are connected via cable to Modbus ports on the comm processor module in the controller; the J878 module contains one modem.

An S978 Modem accepts digital data from the slave controller in which it resides and modulates the data into an FM analog signal—a form of transmission suited to four-wire cable. It transmits the analog FM signal to the host's Modbus Master device, where it is demodulated to digital data. Conversely, the Modbus Master transmits digital data, which is modulated to an FM analog signal on its way back to the S978 Modem. The S978 demodulates the analog signal to digital data and sends the data to the slave controller in which it resides.

For more information about Modbus network capabilities, see Section 4.6.

2.6.2 Modbus II Modules

The S975 Modbus II Interfaces are option modules that allows a 984 controllers to be used as a processing node in the Modbus II network. The AM-S975-100 module may be used with any *chassis mount* controller, and the AM-S975-820 module may be used with 984-685, -780, or -785 *slot mount* controllers.

Modbus II provides peer-to-peer communication capabilities between 984 controllers and other Modbus II devices over a local area network. For more information about Modbus II networking, see Section 4.9.

Special software must be loaded into the controller to program Modbus II communications in ladder logic. Two loadable function blocks—MBUS and PEER (*described in Chapter 21*)—are used to initiate communications. MBUS writes information to or reads information from a single controller. PEER writes register information to up to 16 controllers simultaneously.

2.6.3 The Modbus Plus Options

Several 984 controllers have a Modbus Plus capability built directly into the controller—i.e, the slot mount 984-385, 984-485, 984-685, and 984-785 Controllers, the Compact 984-145 Controller, and the host based AT-984 and MC-984 Controllers.

For the chassis mount controllers and for the slot mount controllers that accept option modules (the 984-68*x* and -78*x*), various S985 Modbus Plus Adapter cards are available as option modules. An S985 comes with a loadable version of the MSTR function block (described in Chapter 17), which allows you to initiate Modbus Plus communication functions; in 984 controllers with *built-in* Modbus Plus capabilities, the MSTR function is part of the standard executive firmware. The AM-S985-000 card is used with a 984X Controller, the AM-S985-020 is used with a 984A Controller (with an S908 RIOP), and the AM-S985-040 is used with the 984B Controller (with an S908 RIOP).

2.6.4 The Distributed Communications Option

The AS-D908-110 and AS-D908-120 Distributed Control Processors allow you to extend programmable control capabilities over the S908 remote I/O link. These option modules allow entire 984 control systems (CPU and I/O) to appear as remote I/O drops on a higher level remote I/O link. The distributed link is described in Section 4.10.

The D908 modules may be used with a 984-680, -685, 780, and -785 *slot mount* controllers installed at remote locations and connected to a higher level 984 controller via the S908 remote I/O cable. The higher level controller sees this distributed controller as a J890 remote I/O drop. The D908-110 option module supports one cable connection; the D908-120 supports two connections.

Chapter 3
984 I/O Subsystems

☐ I/O Subsystems

☐ Local I/O

☐ Remote I/O

☐ ASCII Communication at Remote I/O Drops

☐ Overview of I/O Support for 984 Controllers

☐ 800 Series I/O Modules

☐ Power Supplies for Local and Remote 800 Series I/O Drops

☐ 200 Series I/O Modules

☐ 500 Series I/O Modules

☐ A120 Series I/O Modules

☐ 300 Series I/O Modules

3.1 I/O Subsystems

The application logic that is stored in and solved by the controller is implemented on the factory floor by input and output modules. These I/O modules are field-wired to sensing or switching devices on the shop floor and linked to the controller over an I/O bus to create a complete control system. Modicon provides several series of I/O modules that may be implemented by different 984 controllers.

3.1.1 Input and Output Modules

An input module accepts electrical signals from field sensing devices, isolates these signals from the controller, and converts them into acceptable voltage levels that update the controller's State RAM.

An output module accepts electrical signals from the controller's state RAM, isolates these signals from the field, and converts them into voltage or current levels necessary to activate working devices or indicator displays on the factory floor.

3.1.2 I/O Module Types

Input and output modules are wired to industrial field devices that send or receive application data. When you plan your I/O layout, match the electrical signal used in the I/O modules with the signal used by the field device to which it is wired. Modicon offers a wide range of I/O modules:

- *Discrete in*, which convert signals coming from field input devices such as pressure switches, limit and proximity switches, or photo sensors into voltage levels that can be used by the controller
- *Discrete out*, which convert voltage levels generated by the controller's logic solving into output signals used by output field sensing devices such as relays, lamps, or solenoids

  Discrete input and output modules are available to support AC, DC, and TTL field input devices
- *Analog in*, which convert analog input signals coming from field input devices such as pressure, level, temperature, or weight sensors into numerical data

that can be used by the controller—this numerical data ranges from 0000 to 4095

- *Analog out*, which convert numerical data generated by the controller's logic solving into analog output signals to be used by output field devices—such as heaters or pumps
- *Special purpose*, designed for unique field applications such as multiplexing, high speed counting, and temperature reading
- *Intelligent*, designed for unique field applications requiring bidirectional (in/out) capabilities and on-board processing power

3.1.3 Local and Remote I/O

I/O subsystems may be *local*—located together with or in close proximity to the controller—or *remote*—located at distances up to 15,000 ft (4.5 km) from the controller, depending on the cable type.

3.2 Local I/O

When local I/O is supported, it consists of one drop only, always designated as drop #1 in your system configuration. Your controller restricts you to one specific series of I/O modules at the local drop.

984 Controllers that Support Local I/O	Local I/O Supported	I/O-to-Controller Connectivity	Local Devices Supported
984X	800 Series I/O	I/O in secondary 800 Series housings* up to 12 ft from controller, connected by W929 cable	Up to five housings supported
984-780, -785	800 Series I/O	In the primary 800 Series I/O housing with controller	Up to five housings supported
984-680, -685	800 Series I/O	In the primary 800 Series I/O housing with controller	Up to five housings supported
984-480, -485	800 Series I/O	In the primary 800 Series I/O housing with controller	Up to two housings supported
984-380, -381, -385	800 Series I/O	In the primary 800 Series I/O housing with controller	Up to two housings supported
Micro-984	300 Series I/O	Built-in I/O bus with side-to-side connectors between controller and other modules	Up to 14 I/O modules supported
984-120, -130, -145	A120 Series I/O	In primary DTA housing with controller	Up to 18 I/O modules supported in up to four DTA housings

* Because the I/O modules reside in a separate housing from the 984X Controller, the I/O modules must receive their power from one or more independent slot mount power supply modules.

3.3 Remote I/O

When remote I/O is supported, the 984 controller may support several drops—in some cases as many as 32. In a remote I/O configuration, an RIO processor in the controller is connected via a coaxial cable system to an RIO interface device at each remote drop.

All 984 controllers that support remote I/O have been designed to drive 800 Series I/O at the remote drops. Several option modules and/or field modification kits are available that allow you to drive installed bases of 200 and 500 Series I/O at remote drops as well.

3.3.1 Remote I/O Drop Interfaces

At each remote drop is a remote I/O (RIO) interface device that communicates over the coaxial cable with the RIO processor in the controller. The RIO interface passes data to and from the I/O modules in the drop over the I/O housing backplane and passes data to and from the 984 controller over the RIO cable system. An RIO interface also contains a set of switches that you use to address all the drops in your system.

There are various kinds of RIO Interfaces you can use, depending on the I/O Series in the drop and the type of RIO processor in the controller. According to your application requirements, you may select RIO Interfaces that provide the drop with ASCII device support.

For a detailed discussion of the planning, installing, and testing an RIO cable system, refer to the ***Modicon Remote I/O Cable System Planning Guide*** (GM-0984-RIO).

3.4 ASCII Communication at the Remote I/O Drops

A 984 Controller that communicates with remote I/O allows you to connect ASCII data entry and data display devices at as many as 16 drop sites. Special types of remote I/O interface devices must be used at drops when ASCII devices are used.

3.4.1 RIO Interfaces that Support ASCII Communication

The J812 and J892 Remote I/O Interfaces (for 800 Series I/O) and P453 Remote I/O Interface (for 200 and 500 Series I/O) have 25-pin female ASCII ports; the P892 RIO Interface (for 800 Series I/O) has 9-pin female ASCII ports:

25-Pin Male
RIO ASCII Port
(J812, J892, P453)

SHIELD	1
TX	2
RX	3
RTS	4
CTS	5
DSR	6
GROUND	7
DTR	20

9-Pin Male
RIO ASCII Port
(P892)

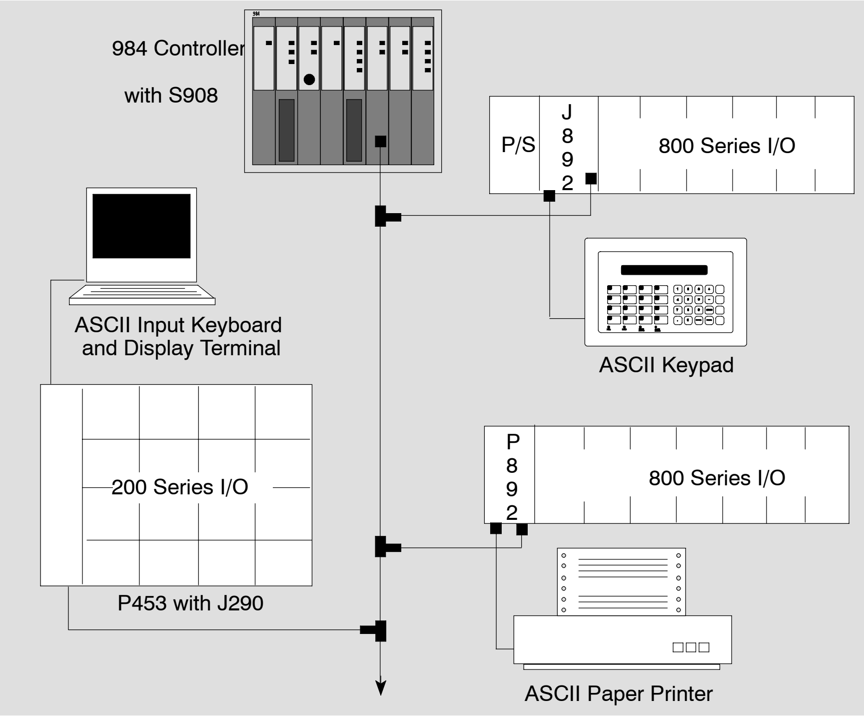
SHIELD	1
RX	2
TX	3
DTR	4
GROUND	5
DSR	6
RTS	7
CTS	8

Each of these RIO Interface devices can support two ASCII devices. As many as 32 ASCII devices can be run from a 984 controller, two/drop from up to 16 drops.

3.4.2 ASCII Device Programming

Two three-node function blocks—READ and WRIT—are provided in the Executive PROM of all 984 controllers with RIO capabilities. The function blocks are implemented in user logic to handle ASCII message passing between the remote devices and controller memory.

ASCII messages may be written to 984 system memory from an ASCII input device (a keyboard, a bar code reader, a pushbutton panel) at a remote drop via a READ function; the controller may send messages to an ASCII display device (a CRT, a printer) via a WRIT function.



An ASCII editor in your panel software allows you to create, edit, and manage a library of ASCII messages to be read or written over the RIO communication link. These ASCII messages reside in a table that occupies space in user logic memory.

3.4.3 The ASCII Operator Keypad

An ASCII Operator Keypad with an AS-KPPR-000 option board can be connected directly to an S908 RIO network and can be cofigured as a drop on that network. This keypad has two ASCII ports associated with it, one as the keypad interface and one that can be connected to another external device.

3.5 Overview of I/O Support for 984 Controllers

984 Type	I/O Series	Local	RIO	RIO Processor	RIO Drop Interface	ASCII
984A,	800		✓	S908	J890/P890	No
984B					J892/P892	Yes
				S901	J810	No
					J812	Yes
	200		✓	S908	P451 & J291	No
					P453 & J290	Yes
				S901	P451	No
					P453	Yes
	500		✓	S908	P451 & J291 *w* J540	No
					453 & J290 *w* J540	Yes
				S901	P451 *w* J540	No
					P453 *w* J540	Yes
984X	800	✓	✓	S929	J890/P890	No
					J892/P892	Yes
	200		✓	S929	P451 *w* J291	No
					P453 *w* J290	Yes
	500		✓	S929	P451 *w* J540 & J291	No
					P453 *w* J540 & J290	Yes
984-785,	800	✓	✓	S908	J890/P890	No
984-780,					J892/P892	Yes
984-685,	200		✓	S908	P451 *w* J291	No
984-680,					P453 *w* J290	Yes
984-485,	500		✓	S908	P451 *w* J540 & J291	No
984-480					P453 *w* J540 & J290	Yes
984-385,						
984-381,						
984-380	800	✓		N/A	N/A	No
AT-984,	800		✓	N/A	J890/P890	No
MC-984,					J892/P892	Yes
Q984	200		✓	S908	P451 *w* J291	No
					P453 *w* J290	Yes
	500		✓	S908	P451 *w* J540 & J291	No
					P453 *w* J540 & J290	Yes
984-120,	A120	✓		N/A	N/A	No
984-130,						
984-145						
Micro-984	300	✓		N/A	N/A	No

3.6 800 Series I/O Modules

3.6.1 800 Series Discrete Input Modules

Model	Voltage Range	Disc. Ins	Number/ Common	Power Draw (mA) +5.0V	+4.3V	-5.0V	Connector
AS-B803-008	115VAC	8	1	27	1	2	AS-8534-000
AS-B805-016	115VAC	16	8	40	1	14	AS-8535-000
AS-B807-032	115VAC	32	8	80	2	0	AS-8535-000
AS-B809-016	230VAC	16	8	42	1	15	AS-8534-000
AS-B817-116	115VAC	16	1	25	25	8	AS-8535-000
AS-B817-216	230VAC	16	1	25	25	8	AS-8535-000
AS-B821-008	10...60VDC	8	2	27	1	0	AS-8534-000
AS-B825-016	24VDC	6	8	27	2	0	AS-8534-000
AS-B827-032	24VDC	32	32	30	1	0	AS-8535-000
AS-B829-116	5V TTL	16	16	27	1	0	AS-8534-000
AS-B833-016	24VDC	16	8	27	2	0	AS-8534-000
AS-B837-016	24VAC/DC	16	8	40	1	15	AS-8534-000
AS-B849-016	48VAC/DC	16	8	40	1	15	AS-8534-000
AS-B853-016	115VAC 125VDC	16	8	40	1	15	As-8534-000
AS-B881-001*	24VDC	16	16	30	1	0	As-8534-000

*The B881 Module must be addressed as one register IN (3*x*) and one register OUT (4*x*).

3.6.2 800 Series Discrete Output Modules

Model	Voltage Range	Disc. Outs	Number/ Common	Power Draw (mA) +5.0V	+4.3V	-5.0V	Connector
AS-B802-008	115VAC	8	2	76	240	0	AS-8534-000
AS-B804-016	115VAC	16	8	76	480	0	AS-8534-000
AS-B806-032	115VAC	32	8	210	1	N/A	AS-8535-000
AS-B808-016	230VAC	16	8	76	480	0	AS-8534-000
AS-B810-008	115VAC	8	1	50	240	0	AS-8534-000
AS-B814-108	Relay	8	1	107	800	0	AS-8534-000
AS-B820-008	10...60VDC	8	2	90	80	0	AS-8534-000
AS-B824-016	24VDC	16	8	32	260	0	AS-8534-000
AS-B826-032	24VDC	32	32	90	1	0	AS-8535-000
AS-B828-016	5V TTL	16	16	32	220	0	AS-8534-000
AS-B832-016	24VDC	16	8	32	235	0	AS-8534-000
AS-B836-016	12...250VDC	16	1	50	603	0	AS-8535-000
AS-B838-032	24VDC	32	8	160	1	0	AS-8535-000
AS-B840-108	Reed Relay	8	1	67	400	0	AS-8534-000
AS-B881-108	120 VAC	8	optional	285*	240	0	AS-8535-000
AS-B882-032	24 VDC	32	8	300**	10	0	AS-8535-000

* When all outputs are ON, power draw at +5 V is 285 mA maximum on the B881-108; when all outputs are OFF, power draw at +5 V is210 mA maximum.

** When all outputs are ON, power draw at +5 V is 300 mA on the B882-032; when all outputs are OFF, power draw is 200 mA.

3.6.3 800 Series Analog Input Modules

Model	Application Ranges	Analog Inputs	Power Draw (mA) +5.0V	 +4.3V	 -5.0V	Connectors
AS-B873-001	4...20mA; 1...5V	4	300	300	0	AS-8533-002 (Included)
AS-B873-002	-1...+10V	4	300	300	0	AS-8533-002 (Included)
AS-B875-002	4...20mA; 1...5V	4	300	300	0	AS-8533-002 (Included)
AS-B875-012	-10...+10V	4	300	300	0	AS-8533-002 (Included)
AS-B875-101	4...20mA; -10...+10; -5...+5V; 0...10V; 0...5V; 1...5V	8	650	975	0	AS-8533-004 (Included)
AS-B875-111	0...5V, 1...5V -5...+5V, 0...10V, -10...10V, 0...2mA, 0.4...2mA, -2...+2mA	8 Differential 16 Single-ended	500	900	0	AS-8535-000 (included)

3.6.4 800 Series Analog Output Modules

Model	Application Ranges	Analog Outputs	Power Draw (mA) +5.0V	 +4.3V	 -5.0V	Connectors
AS-B872-100	4...20mA	4	800	5	0	AS-8535-000 (included)
AS-B872-200	0...5V, 0...10V -5...+5V, -10...+10V	4	800	5	0	AS-8535-000 (included)

3.6.5 800 Series Special Purpose I/O Modules

Model	Description	Power Draw (mA) +5.0V	+4.3V	-5.0V	Addressable Registers(I/o)	Connector
AS-B846-001	MUX: 16 Voltage Inputs	65	1	0	0/1	AS-8535-000
AS-B846-002	MUX: 16 Current Inputs	65	1	0	0/1	AS-8535-000
AS-B864-001	TTL Register: 8 outputs; 8/common	220	180	0	0/8	AS-8535-000
AS-B865-001	TTL Register: 8 inputs; 8/common	400	600	0	8/0	AS-8535-000
AS-B882-239	High Speed Counter: 2 UpCounts 0...30kHz	188	0	0	2/2	AS-8533-005 (Included)
AS-B883-001	High Speed Counter: 2 Up/Down Counts: 0...50kHz; Internal Clock	680	0	0	3/3	52-0325-000 (Included)
AS-B883-200	Reads ten Thermocouple Inputs: Types B,E,J,K, R,S,T,N, or linear mV	300	0	0	3/3	52-0325-000 (Included)
AS-B883-201	Reads 8 RTD Inputs: 2 or 3-wire; American or European 100Ω Platinum	400	5	0	3/3	52-0325-000 (Included)

3.6.6 800 Series Intelligent I/O Modules

Intelligent I/O modules perform tasks that require special on-board processing capabilities.

Model	Description	Power Draw (mA) +5.0V	+4.3V	-5.0V	Addressable Registers(I/O)	Connector
AS-B883-101	CAM Emulator: Absolute Encoder Input, 8 Discrete Outputs	1000	0	0	3/3	52-0325-000 (Included)
AS-B883-111	CAM Emulator w/ Velocity Compensation	1000	0	0	3/3	52-0325-000 (Included)
AS-B884-002	PID: 2 Loops, Cascadable, Standalone, 11 Total I/O	50	0	0	4/4	AS-8644-000 (Included)
AS-B885-002	ASCII/BASIC: 64K RAM, 2 RS232/422 Ports	500	1760	0	6/6	N/A
AS-B984-100	Discrete High Speed Logic Solver	0	0	0	4/4 or 8/8	AS-8533-004 (Included)

3.6.7 800 Series MMI Operator Panels

A variety of prepackaged man-machine interface (MMI) devices may also be connected to the RIO network.

Two types of 32 Element Pushbutton Panels may be installed and traffic copped like I/O at remote S908/S929 drops. The MM-32SD-000 Panel is connected via a W801 cable to an 800 Series I/O drop being driven by an S908-compatible RIO interface device. By adding an MM-32PR-000 Primary Option board to this operator panel, you create a primary device that can be connected directly to the S908 RIO network.

A PanelMate Plus Video Control Panel may also be installed as a drop on an RIO network. PanelMate Plus is traffic copped like a D908 Distributed Control Processor (see Section 4.10).

3.7 Power Supplies for Local and Remote 800 Series I/O Drops

To determine the power requirements of a drop, add the individual power draws of each module in the drop. A *primary* power supply is required in the first slot of the primary housing in a remote I/O drop; an *auxiliary* power supply may be installed in the first slot of a secondary housing:

Power Supplies for a Remote 800 Series I/O Drop

Model	Description	Voltage	I/O Power (in mA) +5V	+4.3V	-5V	RIO Interface Power (@ +5V)
AS-P810-000	primary/aux	120/220VAC	5000*	5000*	300	7500 mA* **
AS-P802-001	primary/aux	120/220AC	2500***	10100***	500	9500 mA***
AS-P884-001	primary/aux	120/220VAC	5000	10100	500	11000 mA
AS-P800-003	primary/aux	120/22VAC	2500***	10100***	500	9500 mA***
AS-P890-000 AS-P892-000	primary (in an RIO interface)	115/230VAV 24VDC	3000#	3000#	250	N/A
AS-P830-000	auxiliary only	120/240VAC 24VDC	5000##	6000##	500	N/A

* Total maximum of +5V I/O, +4.3V I/O, and +5V Interface cannot exceed 13500 mA
** Total maximum of +5V I/O and +4.3V I/O cannot exceed 5000 mA
*** Total maximum of +5V I/O, +4.3V I/O, and +5V Interface cannot exceed 16100 mA
\# Total maximum of +5V I/O and +4.3V I/O cannot exceed 3000 mA
\## Total maximum of +5V I/O and +4.3V I/O cannot exceed 6000 mA

A slot mount 984 controller provides the primary power supply for its local I/O drop; auxiliary power supplies listed above may be used in secondary housings:

Primary Power Supplies for a Local 800 Series I/O Drop

Model	Voltage	I/O Power (in mA) +5V	+4.3V	-5V	Total Maximum Power (in mA)
PC-0984-785/ -780/-685/-680	120/220VAC 24VDC	8000	6000	500	8000
PC-0984-485/ -480/-385/-381/-380	120/220VAC 24VDC	3000	3000	250	3000

3.8 200 Series I/O Modules

200 Series I/O modules may be used at remote I/O drops in conjunction with any chassis mount, slot mount, or host based 984 controller; they cannot be used at local drops. The 200 Series provides discrete in, discrete out, analog in, analog out, and special purpose I/O modules.

3.8.1 200 Series Discrete Input Modules

Model	Voltage Range	Number of Inputs	Number per Common
AS-B225-001	24VDC (True High)	16	1
AS-B231-501	115VAC	16	4
AS-B233-501	24VDC	16	4
AS-B235-501	220VAC	16	4
AS-B237-001	5VDC (TTL)	16	4
AS-B245-001	220VAC (Isolated)	8	Separate Commons
AS-B247-001	115VAC	8	Separate Commons
AS-B271-001	36...60VAC	16	4
AS-B273-001	12VDC (Intrinsically Safe)	16	4
AS-B275-501	10...60VDC	16	4
AS-B279-001	18...30VAC	16	4

3.8.2 200 Series Discrete Output Modules

Model	Voltage Range	Number of Outputs	Number per Common
AS-B224-001	24VDC (True High)	16	1
AS-B230-501	115VAC	16	4
AS-B232-501	24VDC	16	4
AS-B234-501	220VAC	16	4
AS-B236-501	5VDC (TTL)	16	4
AS-B238-001	24VDC (True Low)	16	4
AS-B244-101	230VAC (Isolated)	8	Separate Commons
AS-B246-501	115VAV (Isolated)	8	Separate Commons
AS-B248-501	10...60VDC	16	4
AS-B266-501	115VAC (Reed Relay, NO)	8	Separate Commons
AS-B268-001	230VAC (Reed Relay, NO)	8	Separate Commons
AS-B270-001	48VAC	16	4
AS-B274-001	115VAV (Relay, NC)	8	Separate Commons
AS-B276-001	230VAC (Relay, NC)	8	Separate Commons
AS-B278-001	10..60VAC	16	4

3.8.3 200 Series Analog Input Modules

Model	Application Range	Number of Channels	Words(I/O)
AS-B243-105	1...5VDC, 4...20MADC,	4	4/0
AS-B243-110	0...10VDC, -10...+10VDC	4	4/0

3.8.4 200 Series Analog Output Modules

Model	Application Range	Number of Channels	Words(I/O)
AS-B260-005	1...5VDC	4	0/4
AS-B260-010	0...10VDC	4	0/4
AS-B262-001	1...5VDC, 4...20VDC	4	0/4

3.8.5 200 Series Special Purpose I/O Modules

Model	Description	Number of Inputs	Words(I/O)
AS-B239-001	Dual High Speed Counter	2	2/2
AS-B258-101	16-to-1 Analog MUX (used with a B243 Module)	16	0/1
AS-B281-001	Thermocouple Module	10	10/0
AS-B283-001	RTD Input Module	8	8/0

3.9 500 Series I/O Modules

500 Series I/O modules may be used at remote I/O drops in conjunction with any chassis mount, slot mount, or host based 984 controller; they cannot be used at local drops. The 500 Series provides discrete in, discrete out, and special purpose I/O modules.

3.9.1 500 Series Discrete Input Modules

Model	Voltage Range	Number of Inputs	Number per Common
AS-B531-001	5...28VDC	4 (Latched)	2
AS-B551-001	115VAC	4	Separate Commons
AS-B553-001	9...56VDC	4 (True High)	2
AS-B557-001	5VDC (TTL)	4	2
AS-B559-001	9...56VDC (Current Sink)	4 (True Low)	2
AS-B561-001	90...150VDC	4	Separate Commons
AS-B565-001	18..30VAC	4	Separate Commons
AS-B569-001	30...60VAC	4	Separate Commons
AS-B583-001	Proximity Switch	8 (Intrinsically Safe)	2

3.9.2 500 Series Discrete Output Modules

Model	Voltage Range	Number of Outputs	Number per Common
AS-B550-001	115VAC	4	2
AS-B552-001	9...56VDC	4	2
AS-B554-001	220VAC	4	2
AS-B556-001	5VDC (TTL)	4	2
AS-B558-001	9...56VDC (Current Sink)	4	2
AS-B560-001	90...150VDC	4	Separate Commons
AS-B564-001	20..60VAC	4	2
AS-B592-001	115VAC (Reed Relay, NO)	4	Separate Commons
AS-B596-001	115VAC (Reed Relay, NC)	4	Separate Commons

3.9.3 500 Series Special Purpose I/O Modules

Model	Description	Number of Inputs	Words(I/O)
AS-B570-001	Output Register MUX (16 three-digit, Latch-on-High LEDs)	16	0/8 0r 0/16
AS-B571-001	Input Register MUX (16 three-digit, 9's complement Thumbwheels)	16	8/0 or 16/0
AS-B572-001	D/A Converter 0...10V	2	0/2
AS-B581-001	Absolute Encoder Module	12 bits	1/0

3.10 A120 Series I/O Modules

A120 Series I/O modules are used as local I/O with the -120, -130, and -145 Compact 984 Controllers; they cannot be used in remote I/O configurations. The A120 Series provides discrete in, discrete out, analog in, analog out, and special purpose I/O modules.

3.10.1 A120 Discrete Input Modules

Model	Voltage Range	Disc. Ins	Power Draw Internal (5 V)	Opto-isolation from I/O Bus
AS-BDEP-208	230 VAC	8	< 50 mA	Yes
AS-BDEP-209	120 VAC	8	< 30 mA	Yes
AS-BDEP-216	120 VAC	16	< 15 mA	Yes
AS-BDEO-216	24 VDC	16	< 15 mA	No
AS-BDEP-220	24 VDC	16	< 15 mA	Yes

3.10.2 A120 Discrete Output Modules

Model	Voltage Range	Outs	Power Draw Internal (5V)	Power Draw External (24 V)	Opto-isolation from I/O Bus
AS-BDAP-204	24 VDC or 220 VAC	4 relays	< 25 mA	< 150 mA	Yes
AS-BDAP-208	24 VDC or 220 VAC	8 relays	< 60 mA	< 150 mA	Yes
AS-BDAP-209	120 VAC	8 disc	< 88 mA		Yes
AS-BDAP-216	24 VDC	16 disc	< 50 mA		Yes

3.10.3 A120 Combo Modules

Model	Voltage Range	Ins/ Outs	Power Draw Internal (5 V)	Power Draw External (24 V)	Opto-isolation from I/O Bus
AS-BDAP-212	24 VDC	8 disc/ 4 relays	< 25 mA	< 150 mA	Yes
AS-BDAP-220	24 VDC	8 disc/ 8 disc		< 25 mA	Yes

3.10.4 A120 Analog Input Modules

Model	Application Range (Recommended)	Analog Ins	Power Draw Internal (5 V)	Opto-isolation from I/O Bus
AS-BADU-204	–500 mV ... +500 mV Pt 100 RTD	4	< 30 mA	No
AS-BADU-205	–10 V ... +10 V or –20 mA ... +20 mA	4	< 30 mA	No

3.10.5 A120 Analog Output Module

Model	Application Range (Recommended)	Analog Outs	Power Draw		Opto-isolation from I/O Bus
			Internal	External	
AS-BDAU-202	–10 V ... +10 V or –20 mA ... +20 mA	2	< 60 mA	< 150 mA	Yes

3.10.6 A120 Special Purpose Module

Model	Application	Voltage Range	Power Draw		Opto-isolation from I/O Bus
			Internal (5 V)	External (24 V)	
AS-BZAE-201	Positioner or Counter	24 VDC	<100 mA	< 30 mA	Yes

3.11 300 Series I/O Modules

300 Series I/O modules are used in conjunction with the Micro-984 Controller. The 300 Series provides discrete in, discrete out, analog, and BCD register I/O modules.

3.11.1 300 Series Discrete Input Modules

Model	Voltage Range	Number of Inputs
AS-B351-001	115VAC	8
AS-B353-001	24VDC (True Low)	8
AS-B355-001	220VAC	8
AS-B357-001	24VDC (True High)	8
AS-B359-001	24VAC	8

3.11.2 300 Series Discrete Output Modules

Model	Voltage Range	Number of Outputs
AS-B350-001	115VAC	8
AS-B352-001	24VDC (True Low)	8
AS-B354-001	220VAC	8
AS-B356-001	24VDC (True High)	8
AS-B358-001	24VAC	8
AS-B360-001	Dry Contact (Relay, NO)	6
AS-B360-002	Dry Contact (Relay, NC)	6

3.11.3 300 Series Analog I/O Modules

Model	Application Range	Words(I/O)
AS-B373-001	0...10VDC	2/0
AS-B374-001	1...5VDC/4...20mA	0/2
AS-B375-001	1...5VDC/4...20mA	2/0

3.11.4 300 Series BCD Register I/O Modules

Model	Application Range	Words(I/O)
AS-B370-001	0...5VDC; 3 digits	0/2
AS-B371-001	0...5VDC; 3 digits	2/0

Chapter 4
984 Communications Capabilities

- ❐ Modbus Capabilities
- ❐ Modbus Port Pinouts for the P230 Programming Panel
- ❐ Modbus Port Pinouts for the P190 Programming Panel
- ❐ Modbus Port Pinouts for an IBM-XT
- ❐ A Modbus Network
- ❐ A Modbus Plus Network
- ❐ Bridging Modbus Plus Networks
- ❐ A Modbus II Network
- ❐ Distributed Control Processing
- ❐ Network Topology Overview

4.1 Modbus Capabilities

A Modbus communications capability is resident in all chassis mount, slot mount, and micro 984 controllers. Modbus may be used as the connection for a host device such as a programming panel or data access panel or as the port to a multicontroller master-slave network where a single master device can initiate communications with up to 247 slave nodes.

4.1.1 The Modbus Port Parameters

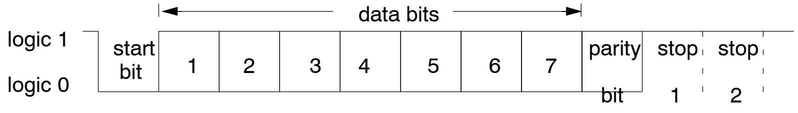
All chassis mount, slot mount, and micro controllers provide at least one Modbus port as a serial communications capability. The communication parameters for your Modbus port(s) may be set by switches on the controller or via the panel software, depending on your controller type. There are three communication parameters:

- *Communication mode*—the protocol, or bit structure, of the message transmissions; either ASCII or RTU (Remote Terminal Unit)
- *Baud*—the data transmission speed, measured in bits/s
- *Parity*—a method of verifying the accuracy of a data transmission, using an additional bit in the message to make the sum of the *1 bits* EVEN or ODD

4.1.1.1 Communication Modes

In ASCII mode, a Modbus port handles messages composed of bytes containing one start bit, seven data bits, one parity bit, and two stop bits:

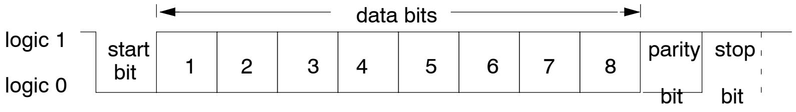
ASCII Mode



ASCII mode uses a restricted character set and character-based message framing, and may be used for communicating with computers, operating systems, packet networks, or other networking devices that may restrict the message content or timing.

In RTU mode, a Modbus port handles messages composed of bytes containing eight data bits and either one parity bit and one stop bit or no parity bit and two stop bits:

RTU Mode

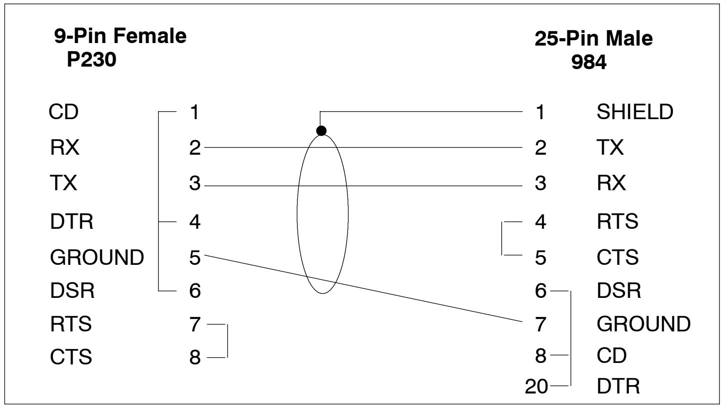


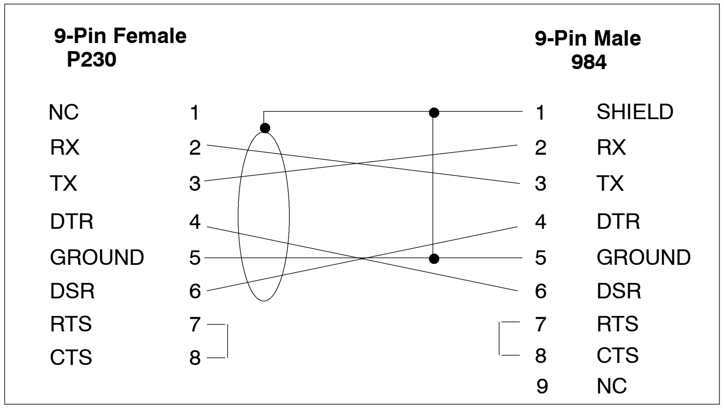
RTU mode packs data bits more compactly in order to increase speed.

4.2 Modbus Port Pinouts for the P230 Programming Panel

The chassis mount controllers provide one or more 25-pin Modbus ports, and the other controllers provide nine-pin ports. Here are the pinouts for for the P230 Panel with these ports. (The same pinouts apply to an IBM-AT Personal Computer and to a FactoryMate Plus Operator Panel.):

P230 to Modbus Pinouts



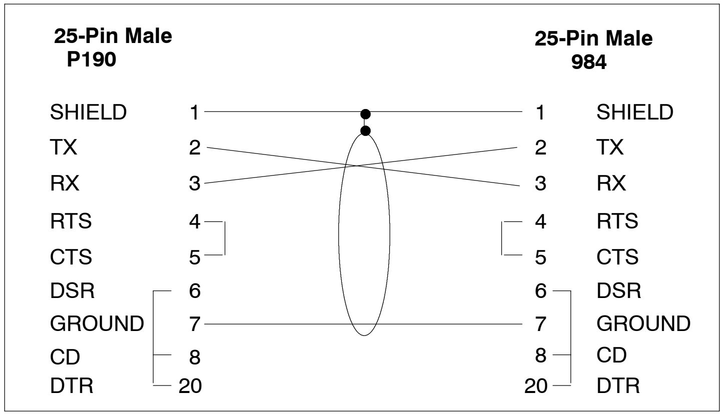


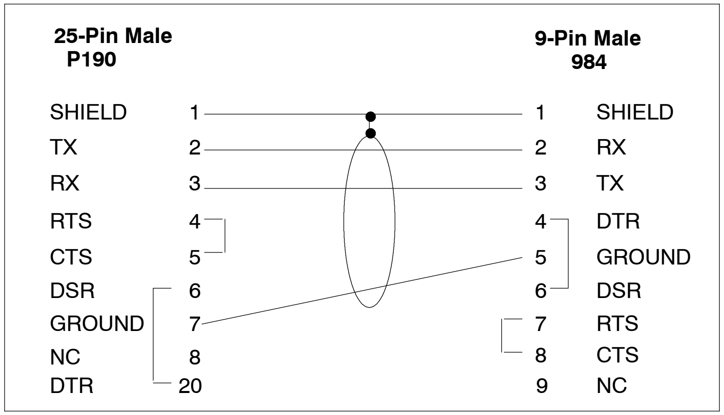
TX:	transmitted data	**DSR**:	data set ready	**CTS**:	clear to send
RX:	received data	**DTR**:	data terminal ready	**NC**:	no connection
RTS:	request to send	**CD**:	carrier detect		

4.3 Modbus Port Pinouts for the P190 Programming Panel

Here are the Modbus port pinouts for the P190 Programming Panel:

P190 to Modbus Pinouts

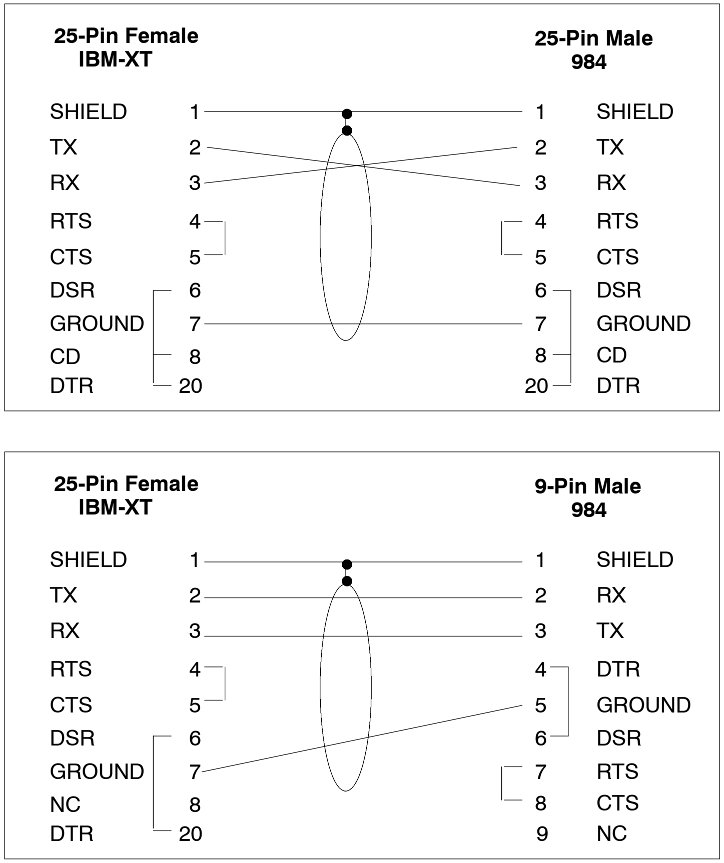




4.4 Modbus Port Pinouts for an IBM-XT

Here are the Modbus port pinouts for an IBM-XT Personal Computer:

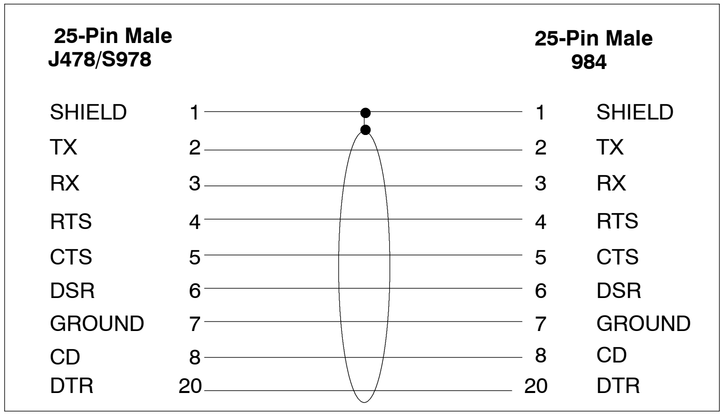
IBM-XT to Modbus Pinouts

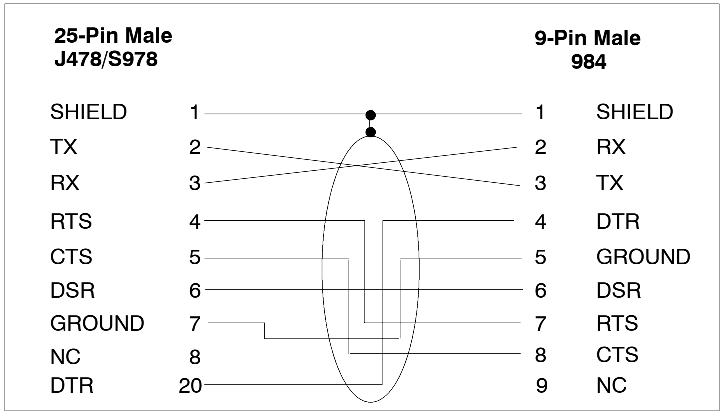


4.5 Modbus Port Pinouts for a Modicon Comm Modem

Here are the Modbus port pinouts for the J478/S978 Modicon Modems:

Comm Modem to Modbus Pinouts





4.6 A Modbus Network

A Modbus network is a master-slave network, and all communications are initiated by a single Modbus master device. The master device requires a modem such as the J478—which transforms digital data into an FM analog signal—and the network slave controllers each require a receptor modem such as a J878, a S978, or another J478 to demodulate FM to digital.

4.6.1 Network Capacity

A Modbus network has one master device that originates all communications to as many as 247 slave nodes throughout the plant (or in remote locations)—the total number of nodes supported depends on the communications equipment used. A Modicon J478 master modem, for example, may support up to 32 slaves over a twisted-pair cable network. Additional J478s may be used as repeaters to extend the number of slave nodes on the network beyond 32.

4.6.2 Communication Media

Slave nodes may be linked via four-wire twisted-pair cable in a local installation up to 15,000 ft (4572 m) long. They may also be linked via common carrier (phone line, radio, microwave) over remote distances or linked locally via other dedicated lines. A well-defined set of network guidelines is available for systems that use Modicon modems and Belden 8777 twisted-pair cable (see ***Modbus System Planning User's Manual***, ML-MBUS-PLN). The requirements for other arrangements depend on the type of commercial facilities selected.

4.6.3 Communication Parameters

All communications on a Modbus network are initiated by the Modbus master. The master device may be a host computer, a dedicated programming panel such as a P190, or a Modicon programmable controller with ASCII (RIO) communication capability. Communications may be of the *query↔response* type—where the master addresses only one slave—or of the *broadcast↔no response* type—where the master simultaneously addresses all slaves.

Commonly used functions over the Modbus network are *READ coil status* (0*x*), *READ input status* (1*x*), *READ/WRIT holding register* (4*x*), *READ input register* (3*x*), and *FORCE coil ON or OFF*.

A library of C functions is available from Modicon—Modcom IIC, SW-APPD-IDC. It allows you to design custom Modbus applications.

The master communicates at a set baud to all slaves on the network. The Modbus ports on all slave devices must be set to a uniform set of communication parameters—this means that if some controllers have a more limited selection of bauds, the entire network is constrained to those selections.

4.7 A Modbus Plus Network

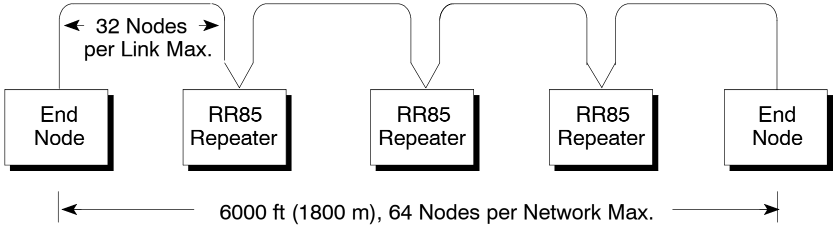
Modbus Plus is a local area network that allows host computers, programmable controllers, and other data sources to communicate as peers throughout an industrial plant via twisted-pair cable. A Modbus Plus network operates at a data transfer rate of one million bits/s.

Modbus Plus networks may be used for

- ❐ Data transfer between controllers
- ❐ Data transfer between controllers and host computers
- ❐ Programming of controllers
- ❐ Uploading/downloading and archiving of application programs from a host

4.7.1 Network Capacity

The network comprises one or more communication links; one comm link may support up to 32 peer devices (nodes); by using an RR85 Repeater, you can join two links to support up to a maximum of 64 Modbus Plus nodes on a network. One communication link may be up to 1500 ft (450 m) long. Additional repeaters (up to three between any two nodes) may be used to extend the network distance—the maximum cable length between any two nodes is 6000 ft (1800 m) in a linear configuration. (The minimum cable length between nodes is 10 ft.)



Maximum Linear Configuration in a Modbus Plus Network

Each node on the network must be assigned a unique address in the range 1 ... 64; the address is generally set via a special DIP switch located on the controller (or on the Modbus Plus Adaptor card inserted in a host computer). Repeaters do not use addresses on the network.

4.7.2 The Logical Network

Nodes on a Modbus Plus network function as peer members of a logical ring, gaining access to the network upon receipt of a token frame. When a node holds a token, it can initiate message transactions with selected destinations—messages may be addressed to any node on the network. The vehicle for initiating a message is the MSTR instruction, an instruction that is standard on 984 controllers that support Modbus Plus. With the MSTR block, you define *source* and *destination* routing information for each message.

4.7.3 The Physical Network

The network medium is two-wire twisted-pair shielded cable, laid out in a sequential multidrop path directly between successive nodes. Use Belden type 9841 cable, available from Modicon in rolls of 100 ft (97-9841-100), 500 ft (97-9841-500), and 1000 ft (97-9841-01K). Taps and splitters are not allowed.

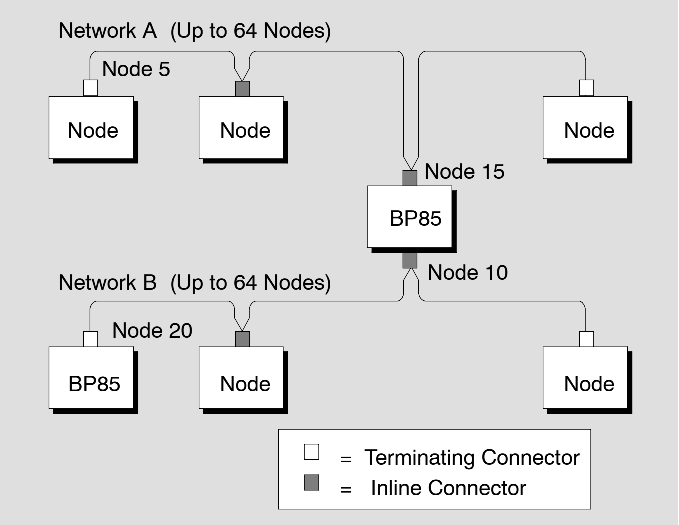
A connector is attached to the cable at each node site and is plugged into a 9-pin Modbus Plus port on each node. Use AS-MBKT-185 terminating connectors at the two ends of a link, and AS-MBKT-085 inline connectors at all other node sites. These connectors are available from Modicon.

4.7.4 Adding and Deleting Nodes from the Network

If your 984 controller is a new or replacement node device on an active Modbus Plus network, you do not need to disable other devices on the network in order to install the new device. Simply disconnect the local drop cable and reconnect it—do not power down the other nodes. The network protocol automatically bypasses a node when it is removed and includes it when it is reconnected. Connectors are built with internal termination resistors and do not have to be connected to a device. You should cover its pins to prevent damage and contamination.

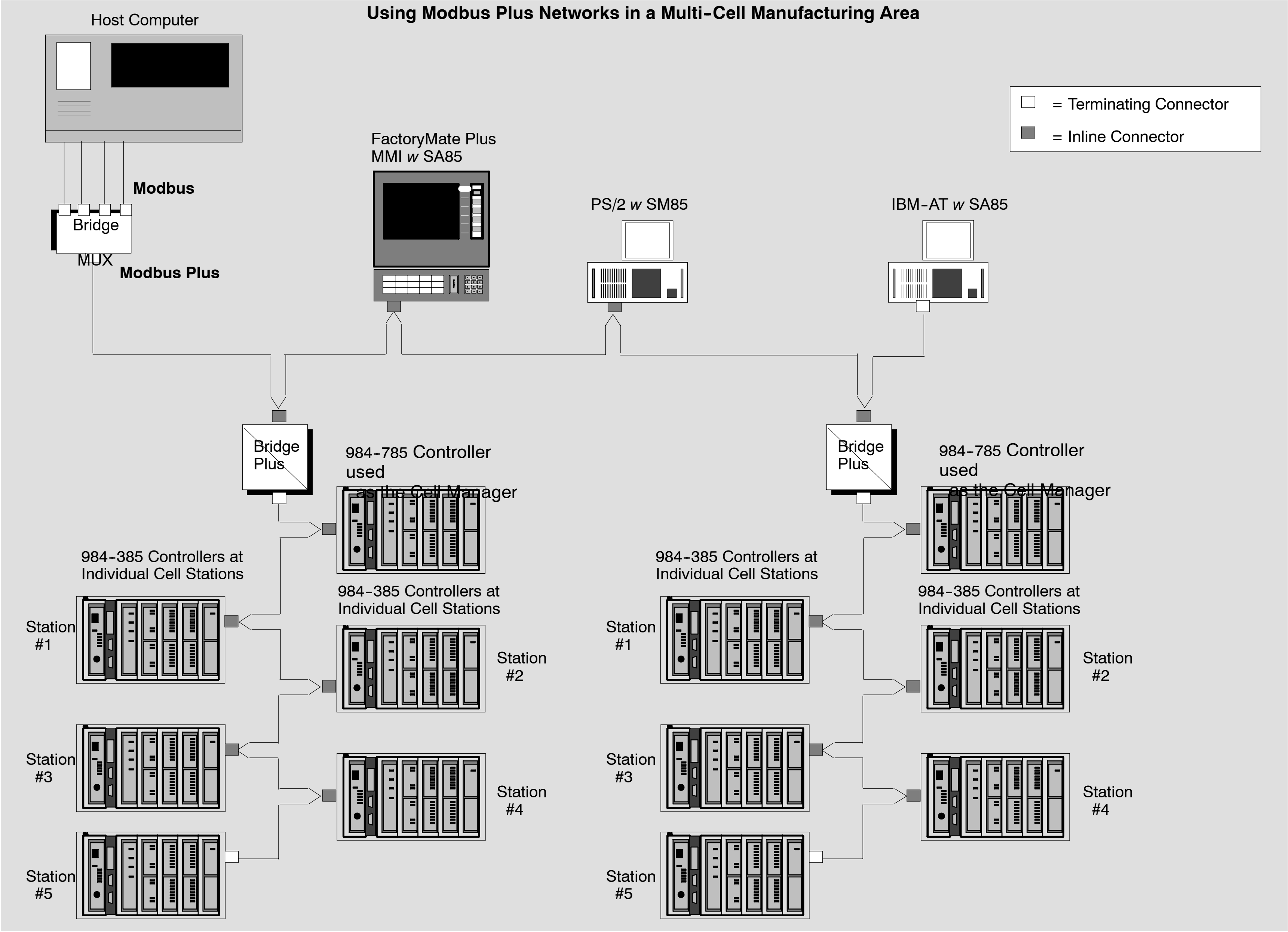
4.8 Joining Modbus Plus Networks

For applications requiring a large number of nodes, you can use the BP85 Bridge Plus device to join multiple Modbus Plus networks. The BP85 has two port connectors and two sets of address switches and is connected as a node on two Modbus Plus networks. The Bridge operates as an independent node on each network, receiving and passing tokens according to each network's address sequence.



The illustration on the following page shows an example of a Modbus Plus system topology.

The Bridge Plus provides the benefit of faster communications on individual networks. Each network maintains faster communication between devices for time-critical control applications, while the bridge facilitates intercommunication between two networks.



Using Modbus Plus Networks in a Multi-Cell Manufacturing Area

4.9 A Modbus II Network

For communication-intensive and time-critical applications, the Modbus II option delivers highly reliable real-time response. It operates at 5 Mbits/s and supports up to 50 nodes. Modbus II is a peer-to-peer network.

A Modbus II network may be used for

- ☐ Data transfer between controllers
- ☐ Data transfer between controllers and host computers
- ☐ Programming of controllers
- ☐ Uploading/downloading and archiving of application programs from a host

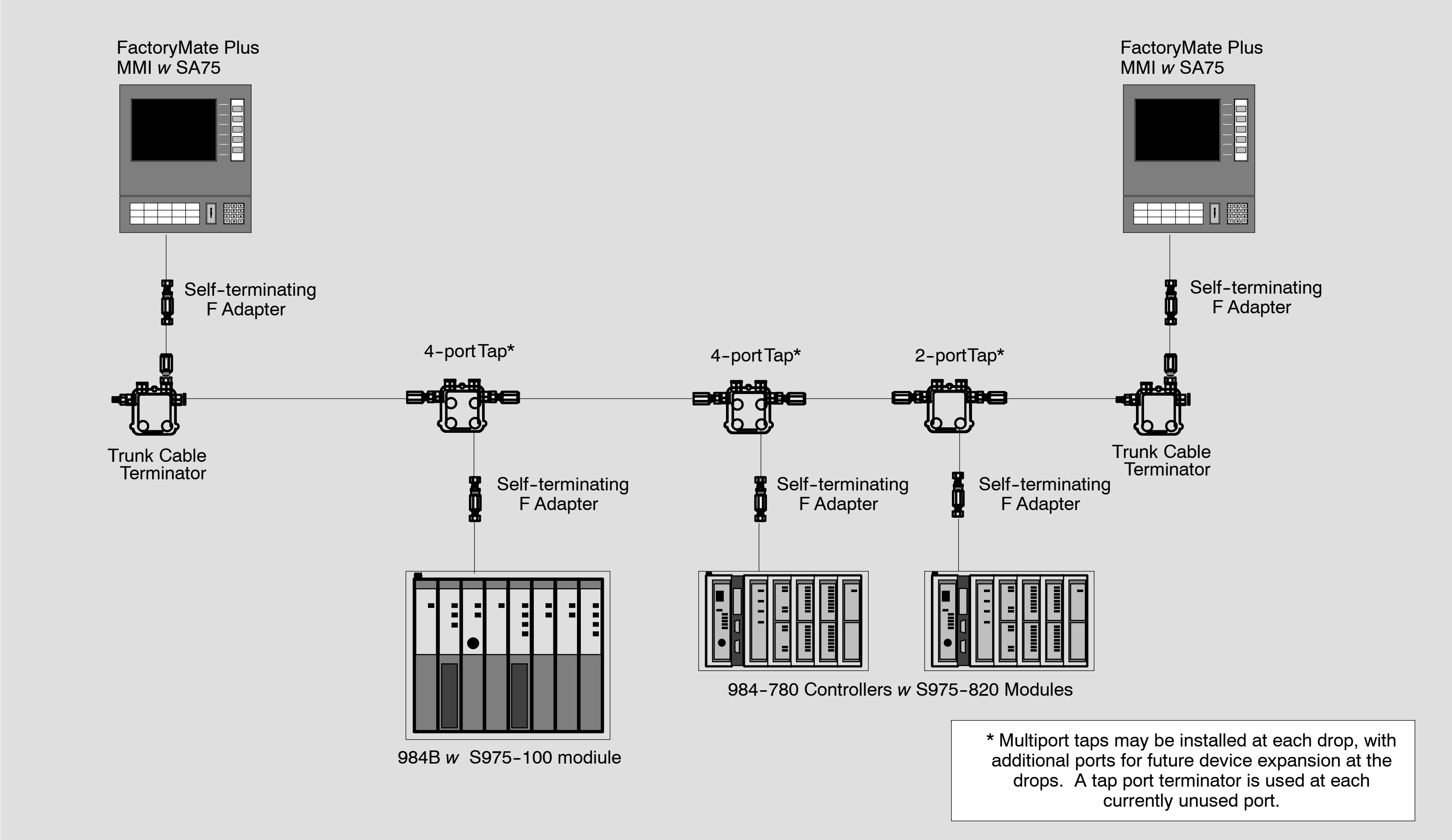
Modbus II communications are conducted over the same type of cable media used in MAP networks.

4.9.1 Modbus II Software

Modbus II network applications are programmed using two loadable instructions—MBUS and PEER. MBUS allows your application to read or write registers or discretes across the network. PEER allows you to write registers simultaneously to as many as 16 nodes on the network, providing rapid updating of common application and process values.

Any node on the network may initiate data transfers across the network using these two instructions. CRC-32 error checking diagnostics automatically assure you of reliable data transfer.

A Modbus II Network

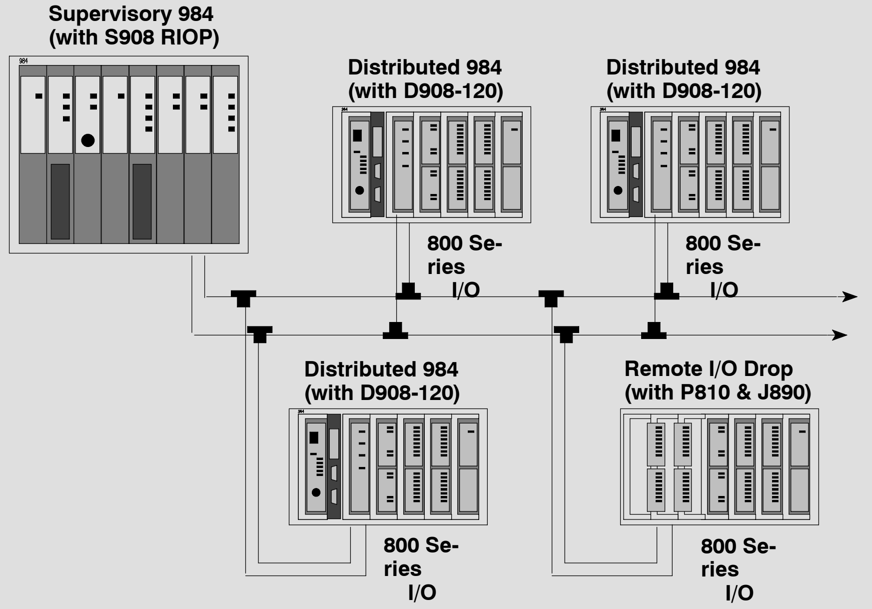


* Multiport taps may be installed at each drop, with additional ports for future device expansion at the drops. A tap port terminator is used at each currently unused port.

4.10 Distributed Control Processing

You can establish a distributed control processing capability using an AS-D908-1*x*0 module in an S908 style of remote I/O communication system. The D908 provides the interface to the high speed (1.5 Mbits/s) communication link. A distributed architecture provides a tightly integrated system that transfers data and control information between the supervisor and the distributed controllers for interlocking and data collection.

A D908 module plugs into an option slot in a *distributed* 984-68*x* or -78*x* Controller. It communicates over the coaxial link with an S908 (or S929) RIO Processor in the *supervisor*. Up to 32 distributed controllers may be linked to the supervisory controller, depending on that supervisor's RIO capabilities.

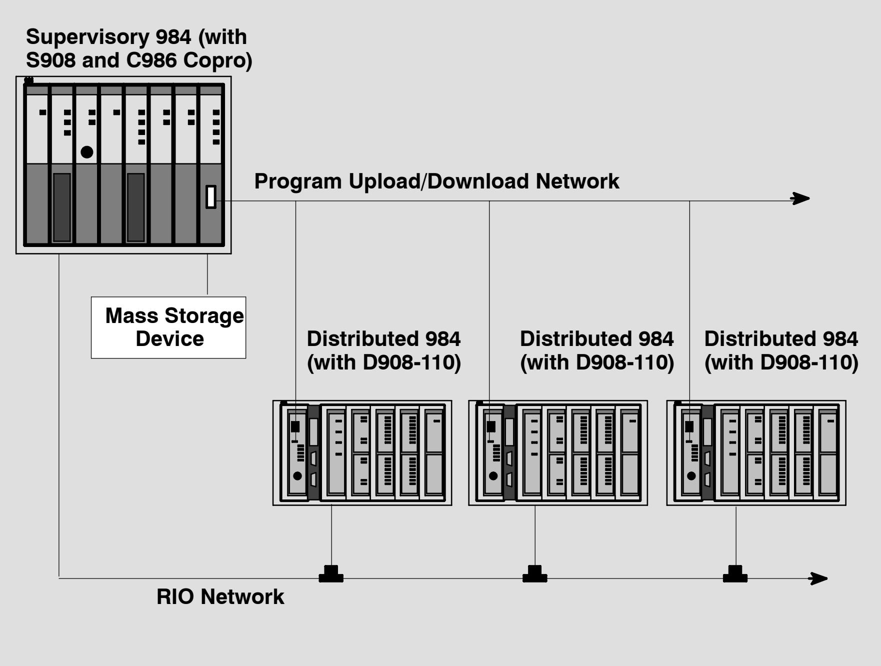


The supervisory controller sees the distributed controller as a J890 I/O drop with input and output addresses Traffic Copped to it. A special D908 Traffic Cop screen is used in the panel software.

Distributed processing means that system control development can be broken up into smaller programs at individual distributed stations while the supervisor controls the interlocking and collects the process information. Smaller programs mean better throughput and easier troubleshooting.

4.10.1 Distributed Control Applications

Distributed processing systems are well suited to transfer line control and material handling applications. In certain cell applications, a supervisory 984 controller with a C986 Coprocessor can act as the cell controller, doing data collection, data logging, and program uploading/downloading and archiving; when process changes are required, new data can be downloaded via the D908s to quickly change parameters and resume production:



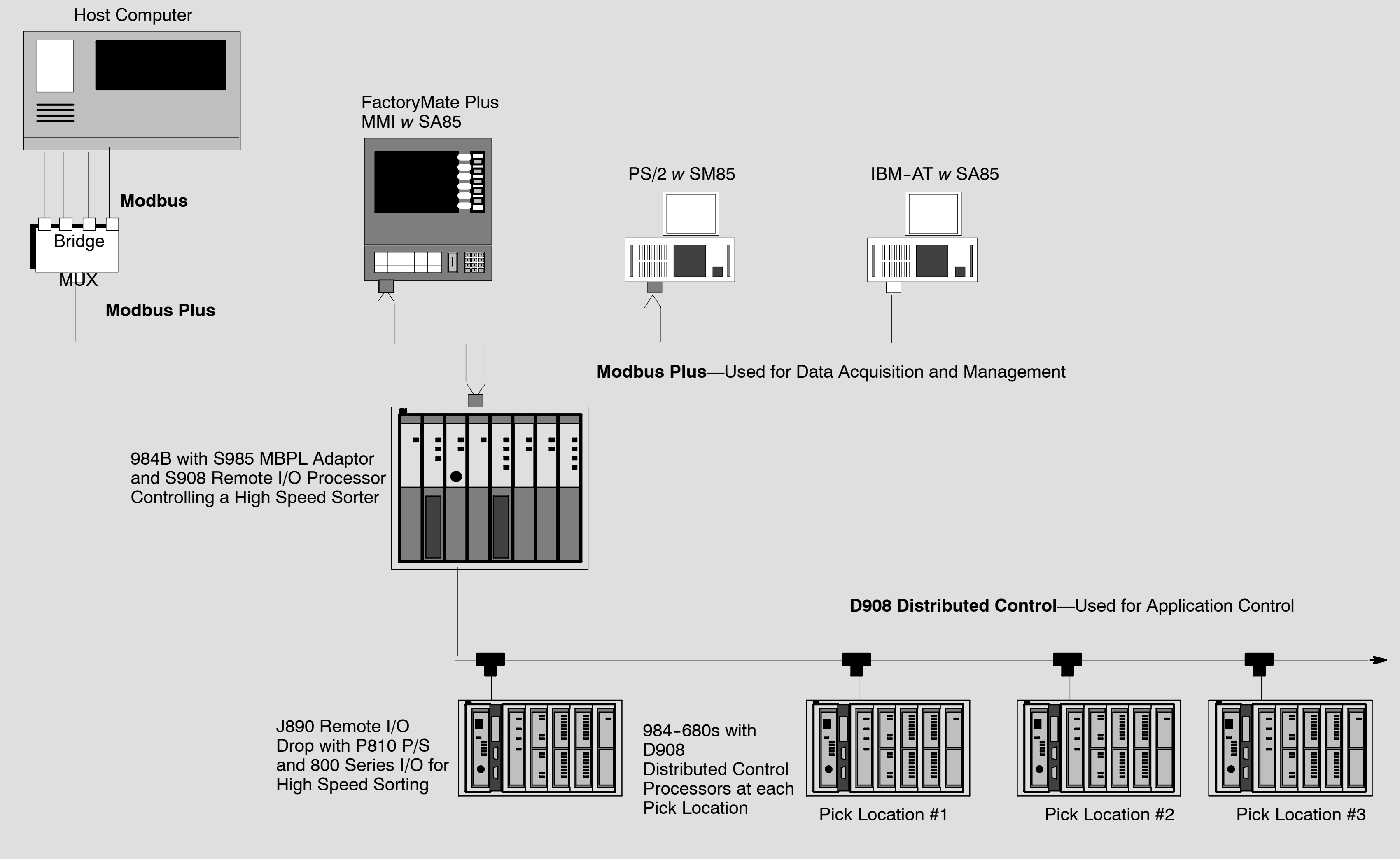
4.11 Network Topology Overview

The illustration on the following page shows, in simplified form, how multiple networks types may be interconnected in a 984 control system. It shows networked hierarchy for controlling a material handling environment.

A D908-based distributed processing is used to link a string of 984-680 Controllers at pick locations along with a standard drop of 800 Series I/O for high speed sorting.

Above the distributed network in the control hierarchy is a Modbus Plus network used for data acquisition and management. It Modbus Plus bridge MUX links the Modbus Plus network via a Modbus interface to the host computer that resides at the top of the control hierarchy.

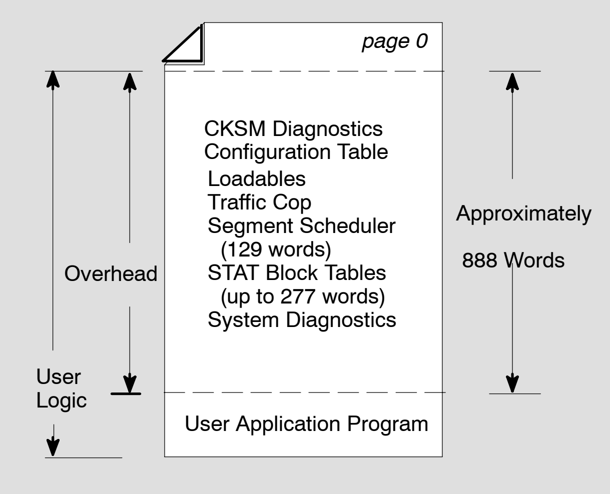
Using Multiple Networks In a Material Handling Environment



Chapter 5
984 Memory Allocation

5.1 User Memory

User memory is the space provided in the controller for your logic program and for system overhead. Optional user memory sizes varying from 1.5K ... 64K words are available, depending on controller type and model. Each word in user memory is stored on page 0 in the controller's memory structure; words may be either 16 or 24 bits long, depending on the controller's CPU size.



5.1.1 System Overhead

System overhead comprises a set of tables that define the system's size, structure, and status. Some tables in system overhead have a predetermined amount of memory space allocated to them—for example, the configuration table always contains 128 words and the order-of-solve table (or segment scheduler) always contains 129 words. Other tables, such as the traffic cop, may consume a large but nonpredetermined amount of memory. Optional pieces of system overhead, such as a loadables table, may or may not consume memory depending on the requirements of your application.

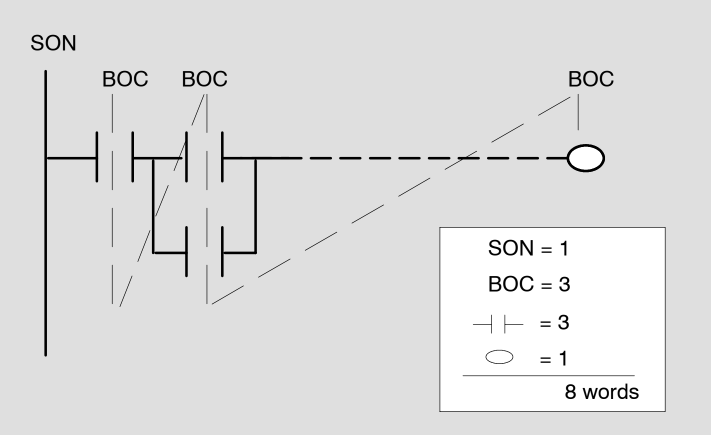
5.1.2 User Logic

The amount of space available for application logic is calculated by subtracting the amount of space consumed by system overhead from the total amount of user logic. System overhead in a relatively conservative system configuration can be expected to consume around 1000 words; system configurations with moderate or large traffic cops will require more overhead.

5.1.3 User Memory Storage

User memory is stored in CMOS RAM. In the event that power is lost, CMOS RAM is backed up by a long-life (typically 12-month) lithium battery.

Ladder logic requires one word of either 16 bit or 24 bit memory to uniquely identify each node in an application program. Contacts and coils each occupy one node, and therefore one word. Function blocks, which usually comprise two or three nodes, require two or three words, respectively. Other elements that control program scanning—start of a network (SON), beginning of a column (BOC), and horizontal shorts—use one word of user logic memory as well. (A vertical short does not use any user logic memory words.)



5.2 State RAM Values

As part of the 984 configuration process (using the Configurator editor in the panel software), you will specify a certain number of discrete outputs (or coils), discrete inputs, input registers, and holding registers available for application control. These inputs and outputs are placed in a table of 16-bit words in an area of system memory called *state RAM*.

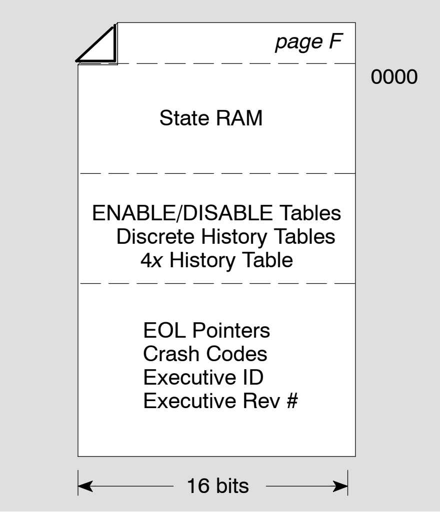
5.2.1 A Referencing System for Inputs and Outputs

The system displays the various types of inputs and outputs using a reference numbering system. Each reference number has a leading digit that identifies its data type followed by a string of digits that defines it unique location in state RAM:

0*x*	A *discrete output* (*or coil*). It can be used to drive a real output through an output module or to set one or more internal coils in State RAM. A specific 0*x* reference may be used only once *as a coil* in a logic program; its status may be used multiple times to drive contacts.
1*x*	A *discrete input*. Its ON/OFF status is controlled by an input module. It can be used to drive contacts in the logic program.
3*x*	An *input register*. This register holds numerical inputs from an external source—for example, a thumbwheel entry, an analog signal, or data from a high speed counter. A 3*x* register can hold 16 consecutive discrete signals, which may be entered into the register in either binary or binary coded decimal (BCD) format.
4*x*	An *output* (*holding*) *register*. It may be used to store numerical (decimal or binary) information in State RAM or to send the information to an output module.
6*x*	Used to store binary information in extended memory area—available only in the 984B Controller (see Chapter 16).

5.2.2 How Discrete and Register Data Are Stored in State RAM

State RAM data are always 16 bit words and are stored on page F in System Memory. The state RAM table is followed immediately by a discrete history table that stores the state of the bits at the end of the previous scan, and by a table of the current ENABLE/DISABLE status of all the discrete (0*x* and 1*x*) values in state RAM.

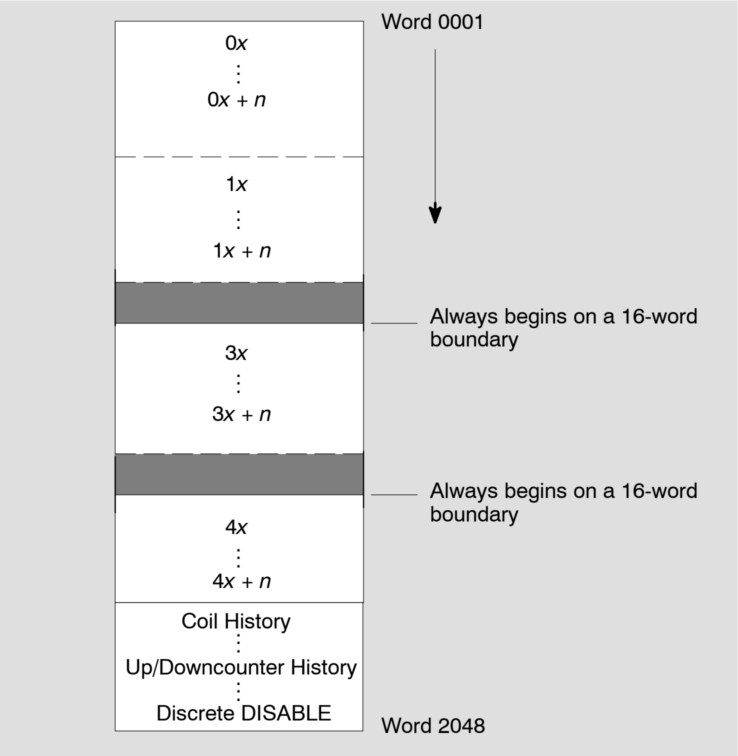


Each 0*x* or 1*x* value implemented in user logic is represented by one bit in a word in state RAM, by a bit in a word in the history table, and by a bit in a word in the DISABLE table. In other words, for every discrete word in the state RAM table there is one corresponding word in the history table and one corresponding word in the DISABLE table.

Counter input states for the previous scan are represented on page F in an upcounter/downcounter history table. Each counter register is represented by a single bit in a word in the table; a value of 1 indicates that the top input was ON in the last scan, and a value of 0 indicates that the top input was OFF in the last scan.

5.3 State RAM Structure

Words are entered into the state RAM table from the top down in the following order:



The discrete words come first in the top-down entry procedure, first the 0*x* words followed immediately by the 1*x* words. The register values follow; the blocks of 3*x* and 4*x* register values must each begin at a word that is a multiple of 16. For example, if you allocate five words for eighty 0*x* references and five words for eighty 1*x* references (5 words x 16 bits/word = 80), you have used words 0001 ... 0010. Words 0011 ... 0016 are then left empty so that the first 3*x* reference begins at word 0017.

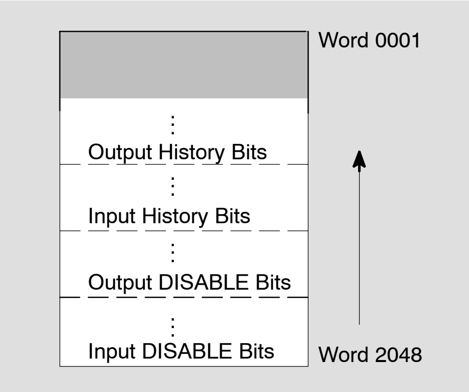
5.3.1 The Required Minimum State RAM Values

In a *minimum configuration*, you must allocate:

- ☐ 48 0*x* discrete references—three words (in MODSOFT);
  16 0*x* discrete references—one word (in P190/P190 emulation software)
- ☐ 16 1*x* discrete references—at least one word
- ☐ One 3*x* register reference—one word
- ☐ Three 4*x* register references—three words (in MODSOFT);
  One 4*x* register reference—one word (in P190/P190 emulation software)

5.3.2 Storing History and Disable Bits for Discrete Values

For each discrete word allocated in state RAM, two words are allocated in the history/disable tables, which follow the state RAM table on page F in system memory. The history/disable tables are generated from the bottom up in the following manner:



5.4 The Configuration Table

The configuration table is one of the key pieces of overhead contained in system memory. It comprises 128 consecutive words and provides a means of accessing information defining your control system capabilities and your user logic program.

With your programming panel software, you can access the configurator editor, which allows you to specify the configuration parameters—such as those shown on the following page—for your control system.

⚠ **Caution When you make a change in an existing 984 configuration table and write the change to system memory, you may erase your ladder logic, traffic cop, and ASCII message table. This may occur if you change the number of:**

- **Discrete inputs**
- **Discrete outputs**
- **Input registers**
- **Holding registers**
- **I/O drops**
- **I/O modules**
- **Logic segments**
- **Modbus ports**
- **ASCII messages**
- **Total ASCII message words**

Back up your application program and ASCII messages before writing the new configuration information. Reenter your traffic cop, then relocate the backed up logic and ASCII message table to the newly configured system memory.

When a controller's memory is empty—in a state called DIM AWARENESS—you are not able to write a traffic cop or a user logic program. Therefore, the first programming task you must undertake with a new controller is to write a valid configuration table using your configurator editor.

5.4.1 Assigning a Battery Coil

A 0*x* coil can be set aside in the configuration to reflect the current status of the controller's battery backup system. If this coil has been set and is queried, it displays a discrete value of either 0, indicating that the battery system is healthy, or 1, indicating that the battery system is not healthy.

5.4.2 Assigning a Timer Register

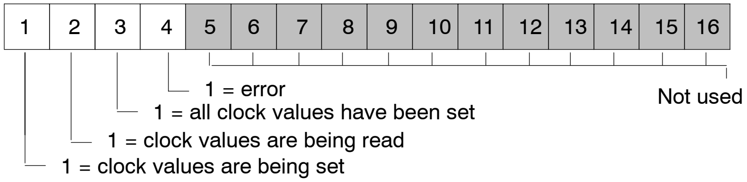
A 4*x* register can be set aside in the configuration as a synchronization timer. It stores a count of clock cycles in 10 ms increments. If this register is set and queried, it displays a free-running value that ranges from 0000 to FFFF hex with wrap-around to 0000.

☞ **Note** *If you are doing explicit address routing in bridge mode on a Modbus Plus network*, the location of the explicit address table in the configuration is dependent on the timer register address—i.e., a timer register must be assigned in order to create the explicit address table. The explicit address table can consist of from 0 ... 10 blocks, each block containing five consecutive 4*x* registers. The address of first block in the explicit address table begins with the 4*x* register immediately following the address assigned to the timer register. Therefore, when you assign the timer register, you must choose a 4*x* register address that has the next 5 ... 50 registers free for this kind of application.

5.4.3 The Time of Day Clock

When a 4*x* holding register assignment is made in the configurator for the time of day (TOD) clock, that register and the next seven consecutive registers (4*x* ... 4*x + 7)* are set aside in the configuration to store TOD information. The block of registers is implemented as follows:

4*x* The control register:



4*x* + 1	Day of the week (Sunday = 1, Monday = 2, etc.)
4*x* + 2	Month of the year (Jan. = 1, Feb. = 2, etc.)
4*x* + 3	Day of the month (1 ... 31)
4*x* + 4	Year (00 ... 99)
4*x* + 5	Hour in military time (0 ... 23)
4*x* + 6	Minute (0 ... 59)
4*x* + 7	Second (0 ... 59)

For example, if you configured register 40500 for your TOD clock, set the bits appropriately as shown above, then read the clock values at 9:25:30 on Tuesday, July 16, 1991, the register values displayed in decimal format would read:

```
40500              0110000000000000
40501                  3 Dec
40502                  7 Dec
50503                  16 Dec
40504                  91 Dec
40505                  9 Dec
40506                  25 Dec
40507                  30 Dec
```

Configuration Data Overview

	Data Type	Format	Default Setting	Notes and Exceptions
Configuration Size				
	# of coils	Even multiple of 16	16	
	# of discrete inputs	Even multiple of 16	16	
	# of register outputs		01	
	# of register inputs		01	
	# of I/O drops	Up to 32, depending on controller type	01	Used only when I/O is configured in drops.
	# of I/O modules	Up to 1024, depending on controller type	00	Not displayed by editor; used by system to calculate Traffic Cop words.
	# of logic segments	Generally equal to # of drops	00	Add one additional segment for subroutines.
	# of I/O channels	Even number from 02 ... 32	02	Used only when I/O is configured in channels.
	Memory size	32K or 64K	32K	64K can be used only on a 984B Controller.
Modbus (RS-232C) Port Parameters				
	Communication mode	ASCII or RTU	RTU	
	Baud rate	50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200	9600	
	Parity	ON/OFF; EVEN/ODD	ON/EVEN	
	Stop bit(s)	1 or 2	2	
	Device addresss	001-247	001	
	Delay time (in ms)	01-20 (representing 10-200 ms)	01 (10 ms)	
ASCII Message Table				
	# of messages	Up to 9999	00	If your controller doesn't support remote I/O, it cannot support ASCII devices.
	Size of message area	Decimal > 0 < difference between memory size (32K or 64K) and sys. overhead (1 word = 2 ASCII characters)	00	
	# of ASCII ports	Two per drop, up to 32	00	
	ASCII port parameters	Baud rate	1200	
		Parity	ON/EVEN	
		# of stop bits	01	
		# of data bits per character	08	
		Presence of a keyboard	NONE	
	Simple ASCII input	A 4*x* value representing the first of 32 registers for simple ASCII input	NONE	Only a 984B Controller supports simple ASCII input.
	Simple ASCII output	A 4*x* value representing the first of 32 registers for simple ASCII output	NONE	Only 984A and 984B Controllers support simple ASCII output.
Special Functions				
	SKIP functions allowed	YES/NO	NO	
	Battery coil	A 0*x* reference reflecting the status of battery backup system	00000	Once a battery coils is placed in a Configuration Table, it cannot be removed.
	Timer register	A 4*x* register set aside to hold a number of 10 ms clock cycles	NONE	
	TOD clock	A 4*x* register, the first of eight reserved for time-of-day values	NONE	
Loadables Instructions				
	Install loadable	PROCEED or CANCEL		Various 984 controllers support different kinds of loadable instruction sets. Make sure that your loadables and controller are compatible.
	Delete loadable(s)	DELETE ALL, DELETE ONE, CANCEL		
Writing Configurator Data to System Memory				
	Write data as specified	PROCEED or CANCEL	NONE	PROCEED will overwrite any previous Configuration Table data.

5.5 The Traffic Cop Table

Just as a programmable controller needs to be physically linked to I/O modules in order to become a working control system, the references in user logic need to be linked in the system architecture to the signals received from the input modules and sent to the output modules. The traffic cop table provides that link.

5.5.1 Determining the Size of the Traffic Cop Table

The traffic cop directs data flow between the input/output signals and the user logic program; it tells the controller how to implement inputs in user logic and provides a pathway down which to send signals to the output modules. The traffic cop table, which is stored on page 0 in system memory, consumes a large but not predetermined amount of system overhead. Its length is a function of the number of discrete and register I/O points your system has implemented and is defined by the type of I/O modules you specify in the configuration table. The *minimum* allowable size of the traffic cop table is nine words.

5.5.2 Writing Data to the Traffic Cop Table

With your programming panel software, you can access a traffic cop editor that allows you to define:

- The number of drops in the 984 I/O system
- The number of discretes/registers that may be used for input and output
- The number, type, and slot location of the I/O modules in the drop
- The reference numbers that link the discretes/registers to the I/O modules
- Drop hold-up time for each I/O drop
- ASCII port addresses (if used) for any drop

Chapter 6
984 Opcode Assignments

- ☐ Translating Ladder Logic Elements in the System Memory Database
- ☐ Translating DX Functions in the System Memory Database
- ☐ Opcode Assignments for Other Functions
- ☐ Extra Opcodes Available in 24 Bit CPUs

6.1 Translating Ladder Logic Elements in the System Memory Database

A 984 automatically translates symbolic ladder elements and function blocks into database nodes that are stored on page 0 in system memory. A node in ladder logic is a 16 or 24 bit word—an element such as a contact translates into one database node, while an instruction such as an ADD block translates into three database nodes.

The database format differs for 16 bit and 24 bit nodes:

16 BIT NODE FORMAT

x	*x*	*x*	*x*	*x*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*z*	*z*	*z*

24 BIT NODE FORMAT

x	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*y*	*z*	*z*	*z*

The five most significant bits in a 16 bit node and the eight most significant bits in a 24 bit node—the *x* bits—are reserved for *opcodes*. An opcode defines the type of functional element associated with the node—for example, the code 01000 specifies that the node is a normally open contact, and the code 11010 specifies that the node is the third of three nodes in a multiplication function block.

6.1.1 Translating Logic Elements and Non-DX Functions

When the system is translating standard ladder logic elements and non-DX function blocks, it uses the remaining (*y* and *z*) bits as *pointers* to register or bit locations in State RAM associated with the discretes or registers used in your ladder logic program.

With a 16 bit node, 11 bits are available as state RAM pointers, giving you a total addressing capability of 2048 words. The maximum number of configurable registers in most 16 bit machines is 1920, with the balance occupied by up to 128 words (2048 bits) of discrete reference, disable, and history bits. An exception is the 984-680/-685 Controllers, which have an extended registers option that supports 4096 registers in state RAM.

With a 24 bit node, 16 bits are available as state RAM pointers. The maximum number of configurable registers in a 24 bit machine is 9999.

Opcodes are generally expressed by their hex values:

Opcodes for Standard Ladder Logic Elements and Non-DX Instructions

16 Bit Nodes (Binary)	24 Bit Nodes (Binary)	(Hex)	Ladder Logic Element/Instruction
00000	00000000	00	Beginning of a column in a network
00001	00000001	01	Beginning of a column in a network
00010	00000010	02	Beginning of a column in a network
00011	00000011	03	Beginning of a column in a network
00100	00000100	04	Start of a network
00101	00000101	05	I/O exchange/End-of-Logic
00110	00000110	06	Null Element
00111	00000111	07	Horizontal short
01000	00001000	08	Normally open contact
01001	00001001	09	Normally closed contact
01010	00001010	0A	Positive transitional contact
01011	00001011	0B	Negative transitional contact
01100	00001100	0C	Nonretentive coil
01101	00001101	0D	Retentive coil
01110	00001110	0E	Constant quantity skip function
01111	00001111	0F	Register quantity skip function
10000	00010000	10	Constant value storage
10001	00010001	11	Register reference
10010	00010010	12	Discrete group reference
10011	00010011	13	Down counter (DCTR) function
10100	00010100	14	Up counter (UCTR) function
10101	00010101	15	One second timer (T1.0) function
10110	00010110	16	0.1 second timer (T0.1) function
10111	00010111	17	0.01 second timer (T.01) function
11000	00011000	18	Add (ADD) math function
11001	00011001	19	Subtract (SUB) math function
11010	00011010	1A	Multiply (MULT) math function
11011	00011011	1B	Divide (DIV) math function

☞ **Note** The opcodes for these standard ladder logic elements and instructions are hard-coded in the system firmware, and they cannot be altered.

6.2 Translating DX Functions in the System Memory Database

6.2.1 How the x and z Bits Are Used in 16 Bit Nodes

When you are using a 16 bit CPU, you are left with only four more x bit combinations—11100, 11101, 11110, and 11111—with which to express opcodes for 18 DX functions. To gain the necessary bit values, the system uses the three least significant (*z*) bits along with the x bits to express the opcodes:

16 Bit Node Format for DX Functions

1	1	1	0	0									*z*	*z*	*z*

0 0 0 = R→T
0 0 1 = T→R
0 1 0 = T→T
0 1 1 = BLKM
1 0 0 = FIN
1 0 1 = FOUT
1 1 0 = SRCH
1 1 1 = STAT

1	1	1	0	1									*z*	*z*	*z*

0 0 0 = AND
0 0 1 = OR
0 1 0 = CMPR
0 1 1 = SENS
1 0 0 = MBIT
1 0 1 = COMP
1 1 0 = XOR
1 1 1 = BROT

1	1	1	1	0									*z*	*z*	*z*

0 0 0 = READ
0 0 1 = WRIT

For Loadable Options {
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

6.2.2 How the x and z Bits Are Used in 24 Bit Nodes

In the 24 bit CPUs, the three most significant x bits are used to indicate the type of DX function:

24 Bit Node Format for DX Functions

x	x	x	1	1	1	0	0									z	z	z

0	0	0	=	R→T	0	0	0
0	0	1	=	T→R	0	0	1
0	1	0	=	T→T	0	1	0
0	1	1	=	BLKM	0	1	1
1	0	0	=	FIN	1	0	0
1	0	1	=	FOUT	1	0	1
1	1	0	=	SRCH	1	1	0
1	1	1	=	STAT	1	1	1

x	x	x	1	1	1	0	1									z	z	z

0	0	0	=	AND	0	0	0
0	0	1	=	OR	0	0	1
0	1	0	=	CMPR	0	1	0
0	1	1	=	SENS	0	1	1
1	0	0	=	MBIT	1	0	0
1	0	1	=	COMP	1	0	1
1	1	0	=	XOR	1	1	0
1	1	1	=	BROT	1	1	1

x	x	x	1	1	1	1	0									z	z	z

0	0	0	=	READ	0	0	0
0	0	1	=	WRIT	0	0	1
0	1	0	}	For Loadable Options	0	1	0
0	1	1			0	1	1
1	0	0			1	0	0
1	0	1			1	0	1
1	1	0			1	1	0
1	1	1			1	1	1

The z bits, which simply echo the three most significant x bits, may be ignored in the 24 bit nodes.

Opcode Representations for Standard 984 DX Functions

Binary	Hexadecimal	DX Instruction
00011100	1C	R→T
00111100	3C	T→R
01011100	5C	T→T
01111100	7C	BLKM
10011100	9C	FIN
10111100	BC	FOUT
11011100	DC	SRCH
11111100	FC	STAT
00011101	1D	AND
00111101	3D	OR
01011101	5D	CMPR
01111101	7D	SENS
10011101	9D	MBIT
10111101	BD	COMP
11011101	DD	XOR
11111101	FD	BROT
00011110	1E	READ
00111110	3E	WRIT
01111110	7E	XMWT*
10011110	9E	XMRD*

* XMWT and XMRD are used for extended memory capabilities available only in the 984B chassis mount Controller. They are not installed in other 24 bit controllers.

☞ **Note** The opcodes for these standard ladder logic elements and instructions are hard-coded in the system firmware, and they cannot be altered.

6.2.3 How the y Bits are Utilized for DX Functions

The *y* bits in a database node holding DX function data contain a binary number that expresses the number of registers being transferred in the function.

A 16 bit database node has eight y bits. A 16 bit CPU is, therefore, machine limited to no more than 255 transfer registers per DX operation.

A 24 bit database node has 13 y bits. A 24 bit CPU is, therefore, capable of reaching a theoretical machine limit of 8191 transfer registers per DX operation; practically, however, the greatest number of transfer registers allowed in a 24 bit DX operation is 999.

6.3 Opcode Assignments for Other Functions

Several 984 controllers have additional instructions in their System Executive. These instructions use the following opcodes:

Opcode Representations for Other Executive Instructions

Binary	Hexadecimal	Instruction
01011110	5E	PID2
11011110	DE	JSR
10111110	BE	LAB
11111110	FE	RET
01111111	7F	EMTH
10011111	9F	BLKT
10111111	BF	CKSM or MSTR*
11011111	DF	TBLK

*MSTR and CKSM share the same opcode and are mutually exclusive EPROM-based instructions. MSTR is included in the Executive of any 984 controller that employs Modbus Plus, and the CKSM instruction is not included on these Executives. CKSM is provided in several 984 controllers that do not implement Modbus Plus.

☞ **Note** If your controller contains these additional functions in its System Executive, the opcodes are hard-coded in the system firmware, and they cannot be altered.

The PID2, BLKT, TBLK, MSTR, and CKSM instructions are also available as *loadable* instructions for some 984 controllers (when a controller does not support these functions in any version of its Executive firmware). The loadable versions of these instructions are assigned the same opcodes.

Various ladder logic instructions are available only in loadable software packages. When instructions are loaded to a controller, they are stored in RAM on page 0 in system memory. They are not resident on the EPROM. The loadable functions have the following opcodes:

Opcode Representations for 984 Loadable Instructions

Binary	Hexadecimal	Loadable Instruction
11111111	FF	HSBY
01011111	5F	CALL, FN*xx*, or EARS (non-chassis mount)
00011111	1F	MBUS
00111111	3F	PEER
11011110	DE	DMTH
10111110	BE	MATH or EARS (for chassis mount only)
11111110	FE	DRUM
01111111	7F	ICMP

☞ **Note** No two instructions with the same opcode can coexist on a controller. As you can see, several loadables have conflicting opcodes. ICMP is also in conflict with EMTH, DMTH is in conflict with JSR, DRUM is in conflict with RET, and MATH is also in conflict with LAB.

6.3.1 How to Handle Opcode Conflicts

The easiest way to stay out of trouble is to never employ two loadables with conflicting opcodes in your user logic. If you are using MODSOFT panel software, it allows you to change the opcodes for loadable instructions. The *lodutil* utility in the Modicon Custom Loadable Software package (SW-AP98-GDA) also allows you to change loadable opcodes—this software package is not available for all 984 controllers (see Section 21.1).

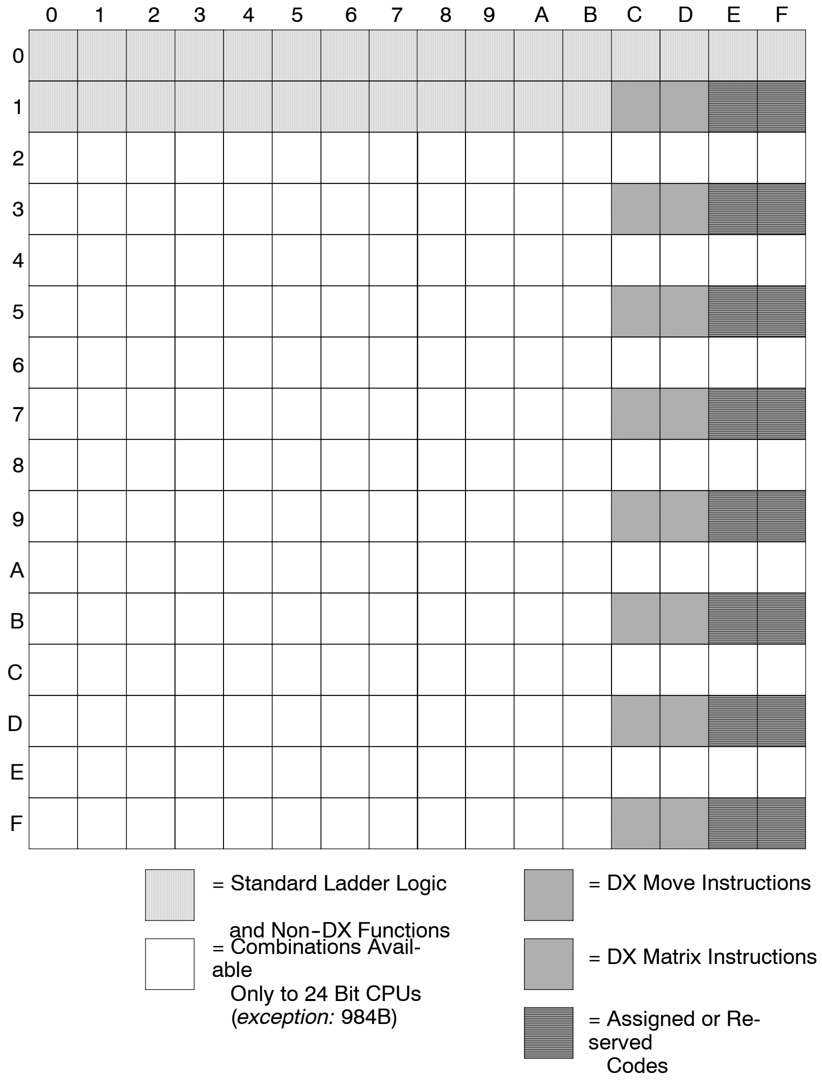
⚠ **Caution If you modify any loadables so that their opcodes are different from the ones shown in this chapter, you must use caution when porting user logic to or from your controller. The opcode conflicts that can result may hang up the target controller or cause the wrong function blocks to be executed in ladder logic.**

☞ **Note** Remember that no opcodes residing in EPROM firmware can be modified.

6.4 Extra Opcodes Available in 24 Bit CPUs

Because the 24 bit CPUs provide eight x bits per node, 2^8 (256) combinations are available for opcode assignments. The 984B chassis mount Controller is the exception—it is design-limited to the x-bit assignments described in Section 6.2.2 in order to enforce conformance with the 16 bit CPUs. The other 24 bit CPUs—e.g., the 984-780/-785, the Q984—can use all opcodes in the hexadecimal range 00 ... FF for loadables and user-defined function blocks.

The matrix on the following page shows how the opcode assignments, indicating which codes are reserved, which codes may be flexibly assigned in either 16 bit or 24 bit CPUs, and which are available for 24 bit CPUs only:



☞ **Note** If you assign an opcode to an instruction and that opcode is a combination available only to a 24 bit CPU, any programs you create using that instruction cannot be ported to a 16 bit CPU (or to a 984B Controller).

Chapter 7
Ladder Logic Overview

- ❐ The Structure of Ladder Logic
- ❐ Ladder Logic Elements and Standard Instructions
- ❐ Additional Ladder Logic Instructions
- ❐ DX MOVE and DX Matrix Functions
- ❐ How Ladder Logic Is Solved
- ❐ Scan Time
- ❐ How to Measure Scan Time
- ❐ Maximizing Throughput
- ❐ The Order of Solve
- ❐ Using the Segment Scheduler to Improve Critical I/O Throughput
- ❐ Using the Segment Scheduler to Improve System Performance
- ❐ Using the Segment Scheduler to Improve Comm Port Servicing
- ❐ Sweep Functions

7.1 The Structure of Ladder Logic

Ladder logic is a highly graphical, easy-to-use programming language that uses relay-equivalent symbology. Its major structural components are segments, networks, and elements.

7.1.1 Ladder Logic Segments

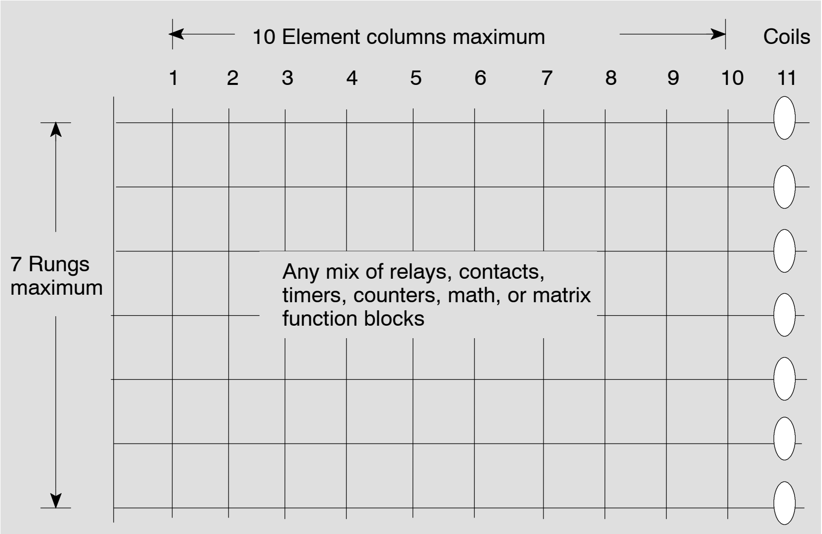
A ladder logic program is a collection of *segments*. As a rule, the number of segments equals the number of I/O drops being driven by the controller, although in many cases there may be more segments than drops (never more drops than segments). A segment is made up of a group of *networks*. There is no prescribed limit on the number of networks in a segment—the size is limited only by the amount of User Memory available and by the maximum amount of time available for the CPU to scan the logic (250 ms).

You can modify the order in which logic is solved with the *segment scheduler*, an editor available with your panel software that allows you to adjust the order-of-solve table in system memory. With some 984 controllers, you may also create an unscheduled segment that contains one or more ladder logic subroutines, which can be called from the scheduled segments via the JSR function.

7.1.2 Ladder Logic Networks

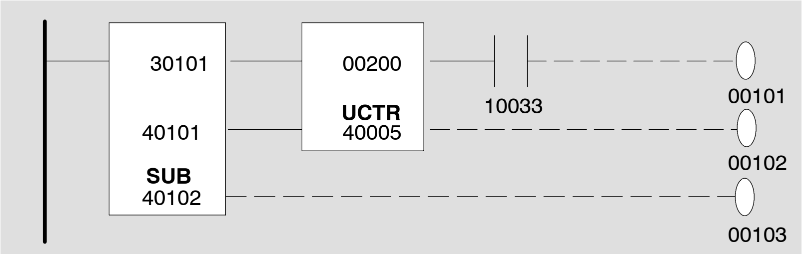
The networks that comprise the ladder logic segment(s) have a clearly defined structure. Each network is a small ladder diagram bounded on the left by a power rail and on the right by a rail which, by convention, is not displayed. Within the rails, the network holds seven rungs (or rows) and eleven columns.

The 77 intersections of the rungs and columns are called *nodes*. Logic *elements* —contacts, coils, horizontal and vertical shorts—and function block instructions are inserted in the nodes of a network. Logic elements and instructions, which are the fundamental building blocks of ladder logic, can occupy the whole 77-node network area or just a portion of it.



In some panel software programming packages, the seven nodes in the 11th column are reserved for displaying coils. If your software treats coil usage this way, then no other logic elements may be displayed in the 11th column, and the remaining 70 nodes may not be used for coils.

Although coils may be automatically displayed in the 11th column, they are not always solved there. The column in which coil 00101 is solved is determined by the position of its controlling logic:



Coil 00103 is solved immediately after the UCTR function block, and coil 00102 is solved immediately after the normally open contact (10033). Coil 00101 is the last coil to be solved in this network.

7.2 Ladder Logic Elements and Standard Instructions

There are six standard one-node ladder logic elements (contacts and coils) in all 984 Controller firmware packages:

Standard One-Node Ladder Logic 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

There are 26 standard (block) instructions available in *all* 984 Controller firmware packages:

Standard Instructions for All 984s

Instruction	Meaning
Counter and Timer Instructions (Two-Node Functions)	
UCTR	Counts up from 0 to a preset value
DCTR	Counts down from a preset value to 0
T1.0	Timer that increments in seconds
T0.1	Timer that increments in tenths of a second
T.01	Timer that increments in hundredths of a second
Calculation Instructions (Three-Node Functions)	
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)	
R→T	Moves register values to a table
T→R	Moves specified 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
FIN	First-in operation to a queue
FOUT	First-out operation from a queue
SRCH	Performs a table search
STAT	Displays status registers from status table in system memory
DX Matrix Instructions (Three-Node Functions)	
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
Skip-Node Instruction (One-Node Function)	
SKP	Skips a specified number of networks in a ladder logic program

7.3 Additional Ladder Logic Instructions

Some special instructions are standard in some 984 controllers but are unavailable in others:

Standard Instructions for Select 984s

Instruction	Meaning
ASCII Communication Instructions (Three-Node Functions) **Standard with All 984s that Support Remote I/O Drops**	
READ	Reads data entered at an ASCII device into 984 Memory
WRIT	Sends a message from the 984 controller to an ASCII device
Ladder Logic Subroutine Instructions (One- and Two-Node Functions) **Standard with Slot Mount and Micro 984s**	
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
Checksum Instruction (Three-Node Function) **Standard on Slot Mount and Micro 984s that Don't Provide Modbus Plus**	
CKSM	Calculates any of four types of checksum operations (CRC-16, LRC, straight CKSM, and binary add)
Network Communication Initiation Instruction (Three-Node Function) **Standard with All 984s that Provide Modbus Plus**	
MSTR*	Specifies a function from a menu of networking operations

* The MSTR block is available in the 984A/B/X chassis mount controllers only as a loadable function, not in firmware.

All standard elements and instructions are stored in the system Executive firmware.

Additional instructions are available for some 984 controllers on an Enhanced Executive PROM:

Enhanced Instructions for Select 984 Controllers

Instruction	Meaning
PID Instruction (Three-Node Function)	
PID2*	Performs a specified proportional-integral-derivative function
Enhanced Math (Three-Node Function)	
EMTH	Performs 38 math operations, including floating point math operations and extra integer math operations such as square root
Enhanced DX Move Instructions (Three-Node Functions)	
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

* The PID2, TBLK, and BLKT blocks are available in the 984A/B/X chassis mount controllers only as loadable functions, not in firmware.

In controllers that offer these instructions as standard features, the instructions are stored in the system Executive firmware.

7.4 DX MOVE and DX Matrix Functions

7.4.1 MOVE Functions

DX MOVE functions copy 16 bit words of data from one memory area to another. The copied data can then be operated on, and the original data remain intact.

A group of consecutive 16 bit registers is called a *table*. The minimum table length is 1—i.e., one word or one register. The maximum table length depends on the DX function and on the type of controller (16 or 24 bit CPU).

Groups of 16 discretes can also be placed in tables. The reference number used is the first discrete in the group, and the other 15 are implied. The number of the first discrete must be of the *first of 16* type—00001, 10001, 00017, 10017, 00033, 10033, ... , etc.

Some DX move functions use a register to indicate which table position the relevant data has been copied from or moved to. This register is called a *pointer*. The pointer value must never exceed the table length. Zero is a valid pointer value, typically indicating that the next operation of the function block will be to copy data from or read data to the first table position. (See examples in Chapter 11.)

7.4.2 Matrix Functions

A *matrix* is a sequence of data bits formed by consecutive 16 bit words derived from tables. DX matrix functions operate on bit patterns within tables.

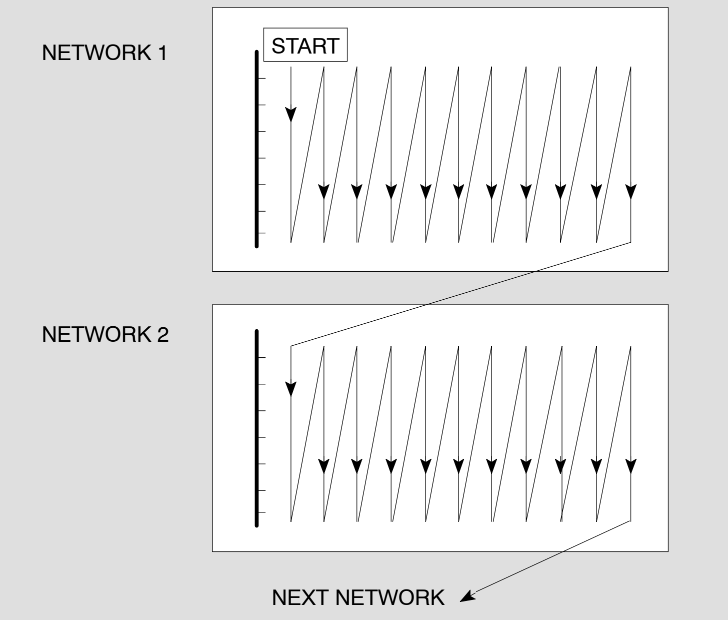
The minimum table length is 1—i.e., one word or one register. The maximum table length depends on the DX function and on the type of controller (16 or 24 bit CPU).

Groups of 16 discretes can also be placed in tables. The reference number used is the first discrete in the group, and the other 15 are implied. The number of the first discrete must be of the *first of 16* type—00001, 10001, 0017, 10017, 00033, 10033, ... , etc. (See examples in Chapter 12.)

7.5 How Ladder Logic Is Solved

The controller's CPU scans the ladder logic program sequentially in this manner:

- ❐ Segments are scanned according to their arrangement in the order-of-solve table—i.e., the *segment scheduler*—in system memory
- ❐ Networks *01* through *nn* within each segment are scanned
- ❐ Nodes within each network are scanned top to bottom, left to right:, in the following manner:



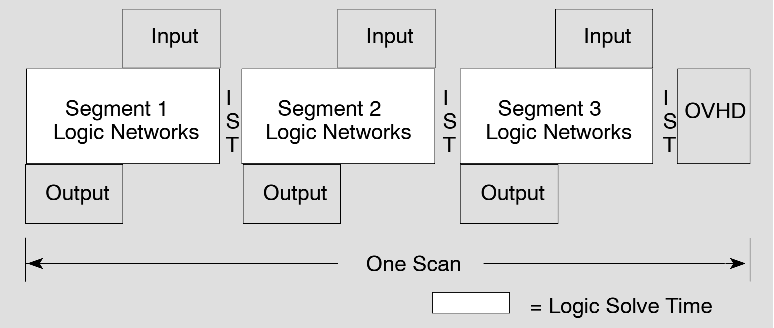
The controller begins solving logic within a network at the top of the leftmost column and proceeds down, then moves to the top of the next column and proceeds down. Each node is solved in the order it is encountered in the logic scan. Power flow within the network is always down each column from left to right, never from bottom to top and never from right to left.

7.6 Scan Time

The time it takes a controller to solve a complete ladder logic program and update all I/O modules is called *scan time*. Scan time comprises the time it takes the 984 controller to solve all scheduled logic—i.e., *logic solve time*, service I/O drops, and perform system overhead—servicing communication ports and option processors, executing intersegment transfer (IST) and system diagnostics.

7.6.1 Logic Solve Time

Logic solve time is the time it takes to solve a complete logic program, independent of the time it takes to service I/O or carry out any system overhead tasks. Logic solve times are different in different types of 984 controllers—the various times, measured in ms/Kwords of logic, are given in the chart in Section 1.2.

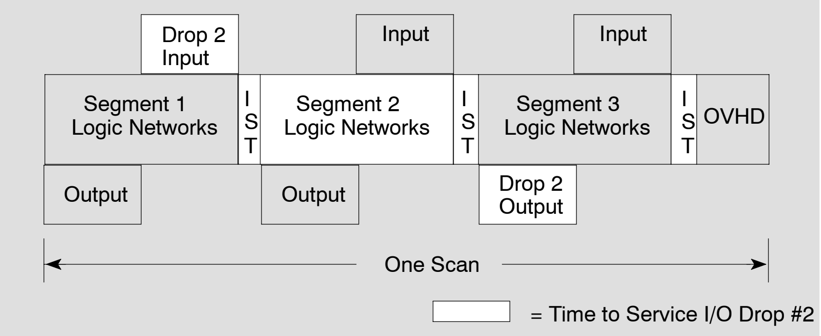


7.6.2 I/O Servicing

In order to optimize system throughput, the 984 control architecture coordinates the solution of ladder logic segments by the controller's CPU with the servicing of I/O drops by the controller's I/O processor. Typically a particular logic segment is coordinated with a particular I/O drop—for example, the logic networks in segment 2 correspond to the real-world I/O points at drop 2. Inputs are read during the previous segment and outputs are written during the subsequent segment.

This method of I/O servicing assures that the most recent input status is available for logic solve and that outputs are written as soon as possible after logic solve.

It ensures predictability between the 984 controller and the process it is controlling.

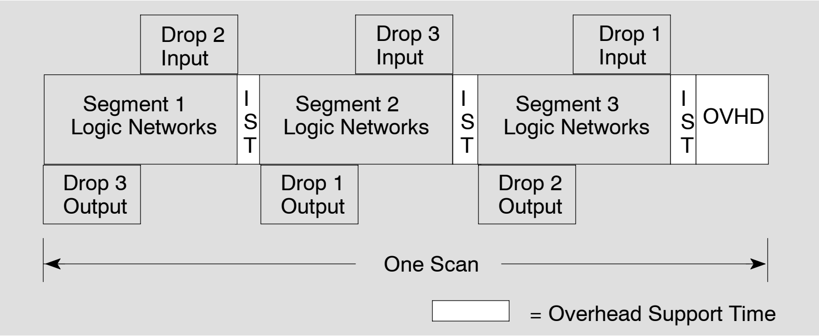


7.6.3 Overhead

An intersegment transfer occurs between each segment, at which time data are exchanged between the I/O processor and the state RAM—previous inputs are transferred to state RAM and the next outputs are transferred to the I/O processor. The logic scan and I/O servicing for each segment are coordinated in this fashion. Using direct memory access (DMA), ISTs typically take less than 1 ms/segment.

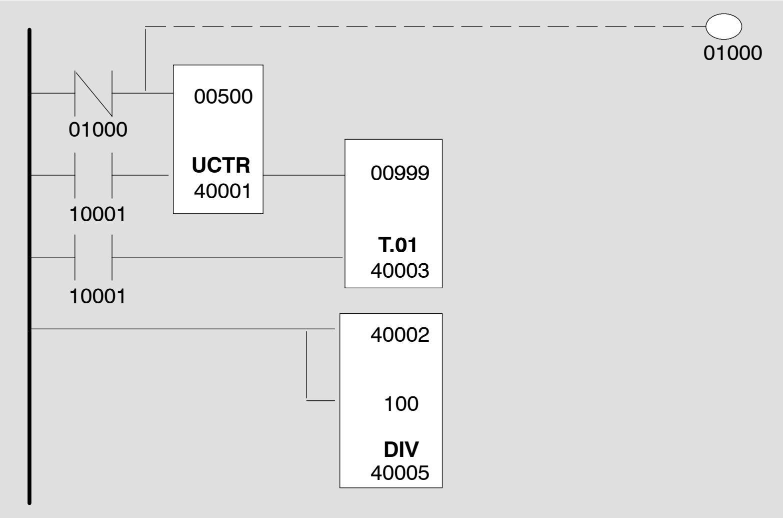
At the end of each scan, *input* messages to the communication ports (Modbus, Modbus Plus, Modbus II) are serviced. The maximum time allotted for comm port servicing is 2.5 ms/scan; typical servicing times are less than 1 ms/scan. If the controller is using any option processors (C986 Coprocessors or D908 Distributed Communications Processors), they are also serviced at the end of each scan and typically require less than 1 ms/scan.

System diagnostics take from 1 ... 2 ms/scan to run, depending on controller type.


Drop 2 Input
Drop 3 Input
Drop 1 Input
Segment 1 Logic Networks
IST
Segment 2 Logic Networks
IST
Segment 3 Logic Networks
IST
OVHD
Drop 3 Output
Drop 1 Output
Drop 2 Output
One Scan
= Overhead Support Time

7.7 How to Measure Scan Time

The following ladder logic circuit may be entered into your program to evalute system scan time:



The upcounter counts 1000 scans as it transitions 500 times. When the counter has transitioned 500 times, the T.01 timer turns OFF and stores the number of hundredths of seconds it has taken for the counter to transition 500 times (1000 scans) in register 40003.

The value stored in 40002/40003 in the DIV block is then divided by 100 and the result—which represents logic solve time in ms—is stored in register 40005.

☞ **Note** 10001 is controlled via a DISABLE or a hard-wired input; if you are running the program in optimized mode, a hard-wired input is required to toggle 10001.

☞ **Note** The maximum amount of time allowed for a scan is 250 ms; if the scan has not completed in that amount of time, a *watchdog timer* in the controller's CPU stops the application and sends a timeout error message to the programming panel display. The maximum limit on scan time protects the controller from entering into infinite loops.

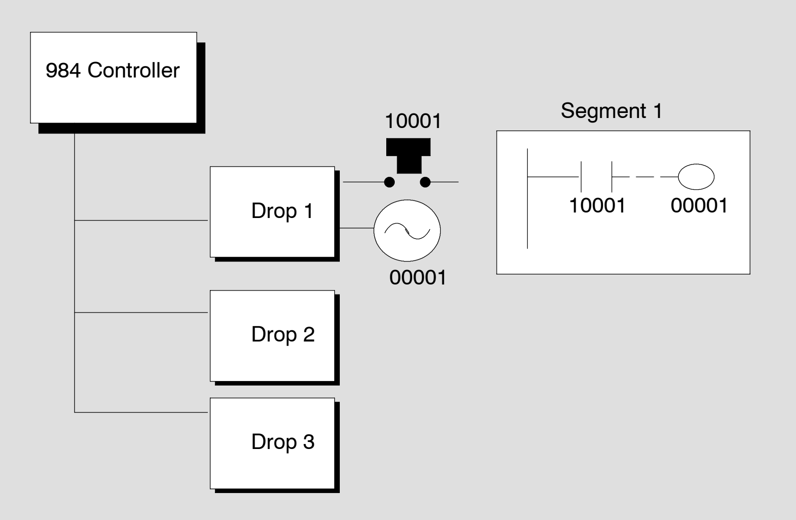
7.8 Maximizing Throughput

The way that the 984 architecture simultaneously solves logic and services I/O drops optimizes system throughput. *Throughput* is the time it takes for a signal received at a field sensing device to be sent as an input to the controller, processed in ladder logic, and returned as an output signal to a field working device. Throughput time may be longer or shorter than a single scan; it gives you a realistic measure of the system's actual performance.

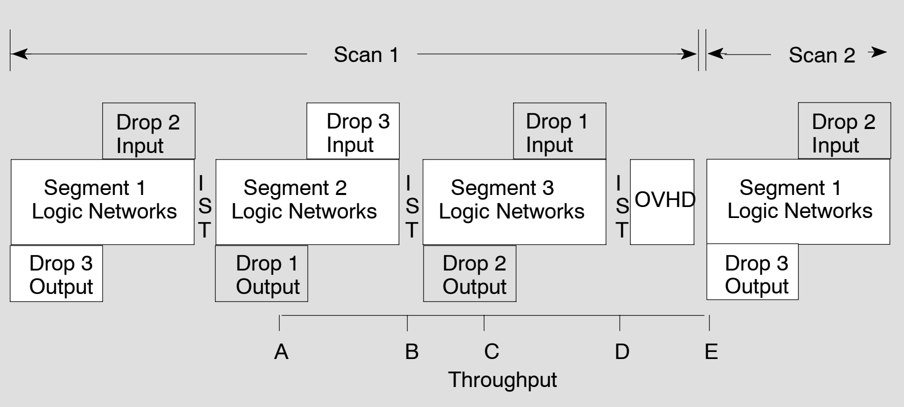
7.8.1 The Ideal Throughput Situation

If the default order-of-solve table is in place, the system automatically solves the logic starting at segment 01 and moving sequentially through segment *nn*. Throughput is optimized when logic referring to real-world I/O is contained in the segment that corresponds to that I/O drop.

For instance, if you are using I/O in drop 1 of a three-drop system to control a pushbutton that starts a motor, the ideal condition is for logic segment 1 to contain all the appropriate logic:



When all logic segments are coordinated with all physical I/O drops in such a manner, the throughput for a given logic segment can be less than one scan:



The illustration above shows the throughput for drop 3—the time beginning with field input data being read by the input modules in drop 3 and ending with the output modules at drop 3 being updated with data from the CPU. Throughput in this best case example is about 75% of total scan time. Five events are shown as drop 3 throughput benchmarks:

- ❐ Event A, where the inputs from drop 3 are available to the I/O processor
- ❐ Event B, where the I/O processor transfers data to state RAM
- ❐ Event C, where the segment 3 logic networks (which correspond to drop 3 I/O) are solved
- ❐ Event D, where data are transferred from state RAM to the I/O processor
- ❐ Event E, where the output data are written to the output modules at drop 3

7.9 The Order of Solve

You specify the number of segments and I/O drops with the configurator editor in your panel software package. The default order-of-solve condition is segment 01 through segment *nn* consecutively and continuously, once per scan, with the corresponding I/O drops serviced in like order. You are able to change the order of solve using the *segment scheduler* editor in your panel software package.

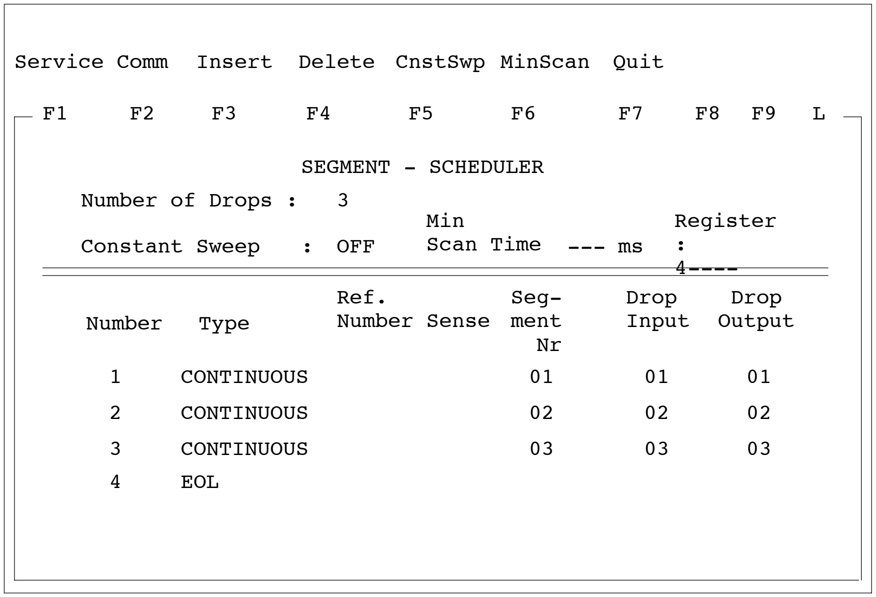
There may be times when you can modify the order of solve to improve overall system performance. The segment scheduler can be used effectively to:

☐ Improve throughput for critical I/O

☐ Improve overall system performance

☐ Optimize the servicing of communication ports

Here is what a standard order-of-solve table might look like, as seen in the MODSOFT segment scheduler editor:



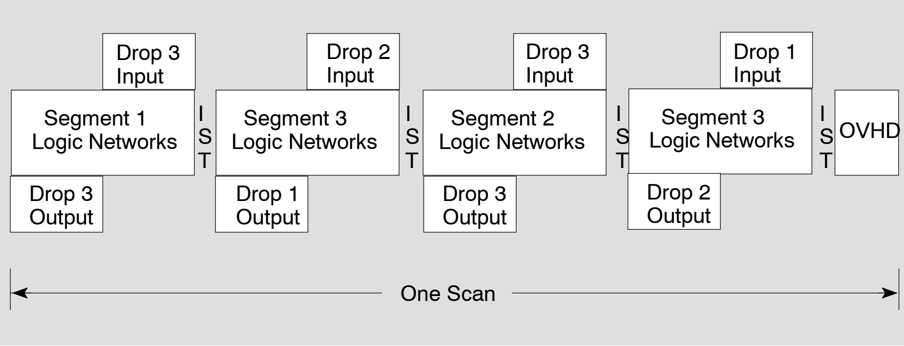
```
Service Comm   Insert  Delete  CnstSwp MinScan  Quit

 F1     F2     F3      F4      F5      F6       F7     F8   F9    L

                      SEGMENT - SCHEDULER

     Number of Drops :   3
                                Min                 Register
     Constant Sweep   :  OFF    Scan Time  --- ms   :
                                                    4----
                         Ref.          Seg-      Drop     Drop
     Number    Type      Number Sense  ment      Input   Output
                                        Nr

       1    CONTINUOUS                 01        01       01

       2    CONTINUOUS                 02        02       02

       3    CONTINUOUS                 03        03       03

       4    EOL
```

A Default Order-of-Solve Table for a Three-Segment Logic Program

7.10 Using the Segment Scheduler to Improve Critical I/O Throughput

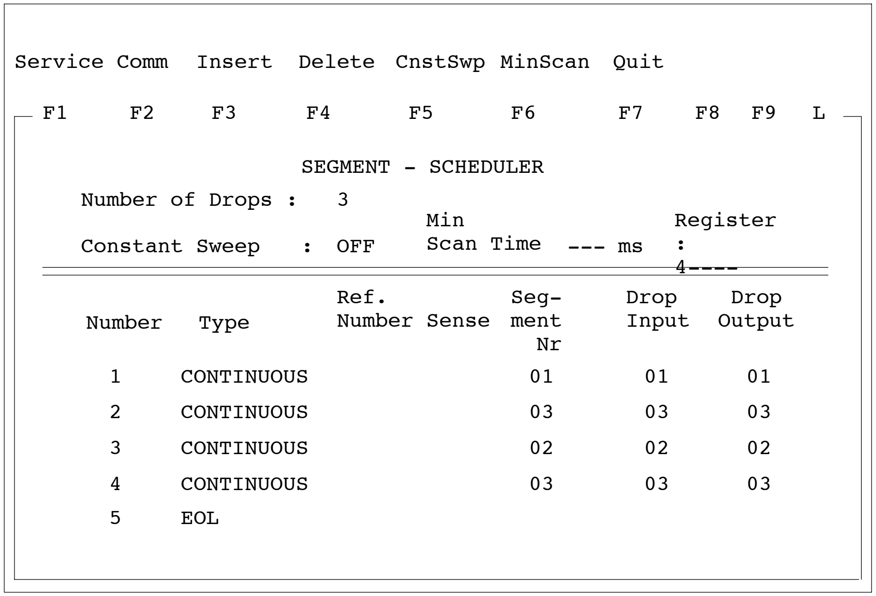
Suppose that your logic program is three segments long and that segment 3 contains logic that is critical to your application—for example, monitoring a proximity switch to verify part presence. Segments 1 and 2 are running noncritical logic such as part count analysis and statistic gathering, The program is running in the standard order-of-solve mode, and you are finding that the controller is not able to read critical inputs with the frequency desired, thereby causing unexceptable system delay.

Using the segment scheduler editor, you can improve the throughput for the critical I/O at drop 3 by scheduling segment 3 to be solved two (or more) times in the same scan:



By rescheduling the order-of-solve table, you actually increase the scan time, but more importantly you improve throughput for the critical I/O supported by logic in segment 3. Throughput is the better measure of system performance.

Here is how the MODSOFT segment scheduler would show the resulting order-of-solve table:



```
Service Comm   Insert  Delete  CnstSwp MinScan  Quit

 F1     F2     F3     F4      F5      F6       F7     F8   F9   L

                      SEGMENT - SCHEDULER
     Number of Drops :   3
                                Min                   Register
     Constant Sweep   :  OFF    Scan Time  --- ms   :
                                                    4----
                                     Ref.        Seg-      Drop     Drop
     Number   Type                   Number Sense ment     Input   Output
                                                   Nr
        1     CONTINUOUS                          01        01       01
        2     CONTINUOUS                          03        03       03
        3     CONTINUOUS                          02        02       02
        4     CONTINUOUS                          03        03       03
        5     EOL
```

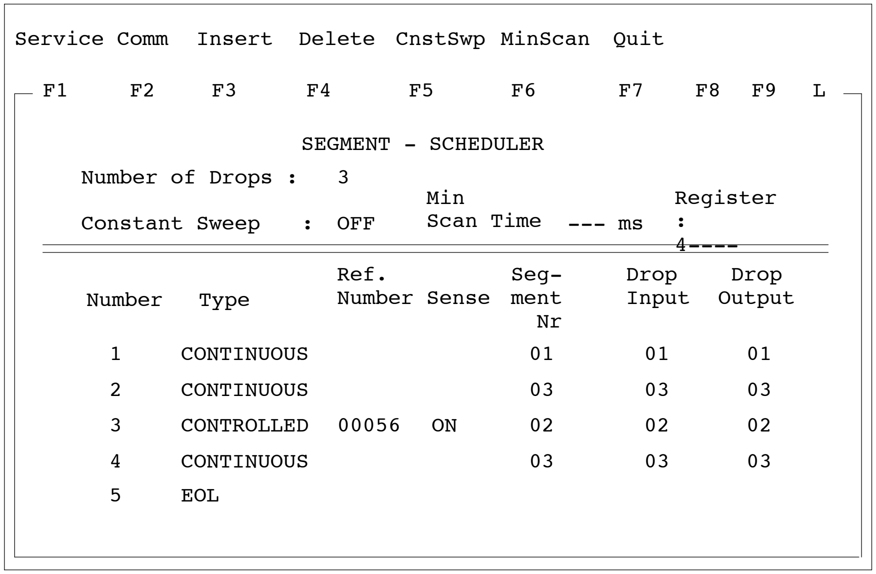
An Order-of-Solve Table Rescheduled for Critical I/O

7.11 Using the Segment Scheduler to Improve System Performance

When certain areas of a ladder logic program do not need to be solved continually on every scan—for example, an alarm handling routine, a data analysis routine, some diagnostic message routines—they can be designated as *controlled segments* by the segment scheduler editor. Based on the status of an I/O or internal reference, a controlled segment may be scheduled to be skipped, thereby reducing scan time and improving overall system throughput.

For example, suppose that you have some alarm handling logic in segment 2 of a three-segment logic program. You can use the segment scheduler editor to *control* segment 2 based on the status of a coil 00056—if the coil is ON, segment 2 logic will be activated in the scan, and if the coil is OFF the segment will not be solved in the scan. I/O servicing is still performed, regardless of the conditional status.

Here is how the MODSOFT segment scheduler would show the resulting order-of-solve table:



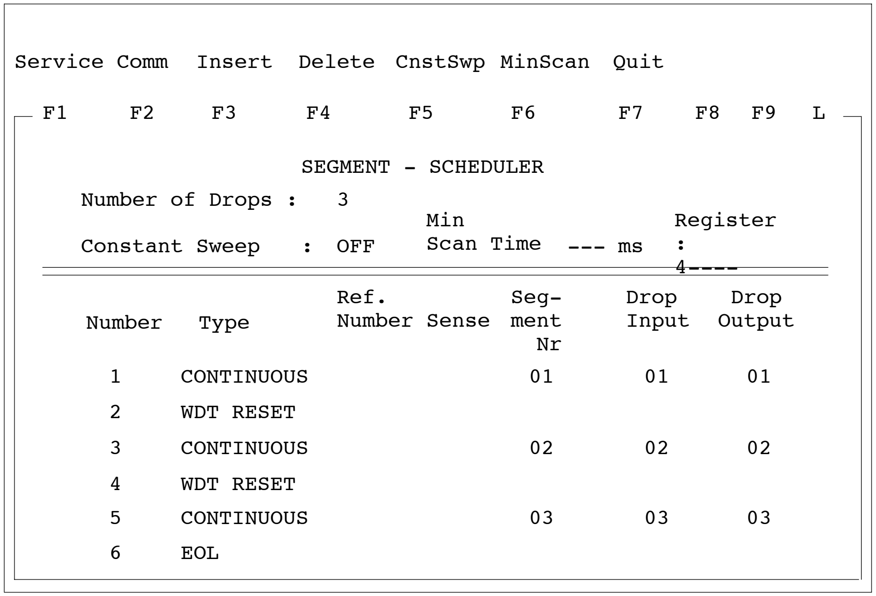
```
Service Comm   Insert   Delete  CnstSwp  MinScan   Quit

  F1      F2     F3      F4       F5       F6       F7     F8   F9    L

                        SEGMENT - SCHEDULER

     Number of Drops :    3
                                    Min                     Register
     Constant Sweep    :  OFF       Scan Time  --- ms       :
                                                            4----

                            Ref.            Seg-      Drop     Drop
     Number    Type         Number  Sense   ment      Input    Output
                                              Nr
       1     CONTINUOUS                     01        01       01
       2     CONTINUOUS                     03        03       03
       3     CONTROLLED     00056    ON     02        02       02
       4     CONTINUOUS                     03        03       03
       5     EOL
```

An Order-of-Solve Table Rescheduled for a Controlled Logic Segment

7.12 Using the Segment Scheduler to Improve Comm Port Servicing

When you find that the frequency of standard end-of-scan servicing of communication ports, option processors, or system diagnostics is inadequate for your application requirements, you can increase service frequency by inserting one or more *reset watchdog timer* routines in the order-of-solve table. Each time this routine is encountered by the CPU, it causes all communication ports to be serviced and causes the system diagnostics to be run.

Here is how the MODSOFT segment scheduler would show an order-of-solve table where the comm ports are serviced after each segment in the logic program:



```
Service Comm   Insert  Delete  CnstSwp MinScan  Quit

 F1     F2     F3     F4      F5      F6      F7    F8   F9   L

                      SEGMENT - SCHEDULER
   Number of Drops :   3
                              Min                   Register
   Constant Sweep   :  OFF    Scan Time  --- ms     :
                                                    4----
                         Ref.          Seg-      Drop    Drop
    Number   Type        Number Sense  ment      Input   Output
                                        Nr
      1     CONTINUOUS                  01        01       01
      2     WDT RESET
      3     CONTINUOUS                  02        02       02
      4     WDT RESET
      5     CONTINUOUS                  03        03       03
      6     EOL
```

An Order-of-Solve Table Rescheduled for Three Comm Port Servicings per Scan

7.13 Sweep Functions

Sweep functions allow you to scan a logic program at fixed intervals. They do not make the controller solve logic faster or terminate scans prematurely.

7.13.1 Constant Sweep

Constant Sweep allows you to set target scan times from 10 ... 200 ms (in multiples of 10). A target scan time is the time between the start of one scan and the start of the next; it is not the time between the end of one scan and the beginning of the next.

Constant Sweep is useful in applications where data must be sampled at constant time intervals.

If a Constant Sweep is invoked with a time lapse smaller than the actual scan time, the time lapse is ignored and the system uses its own normal scan rate.

The Constant Sweep target scan time encompasses logic solving, I/O and Modbus port servicing, and system diagnostics. If you set a target scan of 40 ms and the logic solving, I/O servicing, and diagnostics require only 30 ms, the controller will wait 10 ms on each scan.

Consult your programming documentation for procedures to invoke a Constant Sweep function.

7.13.2 Single Sweep

The Single Sweep function allows 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 performed calculations to be examined for errors.

🛑 **Warning The Single Sweep function should not be used to debug controls on machine tools, processes, or material handling systems when they are active. Once a specified number of scans has been solved, all outputs are frozen in their last state. Since no logic solving is taking place, the controller ignores all input information. This can result in unsafe, hazardous, and destructive operation of the machine or process connected to the controller.**

Consult your programming documentation for procedures to invoke Single Sweep functions.

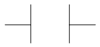
Chapter 8
Contacts, Shorts, and Coils

- ❐ Relay Contacts
- ❐ Vertical and Horizontal Shorts
- ❐ Normal and Latched Coils

8.1 Relay Contacts

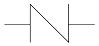
The relay contact is the basic programming element. It can be referenced to a logic coil (0*x*) or a discrete input (1*x*). There are four types of relay contacts:

Normally Open



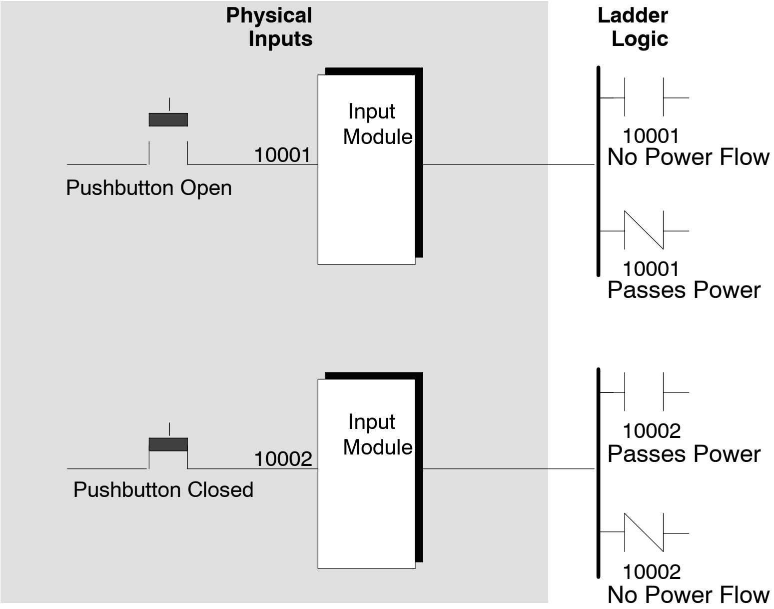
A normally open contact passes power when its referenced coil or input is ON.

Normally Closed

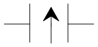


A normally closed contact passes power when its referenced coil or input is OFF.

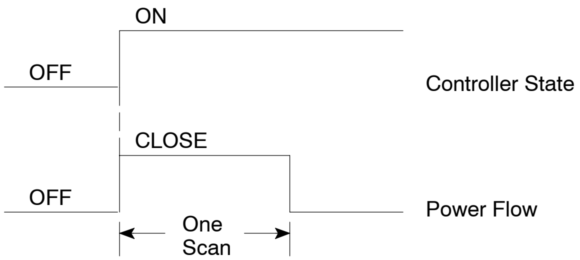
Here is an example of how you might use two sets of normally open and normally closed contacts to create logic for a momentary pushbutton switch:



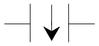
Positive Transitional



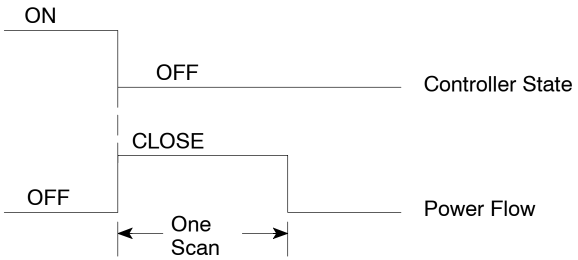
A positive transitional contact passes power for only one scan as the contact or coil transitions from OFF to ON.



Negative Transitional



A negative transitional contact passes power for only one scan as the contact or coil transitions from ON to OFF.



8.2 Vertical and Horizontal Shorts

Shorts are simply straight-line connections between contacts and/or function blocks.

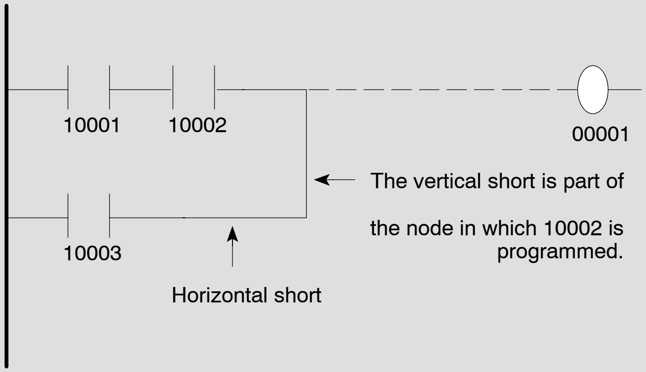
A vertical short connects contacts or function blocks one above the other in a network column. Vertical shorts can also be used to connect inputs or outputs in a function block to create *either/or* conditions. When two contacts are connected by vertical shorts, power is passed when one or both contacts receive power. A vertical short does not consume any user memory.

Horizontal shorts are used in combination with vertical shorts to expand logic within a network without breaking the power flow. A horizontal short consumes one word of memory in a 16 bit CPU and 1.5 words in a 24 bit CPU.

8.2.1 An Either/Or Example

Horizontal and vertical shorts can be combined with relay contacts to create an *either/or* condition in ladder logic.

Ladder Logic for an Either/Or Example



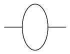
One line of logic contains two contacts (10001 and 10002), and the line below it contains one contact (10003). A horizontal short is placed beside contact 10003, and a vertical short connects the second line with the first line.

Power will pass through to energize coil 00001 if either contacts 10001 and 10002 are energized or if contact 10003 is energized.

8.3 Normal and Latched Coils

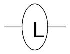
A coil is a discrete output value represented by a 0*x* reference number. Because output values are updated in State RAM by the controller's CPU, a coil may be used internally in the logic program or externally via the Traffic Cop to a discrete output module. Coils are either OFF or ON, depending on power flow in the logic program. When a coil is ON, it may either pass power to a discrete output circuit on the shop floor or change the state of an internal relay contact in state RAM. There are two types of coils:

Normal Coil

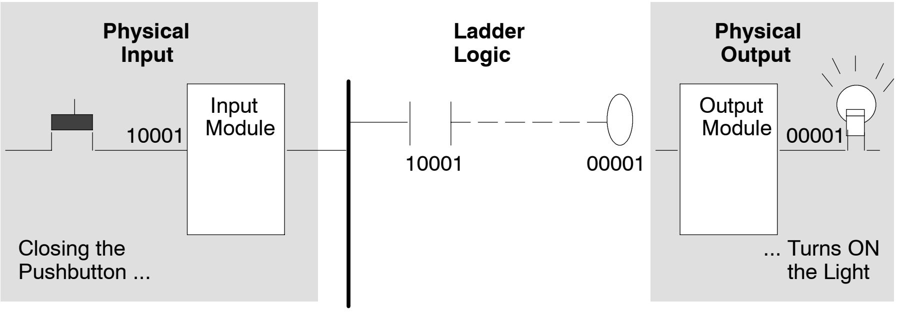


A normal coil is turned OFF if power at the controller is removed.

Latched Coil



If a latched coil has been energized at the time of a controller power loss, the coil will come back up in the same state for one scan once power has been restored.



8.3.1 Coils in a Logic Network

Each network can contain a maximum of seven coils. Each 0*x* reference number can be used as a coil only once, but it can be referenced to any number of relay contacts.

8.3.2 Enable/Disable Capabilities for Discrete Values

Via panel software, you may disable a logic coil or a discrete input in your logic program. A disable condition will cause the input field device to have no control over its assigned 1*x* logic and the logic to have no control over the disabled 0*x* value.

The MEMORY PROTECT switch on your 984 controller must be OFF before you disable (or enable) a coil or a discrete input.

⚠ **Caution There is an important exception you need to be aware of when disabling coils: data transfer functions that allow coils in their *destination* nodes recognize the current ON/OFF state of all coils, whetheer they are disabled or not, and cause the logic to respond accordingly. If you are expecting a disabled coil to remain disabled in the DX function, your application may experience unexpected and undesireable effects.**

8.3.3 Forcing Discretes ON and OFF

The panel software also provides FORCE ON and FORCE OFF capabilities. When a coil or discrete input has been disabled, the only way you can change its state from OFF to ON is with FORCE ON, and the only way to change from ON to OFF with FORCE OFF.

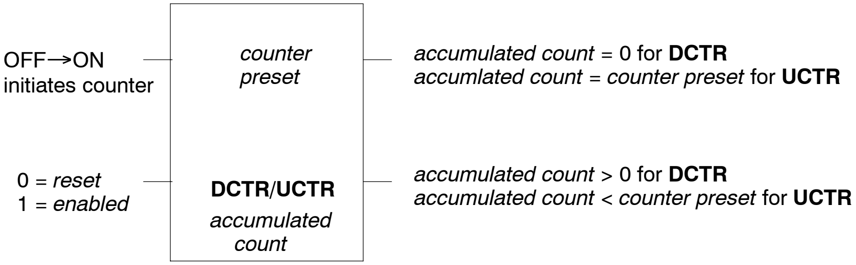
When a coil or input is enabled, it cannot be forced ON or forced OFF.

Chapter 9
Counters and Timers

- ❐ Up Counters and Down Counters
- ❐ Three Kinds of Timers
- ❐ A Real-Time Clock Example

9.1 Up Counters and Down Counters

Two counter instructions are available, **UCTR** and **DCTR**, for up counting and down counting. Both are designed to count control input transitions from OFF to ON either up to or down from a *counter preset* value. Each is a two-node function block structured as follows:

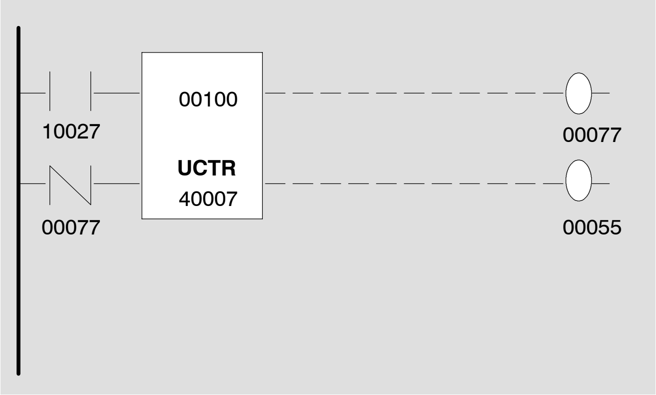


The *counter preset* in the top node can be

- A decimal ranging from 1 ... 999 in 16 bit CPUs and 1 ... 9999 in 24 bit CPUs
- An input register (3*x*)
- A holding register (4*x*)

The bottom node signifies the DCTR or UCTR function and contains a holding register (4*x*) that stores the *accumulated count*.

Here is an example of an up counter:



When contact 10027 is energized, CONTROL IN receives power, and, since contact 00077 is also receiving power, UCTR is enabled.

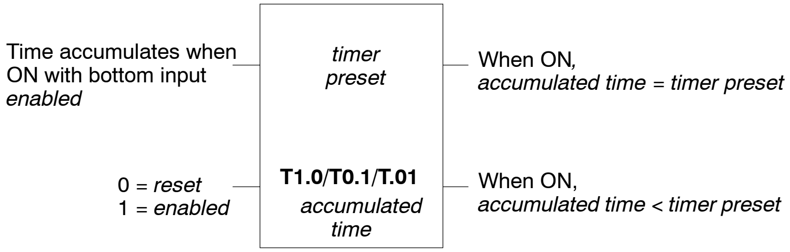
Each time contact 10027 transitions from OFF to ON, the *accumulated count* value increments 1. When the value reaches 100 (when contact 10027 has transitioned 100 times), the top output passes power. Coil 00077 is energized, and coil 00055 is de-energized.

Contact 00077 loses power when coil 00077 is energized, and the *accumulated count* value is reset to 0 on the next scan.

On the next scan, coil 00077 is de-energized. Contact 00077 is then re-energized and the UCTR function is enabled.

9.2 Three Kinds of Timers

Three timer instructions are available for timing an event or creating a delay. They measure time in seconds (**T1.0**), in tenths of a second (**T0.1**), and in hundredths of a second (**T.01**). Each timer is a two-node function block:

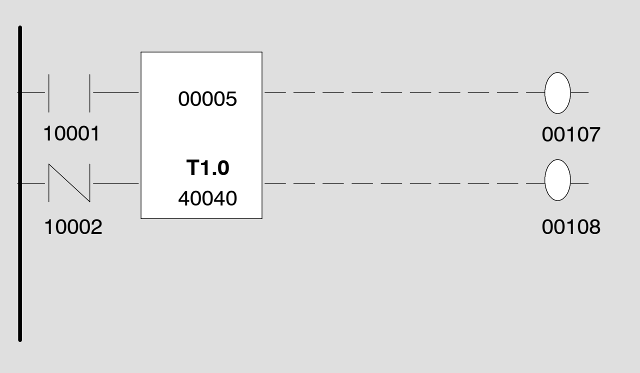


The *timer preset* in the top node can be

- A decimal ranging from 1 ... 999 in 16 bit CPUs and 1 ... 9999 in 24 bit CPUs
- An input register (3*x*)
- A holding register (4*x*)

The bottom node indicates that the timer is incrementing as a **T1.0**, **T0.1**, or **T.01** counter and contains a holding register (4*x*) that stores *accumulated time*.

Caution If you cascade T1.0 timers with *presets* of 1, the timers will time-out together; to avoid this problem, change the *presets* to 10 and substitute a T0.1 timer. The same holds true for a T0.1 timer, in which case you can substitute a T.01 timer.



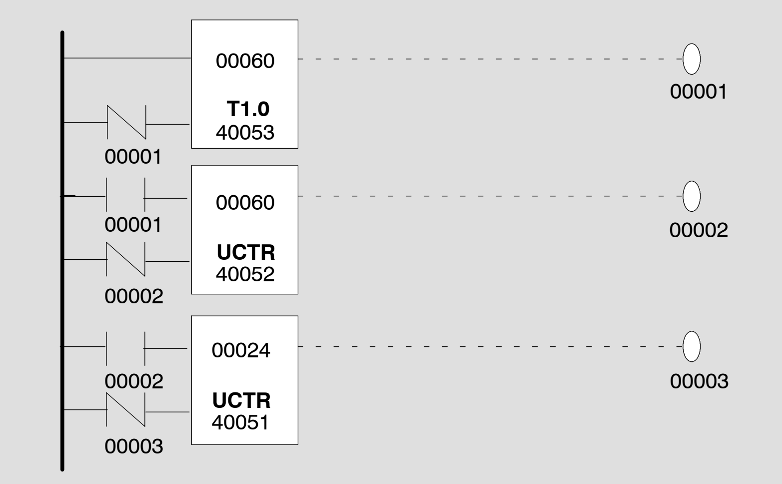
The example above assumes that 10002 is closed (timer enabled) and that the value contained in register 40040 is 0. Because 40040 does not equal the *timer preset* (5), coil 00107 is OFF and coil 00108 is ON.

When 10001 is closed, 40040 begins to accumulate counts at 1 s intervals until it reaches 5. At that point, 00107 is ON and 00108 is OFF.

When 10002 is opened, 40040 resets to 0, coil 00107 goes OFF, and 00108 goes ON.

☞ **Note** If the *accumulated time* value is less than the *timer preset* value, the bottom output will pass power even though no inputs to the block are present.

9.3 A Real-Time Clock Example



The first function block above is a T1.0 instruction programmed as a one minute timer. When logic solving begins, coil 00001 is OFF—both the top and bottom inputs of the timer receive power.

Register 40053 starts incrementing time in seconds. After 60 increments, the top output passes power and energizes coil 00001. Register 40053 is reset. Register 40052 in the first up counter block increments by 1, indicating that one minute has passed.

Because the T1.0 block is no longer equal to the preset, coil 00001 is de-energized and the timer resumes incrementing seconds. When the value in 40052 reaches 60, the top output in the first up counter passes power and energizes coil 00002.

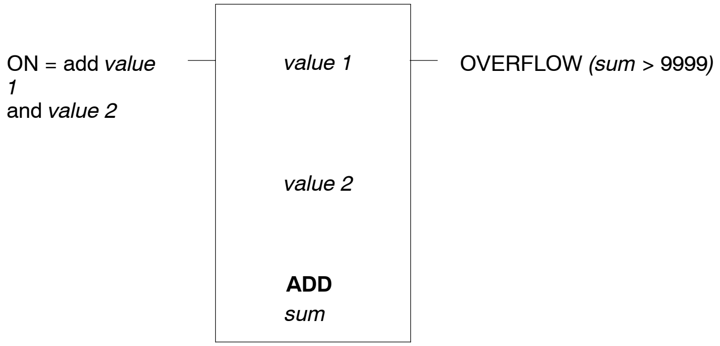
Register 40052 is reset, and the *accumulated count* in the second up counter (register 40051) increases by 1, indicating that one hour has passed.

The correct time of day can be read in registers 40051 (indicating hours), 40052 (indicating minutes), and 40053 (indicating seconds).

Chapter 10
Standard Calculate Functions

10.1 ADD

The ADD instruction adds *value 1* to *value 2* and stores the *sum* in a holding register. ADD is a three-node function block:



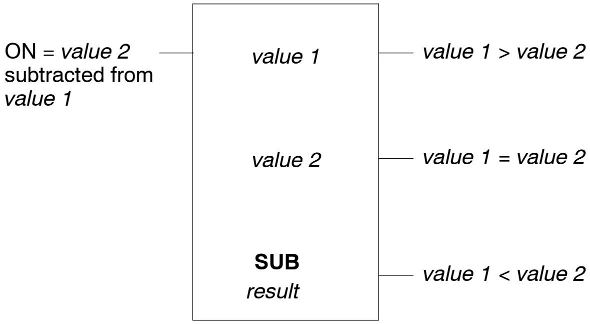
The top node and middle node contain *value 1* and *value 2*, respectively—they can be:

- Decimals ranging from 1 ... 999 in a 16 bit CPU and from 1 ... 9999 in a 24 bit CPU
- Input registers (3*x*)
- Holding registers (4*x*)

The bottom node indicates that this is an ADD function and contains a holding register (4*x*) where the *sum* of the addition is stored.

10.2 SUB

The SUB instruction performs an absolute subtraction (without signs) of *value 1* - *value 2* and stores the *result* in a holding register. It can be used as a comparator, identifying whether *value 1* is greater than, equal to, or less than *value 2*. SUB is a three-node function block:



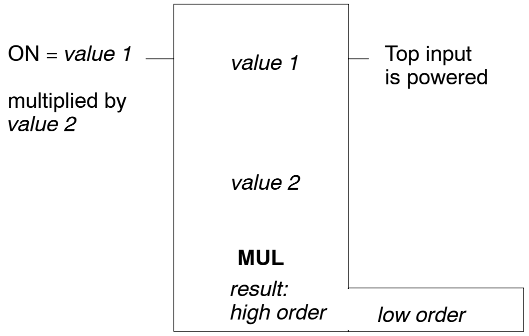
The top node and middle node are *value 1* and *value 2*, respectively—they can be:

- Decimals ranging from 1 ... 999 in a 16 bit CPU and from 1 ... 9999 in a 24 bit CPU
- Input registers (3*x)*
- Holding registers (4*x*)

The bottom node indicates that this is a SUB function and contains a holding register (4*x*) where the *result* of the subtraction is stored.

10.3 MUL

The MUL instruction multiplies *value 1* by *value 2* and stores the *result* in two holding registers. MUL is a three-node function block:



The top node and middle node are *value 1* and *value 2*, respectively—they can be:

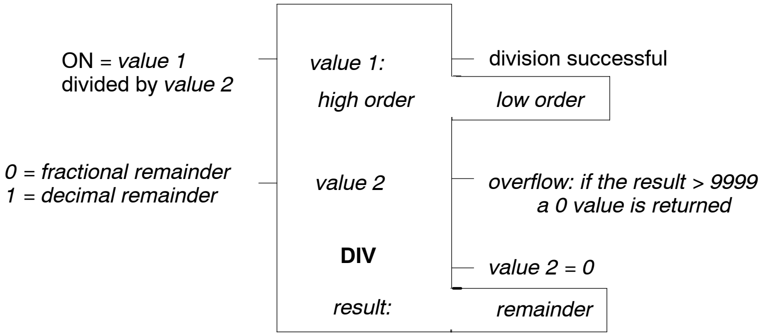
- A decimal ranging from 1 ... 999 in a 16 bit CPU and from 1 ... 9999 in a 24 bit CPU
- An input register (3*x*)
- A holding register (4*x*)

The bottom node indicates that this is a MUL function and contains two consecutive a holding registers (4*x* and 4*x* + 1) where the *result* of the multiplication is stored.

The higher order digits are stored in the register specified in the bottom node, and the lower order digits are stored in the next sequential register. For example, if the top node value is 8000 and the middle node value is 2, the *result* (16,000) is stored in two sequential registers: 4*x* contains the higher order digits (0001), and 4*x* + 1 contains the lower order digits (6000).

10.4 DIV

The DIV instruction divides *value 1* by *value 2* and stores the *result* and the *remainder* in two consecutive holding registers. DIV is a three-node function block:



The top node, *value 1*, can be:

- A decimal ranging from 1 ... 999 in a 16 bit CPU and from 1 ... 9999 in a 24 bit CPU
- Two consecutive input registers, 3*x* for the higher order digits and 3*x* + 1 for the lower order digits
- Two consecutive holding registers, 4*x* for the higher order digits and 4*x* + 1 for the lower order digits

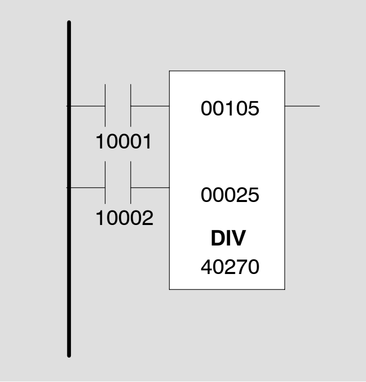
The middle node, *value 2*, can be:

- A decimal ranging from 1 ... 999 in a 16 bit CPU and from 1 ... 9999 in a 24 bit CPU
- An input register (3*x*)
- A holding register (4*x*)

The bottom node indicates that this is a DIV function and contains two holding registers (4*x* and 4*x* + 1). The *result* of the division is stored in the first register, and the *remainder* is stored in the second register. The *remainder* may be expressed as a fraction or a decimal, depending on whether the middle input is a 1 or a 0.

10.5 A DIV Example

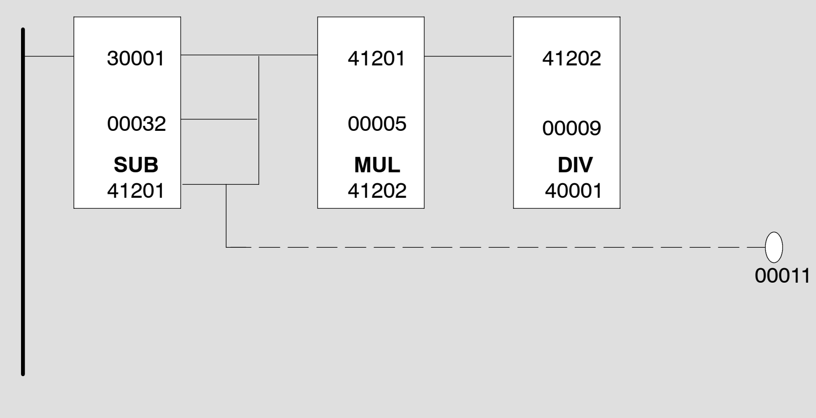
Here is an example of a DIV operation where *value 1* (105) is divided by *value 2* (25). The *result* is stored in register 40270 and the *remainder* is stored in register 40271.



The *result* (4) is stored in register 40270, and the *remainder* (5) is stored in register 40271.

If 10002 is open, the *remainder* is expressed as a fraction (0005). If 10002 is closed, the *remainder* is expressed as a decimal (2000).

10.6 A Fahrenheit-to-Centigrade Conversion Example



☞ **Note** The vertical short to coil 00011 must be to the left of the vertical shorts linking the three SUB block outputs.

We want to implement the formula

$°C = (°F - 32) \times 5/9$

When the top input of the SUB function block receives power, the number 32 is subtracted from the value in register 30001, which represents some number of degrees Fahrenheit. The *result* is placed in register 41201.

The top input to the MUL function block then receives power, whether the SUB *result* is positive, negative, or 0. If the SUB *result* is negative, coil 00011 is energized to indicate a negative value.

The value in register 41201 is then multiplied by 5, and the *result* is placed in register 41202. The top input of the DIV function block is then energized, and the value in register 41202 is divided by 9. The *result*, which is the temperature conversion in degrees Centigrade, is placed in register 40001.

Chapter 11
DX Move Functions

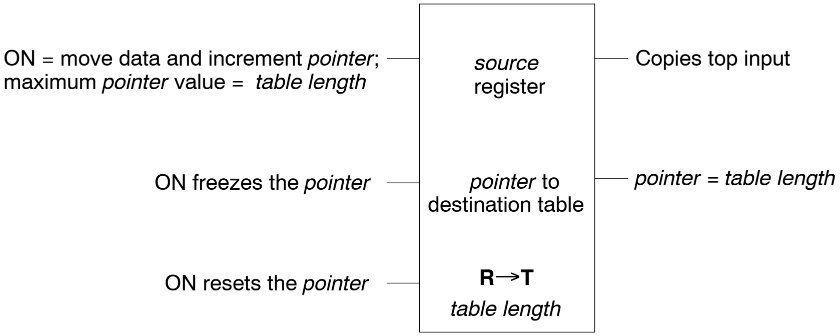
- ❒ Moving Registers and Tables
- ❒ Moving Blocks to Tables and Tables to Blocks
- ❒ Two Functions for Building a FIFO Queue
- ❒ SRCH
- ❒ BLKM
- ❒ A Recipe Storage Example

11.1 Moving Registers and Tables

The 984 standard instruction set provides three function blocks for moving register and table data—one for moving register values to a table (R→T), one for moving table values to a single register (T→R), and one for moving values from one table to another (T→T). Each of these register transfer instructions is a three-node function block, and the system can accommodate the transfer of one register per scan.

11.1.1 Register-to-Table Move

The R→T instruction copies the bit pattern of a register or of 16 discretes to a specific register located in a table:

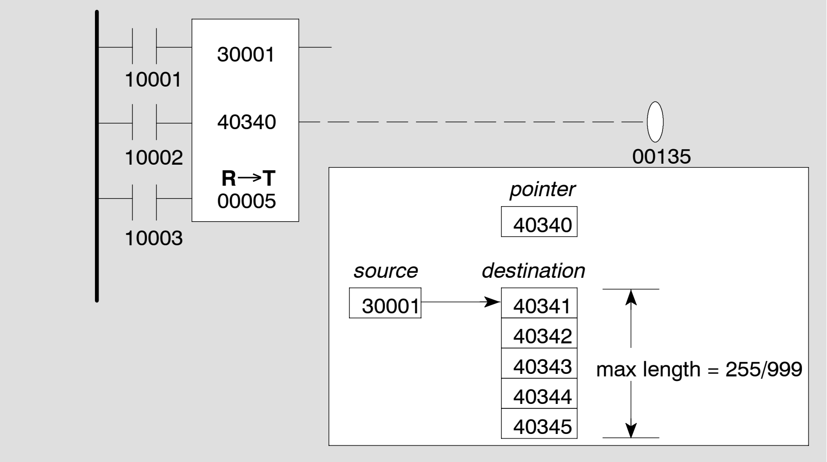


The top node can be:

- The first 0*x* in a table of coils or discrete outputs
- The first 1*x* in a table of discrete inputs
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

The value in the middle node is a *pointer* to the register in the destination table where data will be moved in this scan. The *pointer* is a 4*x* register, and the first register in the destination table is 4*x* + 1. The number of registers in the destination table is specified in the bottom node. A value of 0 in the *pointer* equals the first register in the table.

The bottom node indicates that the function is a register-to-table transfer instruction and specifies the *table length*—it may range from 1 ... 255 in 16 bit CPUs and from 1 ... 999 in 24 bit CPUs.



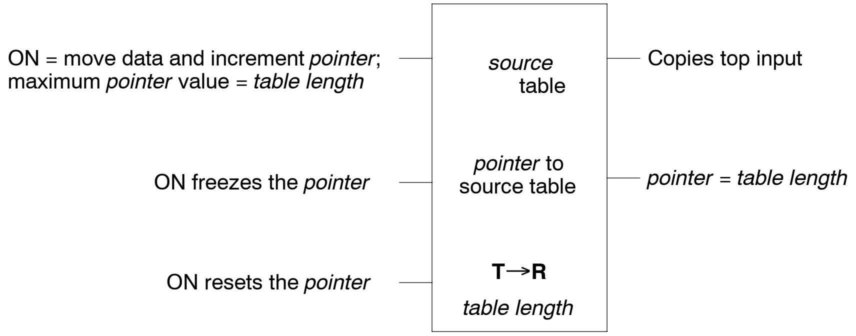
An R→T Example

The first transition of 10001 copies 30001 to 40341 and increments the *pointer* value stored in 40340 to 1; its second transition copies 30001 to 40342 and increments the *pointer* value to 2; and so on through five transitions. At the fifth transition, which copies 30001 to 40345 and increments the *pointer* value to the *table length*, the middle output passes power, energizing coil 00135. No R→T operations are possible while these two values are equal.

If, after the second transition, 10002 were to be energized, the *pointer* value could not be changed. All subsequent transitions of 10001 would cause the value in 30001 to be copied to 40343. When 10003 is energized, the *pointer* will be reset to 0.

11.1.2 Table-to-Register Move

The **T→R** instruction copies the bit pattern of a register or 16 discretes located within a table to a specific holding register:



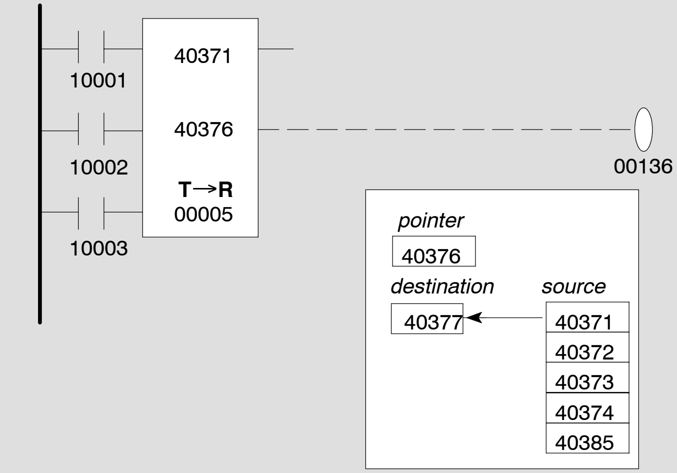
The top node can be:

- The first 0*x* in a table of coils or discrete outputs
- The first 1*x* in a table of discrete inputs
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

The value in the middle node is a *pointer* to the register in the *source* table that will be moved in this scan. The *pointer* is a 4*x* register, and the destination register is 4*x* + 1. A value of 0 in the *pointer* equals the first register in the table.

The bottom node indicates that the function is one of the three register transfer instructions and specifies the *length* of the *source* table—in the range 1 ... 255 in 16 bit CPUs and 1 ... 999 in 24 bit CPUs. The number specifies the total number of registers to be transferred.

A T→R Example



The first transition of 10001 copies the contents of 40371 to register 40377 and increments the *pointer* value stored in 40376 to 1. The second transition of 10001 copies 40372 to 40377 and increments the *pointer* value to 2; the third transition copies 40373 to 40377 and increments the *pointer* value to 3; the fourth transition copies 40374 to 40377 and increments the *pointer* value to 4.

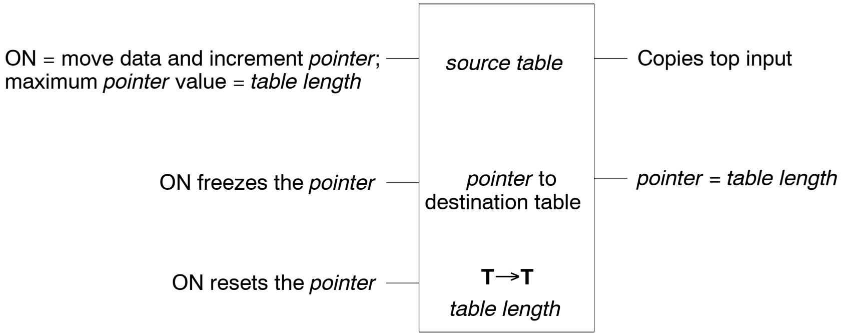
The fifth transition of 10001 copies 40375 to 40377 and increments the *pointer* value to 5. Because the *pointer* value now equals the *table length*, the middle output passes power, energizing coil 00136. No T→R operations are possible while these two values are equal.

If, after the second transition of 10001, 10002 were to be energized, the *pointer* value could not be changed. All subsequent transitions of 10001 would cause the value in 40343 to be copied to 40377.

When 10003 is energized, the *pointer* is reset to 0.

11.1.3 Table-to-Table Move

The **T→T** instruction copies the bit pattern of a register or 16 discretes from a position within one table to the same position in a second table of holding registers:



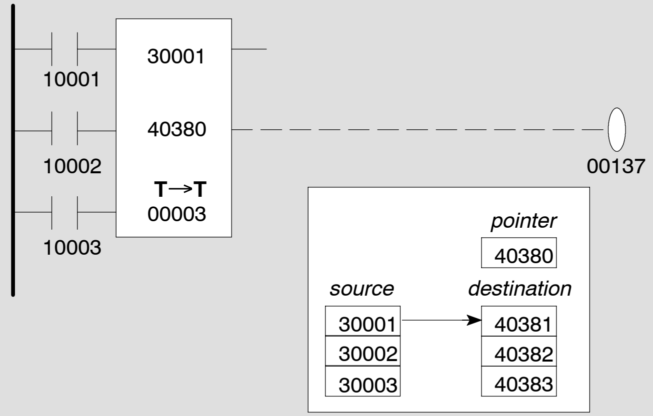
The top node can be:

- The first 0*x* in a *source table* of coils or discrete outputs
- The first 1*x* in a *source table* of discrete inputs
- The first 3*x* in a *source table* of input registers
- The first 4*x* in a *source table* of holding registers

The value in the middle node is a *pointer* to the register in the *source* table to be moved in the scan and to the register in the destination table where the *source* register will go. The *pointer* is a 4*x* register, and the first register in the destination table is 4*x* + 1. The length of the two tables must be equal, and this length is specified in the bottom node. A value of 0 in the *pointer* equals the first register in the table.

The bottom node indicates that the function is a table-to-table register transfer instruction and specifies the *table length* for both the *source* and destination tables. The length may range from 1 ... 255 in 16 bit CPUs and 1 ... 999 in 24 bit CPUs.

A T→T Example



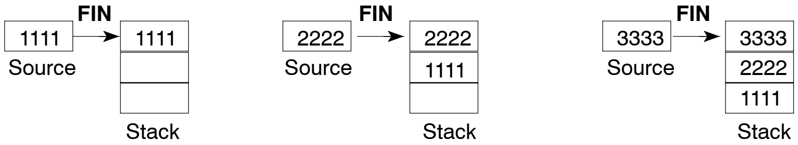
The first transition of 10001 moves the contents of 30001 to register 40381 and increments the *pointer* value stored in 40380 to 1, and the second transition moves the contents of 30002 to register 40382 and increments the *pointer* value to 2.

The third transition of 10001 moves the contents of 30003 to register 40383 and increments the *pointer* value to 3. Because the *pointer* value now equals the *table length*, the middle input passes power and energizes coil 00137. No T→T operations are possible while these two values are equal.

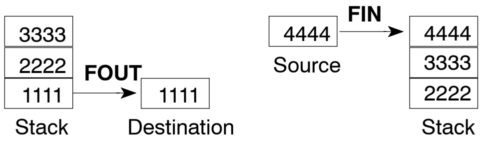
If, after the second transition of 10001, 10002 were to be energized, the *pointer* value would be locked to 2, and all subsequent transitions of 10001 would cause the value in 30003 to be moved to register 40383.

11.2 Two Functions for Building a FIFO Queue

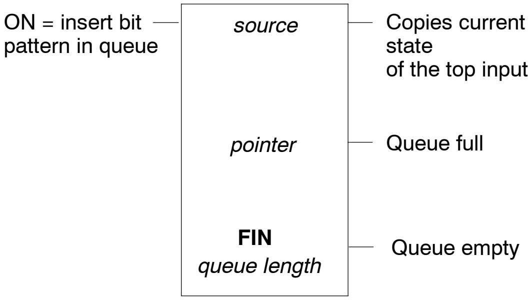
The standard 984 instruction set provides two function blocks that are used to produce a first in-first out queue. The FIN instruction copies the bit pattern of any register or 16 discretes to the first register in a table of holding registers; this register is at the top of the queue:

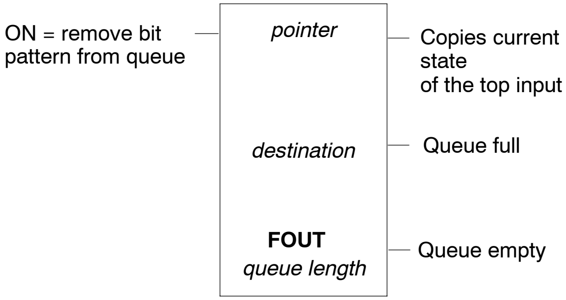


The FOUT instruction moves the bit pattern of a holding register within a table to a *destination* register or to 16 discrete outputs; the oldest data in the queue is moved first. FOUT should be placed before FIN to ensure that the oldest data are removed from a full queue before the newest data are entered. If the FIN block were to appear first, the attempt to enter the new data would be ignored if the queue were full.



Both instructions are three-node function blocks:





The *source*, which is specified in the top node of the **FIN** block, may be:

- ☐ The first of 16 logic coils (0*x*)
- ☐ The first of 16 discrete inputs (1*x*)
- ☐ An input register (3*x*)
- ☐ A holding register (4*x*)

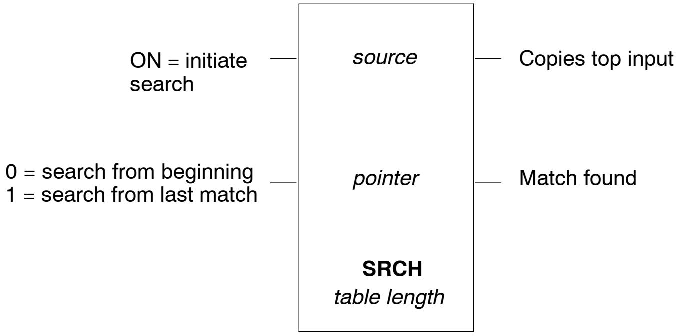
The *pointer*, which is specified in the middle node of the **FIN** block and the top node of the **FOUT** block, is a holding register (4*x*). A *pointer* indicates where in the table the data will be taken from or written to.

The bottom node indicates that the block is either an **FIN** or **FOUT** instruction and specifies the *queue length*, which may range from 1 ... 100 and which represents the number of registers in the queue.

STOP **Warning FOUT will override any disabled coils within a *destination* table without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the FOUT operation.**

11.3 SRCH

The SRCH instruction searches a table of registers for a specific bit pattern. SRCH is a three-node function block:



The top node specifies the *source* table to be searched; it may be

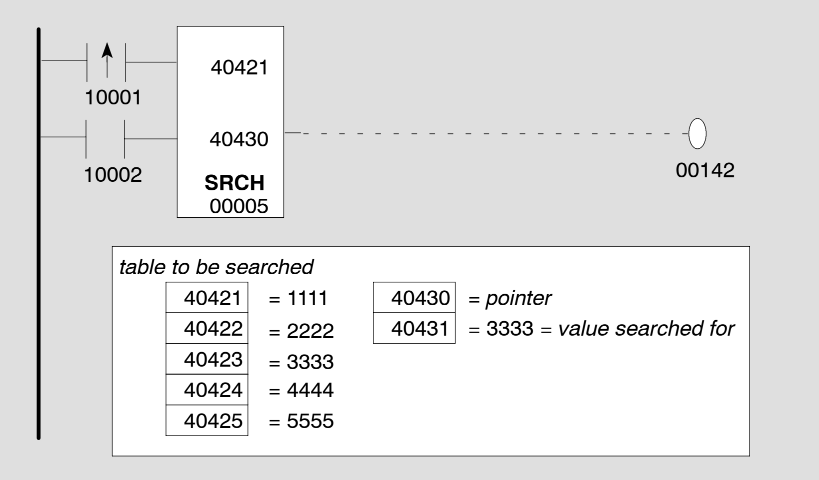
- The first 3*x* in a table of input references
- The first 4*x* in a table of holding registers

The middle node must be a holding register (4*x*). It is a *pointer* to the table being searched (as specified in the top node). The next consecutive register, 4*x* + 1, contains the value or bit pattern being searched for.

The bottom node indicates that this is a SRCH function and specifies a table length, which may range from 1 ... 100.

11.3.1 A SRCH Example

Here we search a five-register table for the register that contains the value 3333.



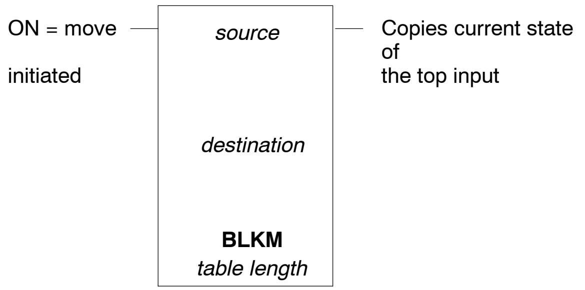
The *source* table is searched for a 3333 on every scan where 10001 transitions from OFF to ON. If 10002 is OFF, the SRCH function finds a match at register 40423 and stops searching for the remainder of the scan. It sets the *pointer* value to 3 for one scan, indicating that a match exists in table position 3. Coil 00142 is energized for one scan.

When 10001 is transitioned a second time, it starts again at 40421 and searches for a match. It will find it again at 40423.

When 10002 is energized and 10001 transitions from OFF to ON, the *source* table is searched for a 3333. The SRCH function finds a match at register 40423 and stops the SRCH. It sets the *pointer* value to 3, indicating that a match exists in table position 3. Coil 00142 is energized for one scan.

11.4 BLKM

BLKM is the block move instruction—in one scan, it copies the entire contents of one table to another table of outputs or holding registers. BLKM is a three-node function block:



The top node—*source*—may be:

- The first 0*x* in a table of output references
- The first 1*x* in a table of input references
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

The middle node—*destination*—may be:

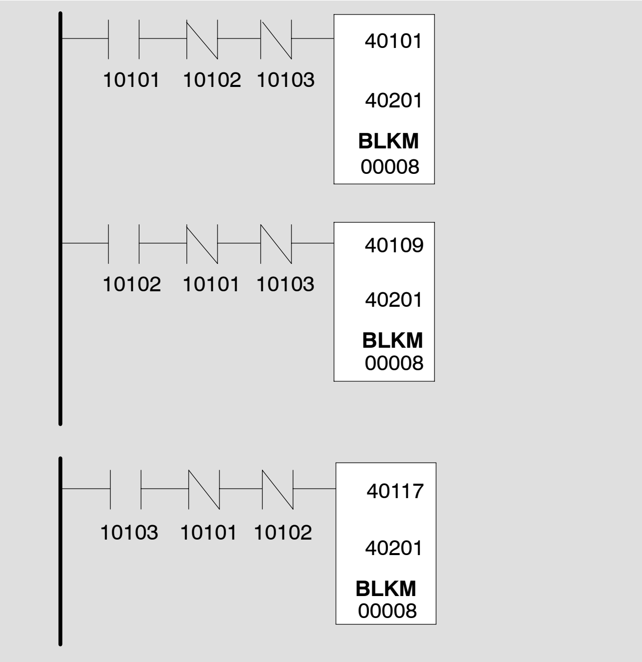
- The first 0*x* in a table of coils or output registers (the one and only time that the referenced coils may be used)
- The first 4*x* in a table of holding registers

The bottom node indicates that this is a BLKM function and specifies a *table size* that can range from 1 ... 100.

Warning BLKM will override any disabled coils within a *destination* table without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the BLKM instruction.

11.5 A Recipe Storage Example

You can use ladder logic to write specific process programs (or *recipes),* store each in a unique table, then write a general process program and store it in another working table. The recipe tables must be structured with similar information in corresponding registers—if a heating temperature is in the third register in one recipe table, it should be in the third register in all recipe tables. Recipes can be pulled into the generic process program with BLKM functions:

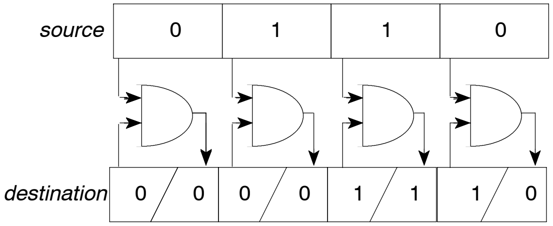


The process is controlled with three input switches—10101, 10102, and 10103. To run process A, turn on 10101, and leave 10102 and 10103 off. When input 10101 is energized, it passes power through normally closed contacts 10102 and 10103. A BLKM function moves the recipe for process A from registers 40101 ... 40108 to registers 40201 ... 40208. This table of registers is a working table, with each register controlling a part of the general process. By using one working table, you can control the output for three separate processes with only one program.

Chapter 12
DX Matrix Functions

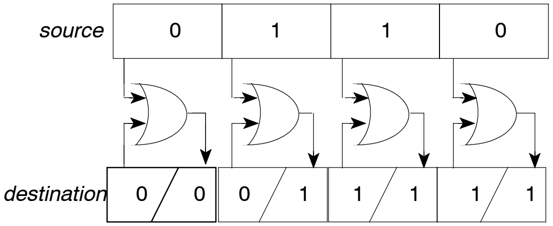
12.1 Three Boolean Functions

The standard 984 instruction set provides three function blocks that perform AND, OR, and Exclusive OR Boolean operations. The **AND** instruction logically ANDs each bit in a *source* matrix with corresponding bits in a *destination* matrix. The result is placed in the *destination* matrix, overwriting the previous contents:



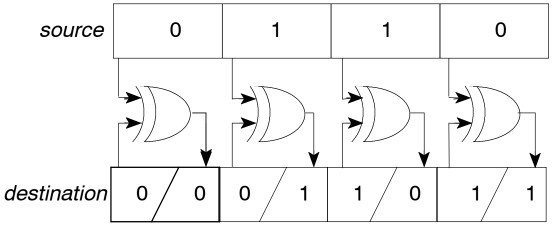
ANDing Operation

The **OR** instruction logically ORs each bit in a *source* matrix with corresponding bits in a *destination* matrix:



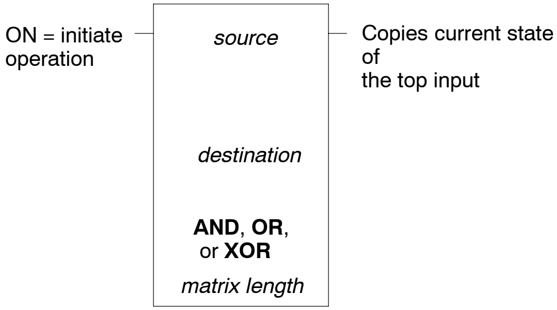
ORing Operation

The **XOR** instruction performs a logical Exclusive OR function on each bit in a *source* matrix with corresponding bits in a *destination* matrix.



XORing Operation

Each of these instructions is a three-node function block:



The top node—*source*—may be:

- The first 0*x* in a table of output references
- The first 1*x* in a table of input references
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

The middle node—*destination*—may be:

- The first 0*x* in a table of output references
- The first 4*x* in a table of holding registers

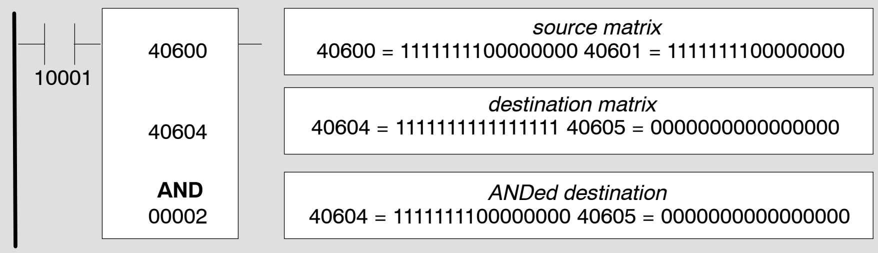
If you specify a 0*x* in the middle node, it counts as the one and only time that the referenced coils may be used.

The bottom node indicates which type of Boolean function to implement and specifies a *matrix length* that may range from 1 ... 100 words—i.e., a length of 2 indicates 32 bits.

Warning These Boolean functions will override any disabled coils within the *destination* group without enabling them. This can cause personal injury if a coil has disabled an operation for maintenance or repair because the coil's state can change as a result of the Boolean operation.

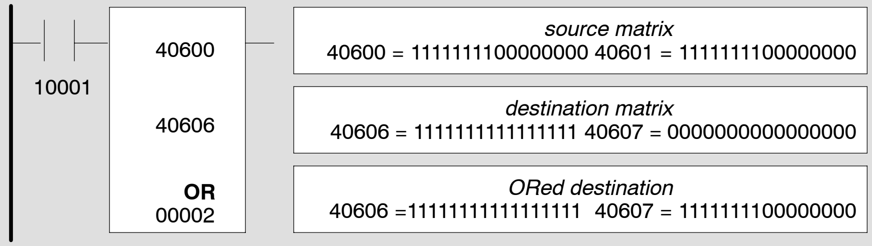
12.2 Some Boolean Examples

ANDing Example



When 10001 passes power, the bit matrix formed by registers 40600 and 40601 are ANDed with the bit matrix formed by registers 40604 and 40605. The result is copied into registers 40604 and 40605, overwriting the previous bit pattern. (If you want to keep the original bit pattern of registers 40604 and 40605, copy the information into another table before performing an AND operation using a BLKM.)

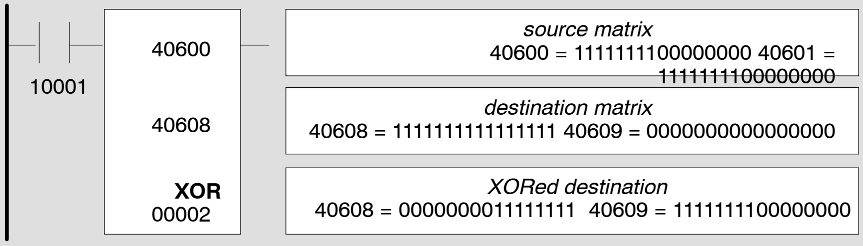
ORing Example



Whenever 10001 passes power, the bit matrix formed by registers 40600 and 40601 is ORed with the bit matrix formed by 40606 and 40607. The result is copied into registers 40606 and 40607.

⚠ **Caution Outputs and coils cannot be turned OFF with the OR instruction.**

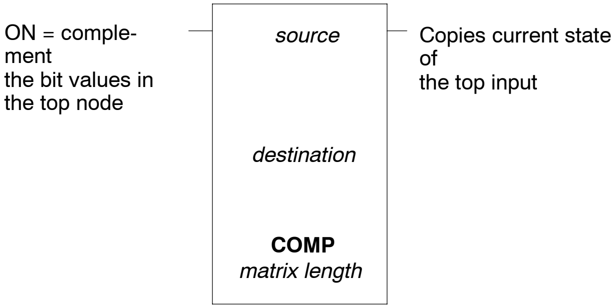
XORing Example



When 10001 passes power, the bit matrix formed by registers 40600 and 40601 is XORed with the bit matrix formed by 40608 and 40609. The result is copied into registers 40608 and 40609.

12.3 COMP

The COMP instruction complements the bit pattern of one matrix (changes all 0’s to 1’s and all 1’s to 0’s), then copies the result into a second matrix, all in the same scan. COMP is a three-node function block:



The matrix specified in the top node is the data *source*; it may be:

- The first 0*x* in a table of output references
- The first 1*x* in a table of input references
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

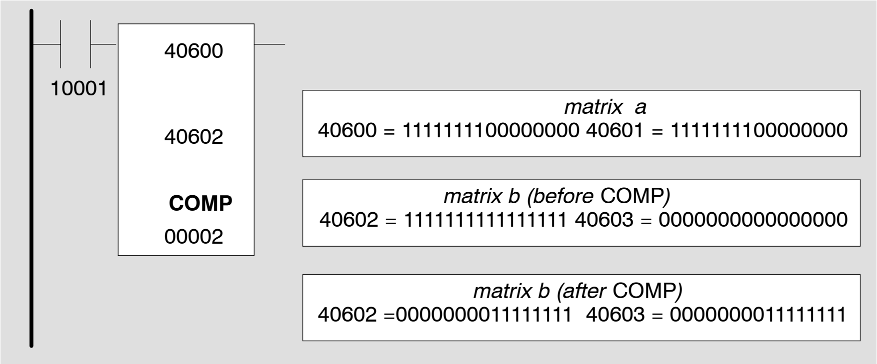
The matrix specified in the middle node is the *destination* for the complemented data; it may be:

- The first 0*x* in a table of output references
- The first 4*x* in a table of holding registers

If the middle node entry is a 0*x*, it counts as the one and only time that the referenced coils may be used.

The bottom node indicates that this is a COMP function and specifies a matrix length that can range from 1 ... 100.

12.3.1 A COMP Example

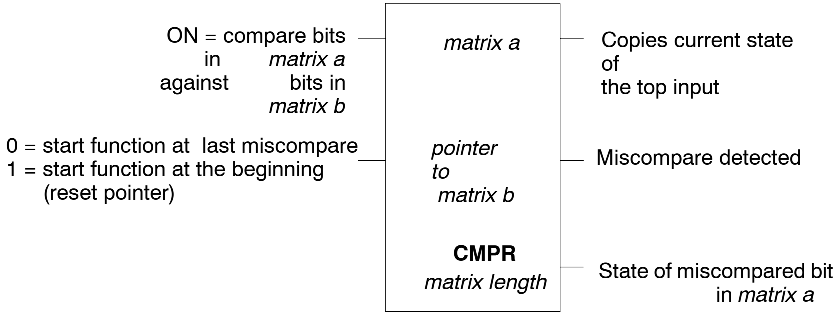


When 10001 passes power, the bit value complements in the *source* matrix (registers 40600 and 40601) are copied into the *destination* matrix (registers 40602 and 40603).

Warning COMP will override any disabled coils within the *destination* matrix without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the COMP instruction.

12.4 CMPR

The CMPR instruction compares the bit pattern of one matrix against the bit pattern of a second matrix for discrepancies. CMPR is a three-node function block:



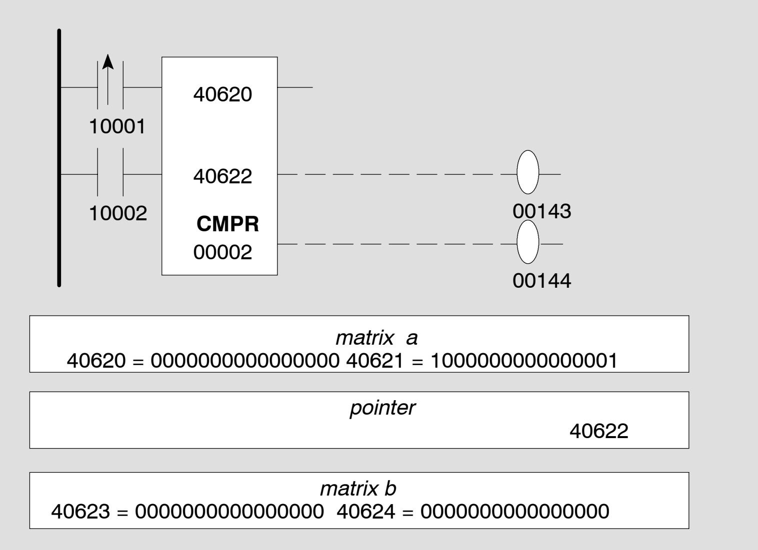
The matrix in the top node specifies the *source* data to be compared; it may be:

- The first 0*x* in a table of output references
- The first 1*x* in a table of input references
- The first 3*x* in a table of input registers
- The first 4*x* in a table of holding registers

The middle node must be a holding register (4*x*); it is the *pointer* to a particular bit in the matrix starting with 4*x* + 1.

The bottom node indicates that this is a CMPR function and specifies a matrix length that can range from 1 ... 100.

12.4.1 A CMPR Example



If 10002 is energized, *matrix a* is compared against matrix b on every scan that 10001 receives power. Matrix b has all bits cleared to 0. The comparison is done bit by bit. This finding of a miscompare is accomplished in one scan.

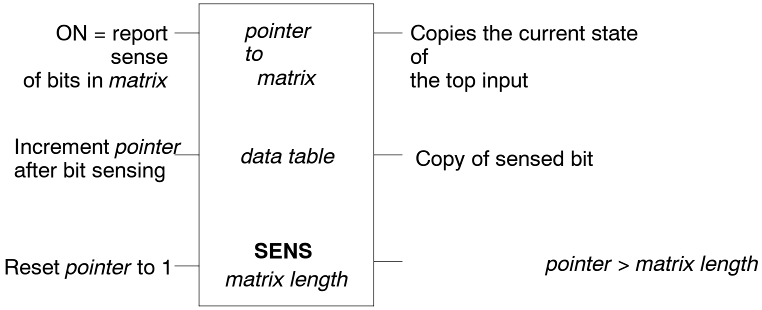
In this example, the comparison continues until bit 17, where *matrix a* = 1 and matrix b = 0. At this point, when 40622 = 17, the function stops; 00143 and 00144 energize for one scan. On the second transition of 10001, the function starts again at bit 1 and stops again when 40622 = 17.

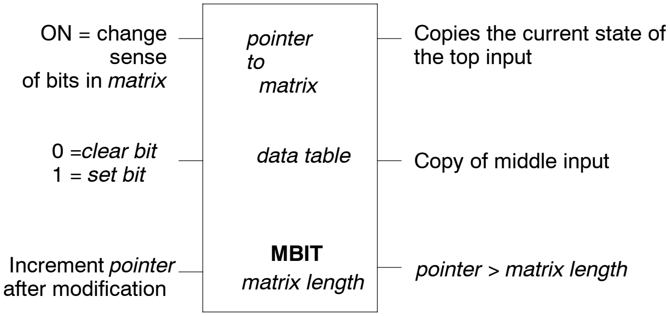
If 10002 is de-energized, the first transition of 10001 will stop the function at 40622 = 17; 00143 and 00144 will energize for one scan. On the second transition of 10001, the function will stop at 40622 = 32; 00143 and 00144 will energize for one scan.

Coil 00144 indicates the sense of the bit in the source matrix when a miscompare occurs.

12.5 Sensing and Modifying Bits in a Matrix

The standard 984 instruction set provides two function blocks that allow you to examine and modify current bit values inside data tables in a matrix. The **SENS** instruction examines and reports the sense—1 or 0—of specific bits within a matrix. The **MBIT** instruction modifies a specific bit within a matrix—a 0 bit is set to 1 or a 1 bit is cleared to 0. One bit may be sensed or modified per scan. Both instructions are three-node function blocks:





☞ **Note** The differences in each of the function blocks are in the way the middle and bottom inputs are treated; the block nodes themselves are essentially the same.

The top node is a *pointer* to a value to be sensed or modified in the *data table*; it may be:

- A constant when the value falls in the range 1 ... 999 in 16 bit CPUs or 1 ... 9600 in 24 bit CPUs
- An input register (3*x*) that may hold a value in the range 1 ... 4080 in 16 bit CPUs or 1 ... 9600 in 24 bit CPUs
- A holding register (4*x*) that may hold a value in the range 1 ... 4080 in 16 bit CPUs or 1 ... 9600 in 24 bit CPUs

The middle node is the first word or register in the *data table*; it may be:

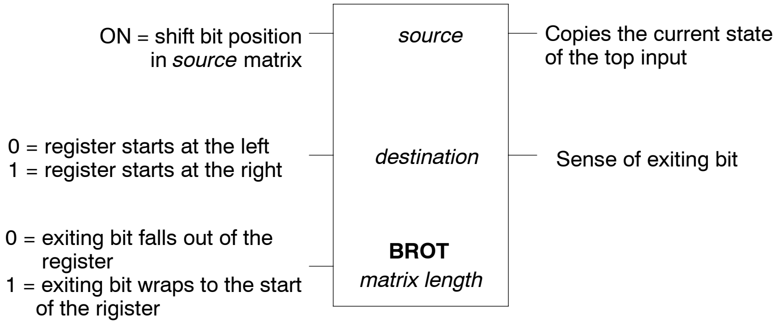
- The first 0*x* in a table of output references
- The first 4*x* in a table of holding registers

The bottom node indicates that the function is either a SENS or MBIT operation and specifies a *matrix length* that may range from 1 ... 255 in 16 bit CPUs and from 1 ... 600 in 24 bit CPUs. The number represents registers or groups of 16 discretes—for example, 200 = 3200 bits.

Warning   MBIT will override any disabled coils within a *destination* group without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the MBIT instruction.

12.6 Rotating a Bit Pattern

The BROT instruction rotates or shifts the bit pattern of a matrix. The bits shift one position per scan. BROT is a three-node function block:



The top node is the *source* node, which can be

- The first 0*x* in a matrix of output references
- The first 1*x* in a matrix of input references
- The first 3*x* in a matrix of input registers
- The first 4*x* in a matrix of holding registers

The middle node is the *destination*, which can be

- The first 0*x* in a matrix of output references
- The first 4*x* in a matrix of holding registers

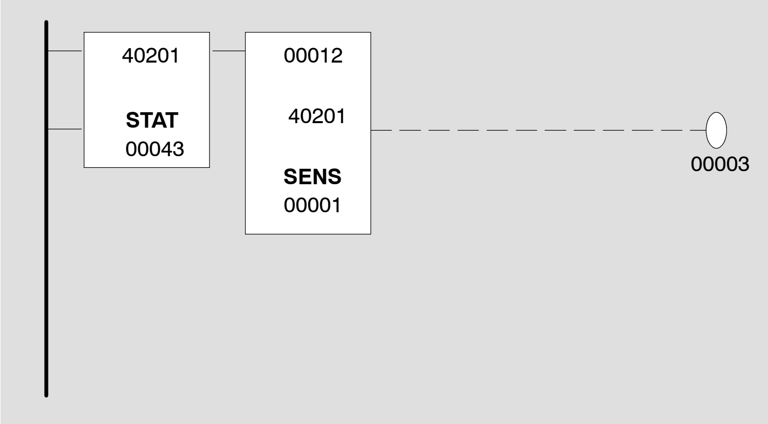
The bottom node indicates that the function is a BROT operation and specifies a *matrix length* that may range from 1 ... 100.

STOP

Warning BROT will override any disabled coils within a *destination* table without enabling them. This can cause injury if a coil has been disabled for repair or maintenance because the coil's state can change as a result of the BROT instruction.

12.7 How to Report Status Information

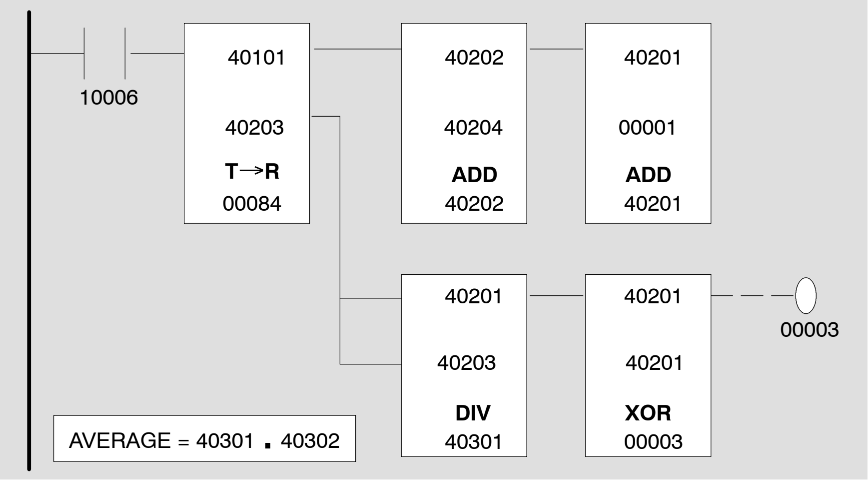
A simple ladder logic construction of a STAT block and a SENS block allows you to report system status information as part of your User Logic program. In this example, bit 12 of register 40201 is being checked. All other bits may be checked using the same method:



The top input to the STAT block receives power on every scan because it is attached to the power rail. Status information is recorded in registers 40201 ... 40243. Register 40201 holds the controller status, which needs to be interpreted.

Since each bit's state represents different information, you can use a SENS block to report incoming bit status. Connect the top output of the STAT block to the top input of the SENS block. This construction lets you check and report the complete bit status on every scan.

12.8 A Simple Table Averaging Example



When input 10006 receives power, the top input to the T→R block receives power and the value in the first register in the table of registers 40101 ... 40184 is copied into the middle node (40204) of the first ADD block. The middle node (40203) in the DIV block holds the pointer value. Because the top output of the T→R block is passing power, the first ADD block receives power, causing the value copied to 40204 to be added to 40202. Register 40202 equals 0 to start.

This routine continues until the pointer value in the T→R block (40203) equals the *table length*—84. The middle output in the T→R block then passes power, and the DIV block receives power. The values in registers 40201 and 40202 are divided by 84 (the value in the middle node of the DIV block). The *result* is placed in register 40301, and the *remainder* is placed in register 40302. Because the middle input of the DIV block is receiving power, the *remainder* is expressed as a decimal.

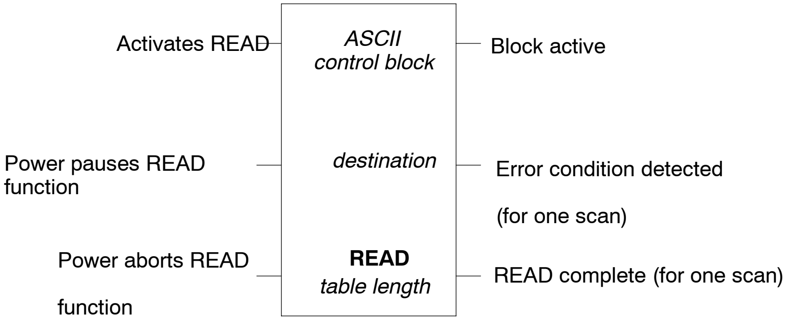
The top output of the DIV block passes power, and the XOR block receives power. By using the XOR function to exclusively OR the values in matrix 40201 ... 40203 with themselves, you clear the matrix to 0. The top output of the XOR block passes power to coil 00003, indicating that the current table averaging operation is complete and that a new one should start.

Chapter 13
ASCII READ/WRITE Functions

- ❒ ASCII Message Handling
- ❒ READ
- ❒ WRIT
- ❒ ASCII Error Status
- ❒ How the READ/WRIT Blocks Handle ASCII Messages
- ❒ The ASCII Character Set

13.1 READ

The READ instruction provides the ability to read data entered at an ASCII device through the RIO interface and into 984 Memory. READ is a three-node function block:



Caution Make sure that no two ASCII READ/WRIT function blocks are active in the same segment at the same time—such a condition will cause the block to return an error or return bad data.

The first register in the *ASCII control block* is specified in the top node. It is the first of seven consecutive (4*x*) holding registers:

Register	Definition
4*x*	bits 0 ... 5 = *port number* (1 ... 32); bits 6 ... 15 = error code
4*x* + 1	message number
4*x* + 2	number of registers required to satisfy format
4*x* + 3	number of registers transmitted thus far
4*x* + 4	status of solve
4*x* + 5	unassigned
4*x* + 6	checksum of registers 0 ... 5

The *destination* register in the middle node is the first in a table of (4*x*) holding registers whose length is determined by the value in the bottom node. Variable data in a READ message are written into this table.

Consider this READ message:

```
please enter password:AAAAAAAAAA
    (Embedded Text)      (Variable Data)
```

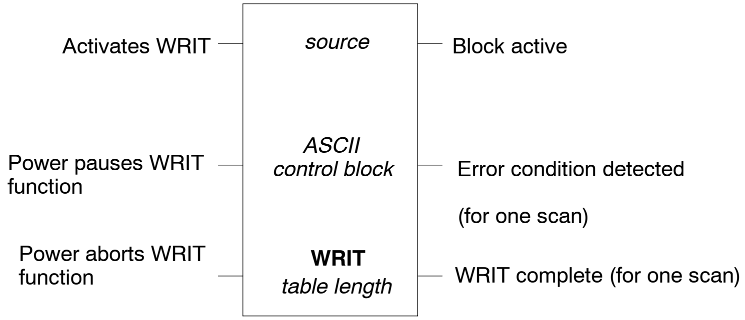
☞ **Note** An ASCII READ message may contain the embedded text—placed inside quotation marks—as well as the variable data in the format statement—i.e., the ASCII message.

The 10-character ASCII field **AAAAAAAAAA** is the variable data field; variable data must be entered via an ASCII input device.

The bottom node indicates that this is an ASCII READ function, and it contains a number specifying *length* of the *destination* table. Table length may range from 1 ... 255 in a 16 bit CPU and from 1 ... 999 in a 24 bit CPU.

13.2 WRIT

The WRIT instruction provides the ability to send a message from the 984 controller over the RIO communications link to an ASCII device. WRIT is a three-node function block:



Caution Make sure that no two ASCII READ/WRIT function blocks are active in the same segment at the same time—such a condition will cause the block to return an error or return bad data.

The *source* register in the top node may be either the first (3*x*) input register or the first (4*x*) holding register in a table whose length is specified in the bottom node.

This table will contain the data required to fill the variable field in a message. Consider the following WRIT message

```
vessel #1 temperature is:III
```

The 3-character ASCII field `III` is the variable data field; variable data are loaded, typically via DX moves, into a table of variable field data.

The *ASCII control block* register specified in the middle node is the first of seven consecutive (4*x*) holding registers:

Register	Definition
4*x*	bits 0 ... 5 = *port number* (1 ... 32); bits 6 ... 15 = error codes
4*x* + 1	message number
4*x* + 2	number of registers required to satisfy format
4*x* + 3	number of registers transmitted thus far
4*x* + 4	status of solve
4*x* + 5	unassigned
4*x* + 6	checksum of registers 0 ... 5

The bottom node indicates that this is an ASCII READ function, and it contains a number specifying *length* of the *source* table. Table length may range from 1 ... 255 in a 16 bit CPU and from 1 ... 999 in a 24 bit CPU.

13.3 ASCII Message Handling

The ASCII READ and WRIT function blocks provide the routines necessary for communication between the ASCII message table in 984 system memory and an RIO interface module that supports ASCII at your RIO drops (such as a J812, J892, P892, or P453). These routines verify correct ASCII parameters—for example, port # and message #—lengths of variable data fields, error detection and recording, and RIO interface status.

Each function requires two tables of registers: one to retrieve and store variable data and the other to identify which port and message numbers are to be used. The port and message table contains seven registers, and the size of the variable data table needs to be specified. The balance of the registers is used for housekeeping.

The 984 provides support logic to monitor the status of a READ or WRIT function, detect errors, and enable you to take corrective action. Two basic errors that require action are *declared* (detected) errors and *timeout* errors.

13.4 How the READ/WRIT Blocks Handle ASCII Messages

Once a READ or WRIT block has been activated (power transitioned from low to high at the top input), you may remove power from the node; the block remains active for as many scans as are necessary to complete the message transaction. Power at the middle or bottom input will stop the function.

When the middle input receives power, the READ/WRIT function pauses—i.e., the middle input *deactivates* the function. When power is removed from the middle input, the READ/WRIT function continues from where it was interrupted unless there has been some communication at the port during the pause. If there has been communication, the message transaction starts at the beginning.

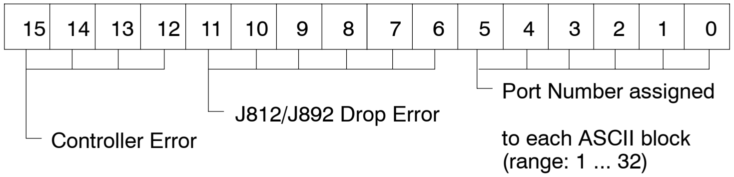
When the bottom input receives power, the READ/WRIT function is aborted. The middle output (error condition detected) passes power for one scan, then loads the four most significant bits of the register specified in the top node with error code 6:

```
user initiated abort
```

To restart an ASCII READ/WRIT function after an abort, the top input must be cycled from low to high.

13.5 ASCII Error Status

When an ASCII message is aborted because of a communication error, an error code gets stored in the 984. To retrieve the error code for an aborted ASCII block, use your programming panel or DAP to display the contents of the register holding the error word. To retrieve an aborted READ block, go to the first register of the *source* node; to retrieve an aborted WRIT block, go to the first register of the *destination* node.



Bits 15 ... 12 (HEX)	**Controller Error**
1	An error has been detected in the input to the RIO interface from the ASCII device.
2	An exception response from the RIO interface indicates bad data.
3	A sequenced number from the addressed RIO interface differs from the expected value.
4	There is a user register checksum error—often caused by altering READ/WRIT registers while the block is active.
5	An invalid port or message number has been detected.
6	A user-initiated abort is indicated; the bottom input of the READ/ WRIT block is energized.
7	No response from the drop indicates a communication error.
8	A node has aborted because of the use of the SKP function.
9	The ASCII message area has been scrambled. Reload memory.
A	A port has not been configured in the traffic cop (*J892 only*).
B	This error indicates an illegal ASCII request *(J892 only*).
C	An unknown response has been received from the ASCII port (*J892 only*).
D	An illegal ASCII element has been detected in user logic— e.g., Duplicate Block.
F	The (S901 or S908) RIO processor in the 984 is down.

Bits 11 ... 6	J812/J892 Drop Error
11	The input from the ASCII device is not compatible with the specified format.
10	There is an input buffer overrun—data are being received too quickly at the (J812/J892) RIO interface.
9	A USART error has been detected—a bad byte has been received at the (J812/J892) RIO interface.
8	An illegal format has been processed—the format has not been received properly by the (J812/J892) RIO interface.
7	The ASCII device is off-line—it has been turned off, disconnected, put into off-line operation, or has activated normal handshaking. Check the cabling to the device.
6	An ASCII message has terminated early (*keyboard mode only*).

ASCII Character Code Chart

Dec	Octal	Hex	Name	Dec	Octal	Hex	Symbol
0	000	00	NUL (null)	64	100	40	@
1	001	01	SOH (start of heading)	65	101	41	A
2	002	02	STX (start of text)	66	102	42	B
3	003	03	ETX (end of text)	67	103	43	C
4	004	04	EOT (end of transmission)	68	104	44	D
5	005	05	ENQ (enquiry)	69	105	45	E
6	006	06	ACK (acknowledge)	70	106	46	F
7	007	07	BEL (bell or audio tone)	71	107	47	G
8	010	08	BS (backspace)	72	110	48	H
9	011	09	HT (horizontal tab)	73	111	49	I
10	012	0A	LF (line feed)	74	112	4A	J
11	013	0B	VT (vertical tab)	75	113	4B	K
12	014	0C	FF (form feed)	76	114	4C	L
13	015	0D	CR (carriage return)	77	115	4D	M
14	016	0E	SO (shift out (red ribbon))	78	116	4E	N
15	017	0F	SI (shift in (black ribbon))	79	117	4F	O
16	020	10	DLE (data link escape)	80	120	50	P
17	021	11	DC1 (device control 1 (X-ON))	81	121	51	Q
18	022	12	DC2 (device control 2 (aux-ON))	82	122	52	R
19	023	13	DC3 (device control 3 (X-OFF))	83	123	53	S
20	024	14	DC4 (device control 4 (aux-OFF))	84	124	54	T
21	025	15	NAK (negative acknowledge (error))	85	125	55	U
22	026	16	SYN (synchronous file)	86	126	56	V
23	027	17	ETB (end of transmission block)	87	127	57	W
24	030	18	CAN (cancel)	88	130	58	X
25	031	19	EM (end of medium)	89	131	59	Y
26	032	1A	SUB (substitute)	90	132	5A	Z
27	033	1B	ESC (escape)	91	133	5B	[
28	034	1C	FS (file separator)	92	134	5C	\
29	035	1D	GS (group separator)	93	135	5D	]
30	036	1E	RS (record separator)	94	136	5E	^
31	037	1F	US (unit separator)	95	137	5F	_
32	040	20	SP (space)	96	140	60	‘
33	041	21	!	97	141	61	a
34	042	22	”	98	142	62	b
35	043	23	#	99	143	63	c
36	044	24	$	100	144	64	d
37	045	25	%	101	145	65	e
38	046	26	&	102	146	66	f
39	047	27	’	103	147	67	g
40	050	28	(	104	150	68	h
41	051	29	)	105	151	69	i
42	052	2A	*	106	152	6A	j
43	053	2B	+	107	153	6B	k
44	054	2C	,	108	154	6C	l
45	055	2D	-	109	155	6D	m
46	056	2E	.	110	156	6E	n
47	057	2F	/	111	157	6F	o
48	060	30	0	112	160	70	p
49	061	31	1	113	161	71	q
50	062	32	2	114	162	72	r
51	063	33	3	115	163	73	s
52	064	34	4	116	164	74	t
53	065	35	5	117	165	75	u
54	066	36	6	118	166	76	v
55	067	37	7	119	167	77	w
56	070	38	8	120	170	78	x
57	071	39	9	121	171	79	y
58	072	3A	:	122	172	7A	z
59	073	3B	;	123	173	7B	{
60	074	3C	<	124	174	7C	|
61	075	3D	=	125	175	7D	}
62	076	3E	>	126	176	7E	~
63	077	3F	?	127	177	7F	DEL (delete)

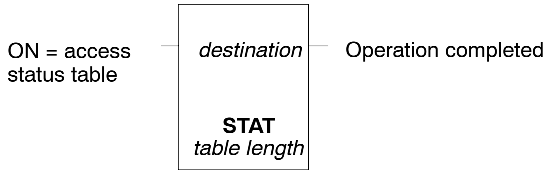
Chapter 14
Monitoring System Status

14.1 The STAT Function

The STAT instruction lets you access the 984 status table in system memory; here vital system diagnostic information is written into a table of registers or discretes, as specified in the *destination* node. This information includes

- Controller status
- Possible error conditions in the I/O modules
- Input-to-controller-to-output communication status

STAT is a two-node function block:



The top *destination* node, where the first word of system status is written, may be

- The first 0*x* in a table of discrete output references
- The first 4*x* in a table of holding registers

Caution We recommend that you do not use discretes in the STAT *destination* node because of the excessive number required to contain status information.

The bottom node indicates that this is a STAT function and specifies the number of registers in the table where status information will be written. The *table length* ranges from 1 ... 75 for controllers using the *S901* RIO protocol and 1 ... 277 for controllers using the *S908* protocol. The *table length* that can actually be read by the STAT block depends on the addressing capabilities of the controller—a 16 bit CPU can access only up to the first 255 words in the STAT table, whereas a 24 bit CPU can access all 277 words.

14.2 The S901 Status Table

The 75 words in the S901 status table are divided into three sections—the first 11 words for controller status information, the next 32 words for I/O module health information, and the last 32 words for I/O communications information:

DECIMAL WORD			HEX WORD
1	Controller Status		01
2	Unused		02
3	Controller Status		03
4	S901 Status		04
5	Controller Stop State		05
6	Number of Segments in User Logic		06
7	Address of End-0f-Logic Pointer		07
8	RIO Redundancy and Timeout		08
9	ASCII Message Status		09
10	Run Load Debug Status		0A
11	Address of Status Word Pointer Table		0B
12	Channel 1 Input	Channel 2 Input	0C
13	Channel 3 Input	Channel 4 Input	0D
14	Channel 5 Input	Channel 6 Input	0E
	” ” ” ” ” ” ” ” ”	” ” ” ” ” ” ” ”	”
27	Channel 29 Input	Channel 30 Input	1B
28	Channel 31 Input	Channel 32 Input	1C
29	Channel 1 Output	Channel 2 Output	1D
30	Channel 3 Output	Channel 4 Output	1E
31	Channel 5 Output	Channel 6 Output	1F
	” ” ” ” ” ” ” ” ”	” ” ” ” ” ” ” ”	”
42	Channel 29 Output	Channel 30 Output	2A
43	Channel 31 Output	Channel 32 Output	2B
44	Remote I/O Channels 5 and 6 First Word		2C
45	Remote I/O Channels 5 and 6 Second Word		2D
46	Remote I/O Channels 7 and 8 First Word		2E
47	Remote I/O Channels 7 and 8 Second Word		2F
	” ” ” ” ” ” ” ” ” ” ” ” ” ” ” ” ” ” ”		”
70	Remote I/O Channels 31 and 32 First Word		46
71	Remote I/O Channels 31 and 32 Second Word		47
72	Remote I/O Channels 1 and 2 First Word		48
73	Remote I/O Channels 1 and 2 Second Word		49
74	Remote I/O Channels 3 and 4 First Word		4A
75	Remote I/O Channels 3 and 4 Second Word		4B

14.3 Accessing S901 Status Data with a Programming Panel

Status words 1 ... 11 can be found in sequential memory starting at absolute memory location 65 (hex). The system keeps a status block pointer in absolute memory location 6F (hex); it points to a table of addresses 76 words long. Addresses 2 ... 76 point to status words 1 ... 75, respectively.

Procedure **Locating a Status Word with a Programming Panel**

Step 1 Read the pointer stored in location 6F.

Step 2 Add the status word number to the pointer.

Step 3 If the most significant hex digit of the pointer is > 8, add E8000 to the pointer as follows:

Pointer	Address
8*xxx*	F0*xxx*
9*xxx*	F1*xxx*
A*xxx*	F2*xxx*
B*xxx*	F3*xxx*
C*xxx*	F4*xxx*
D*xxx*	F5*xxx*
Exxx	F6*xxx*
Fxxx	F7*xxx*

xxx = last three digits of the pointer become last three digits of the address

For example, pointer B984 becomes address F3984.

Step 4 Read the pointer from the pointer table.

Step 5 If the most significant hex digit of the pointer is > 8, convert the address using the procedure described in Step 3.

Step 6 Read the status word from system memory.

14.4 Accessing S901 Status Data with a P965 DAP

Status words 1 ... 11 can be found in sequential memory starting at absolute memory location 300101 (decimal). The system keeps a status block pointer in absolute memory location 300111 (decimal); it points to a table of addresses 76 words long. Addresses 2 ... 76 point to status words 1 ... 75.

Procedure **Locating a Status Word with a P965 DAP**

Step 1 Read the pointer stored in location 300111.

Step 2 Add the status word number to the pointer.

Step 3 Add 300000 to the pointer as follows:

Pointer	Address
xxxxx	3*xxxxxx*

where the last five digits (*xxxxx*) of the pointer become the last five digits of the address. For example, pointer 00984 becomes address 300984.

Step 4 Read the pointer from the pointer table.

Step 5 Convert the address using the procedure described in Step 3.

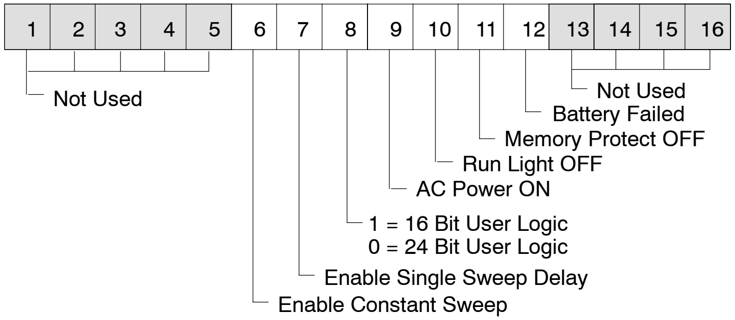
Step 6 Read the status word from system memory.

14.5 S901 Controller Status Words

Words 1 ... 11 display the controller status words:

Word 1 Displays the following aspects of the controller’s status:

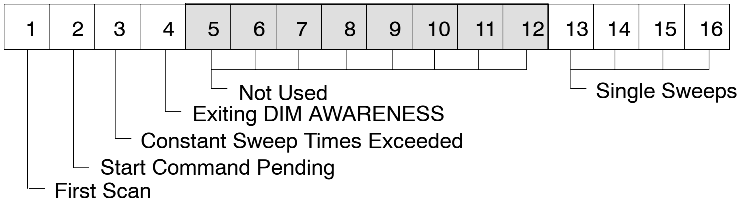
If the bit is set to **1**, then the condition is TRUE.



Word 2 is not used, and therefore all bit values are 0.

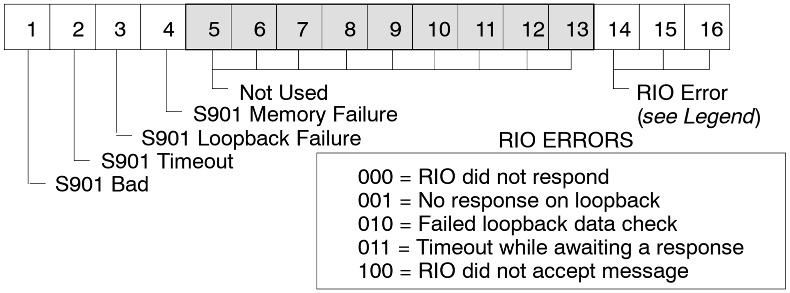
Word 3 Displays the following aspects of the controller’s status:

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



Word 4 Displays the status of the S901 Remote I/O Processor:

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

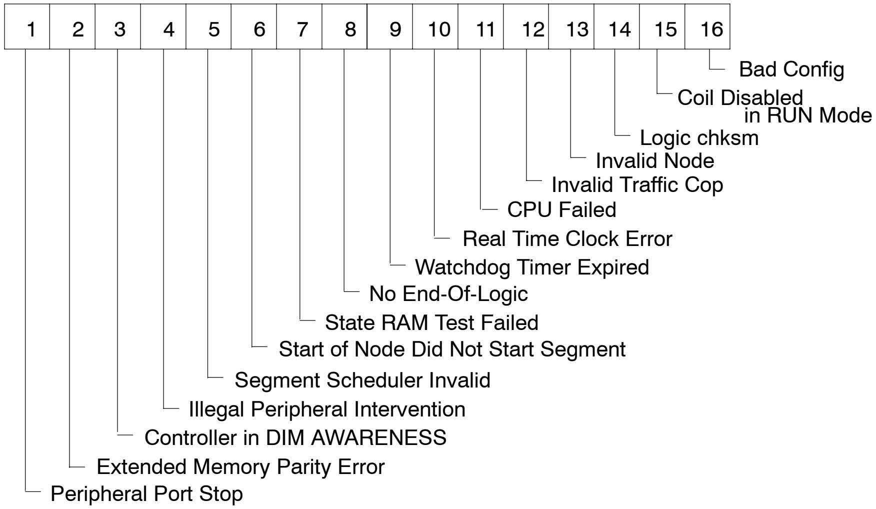


RIO ERRORS

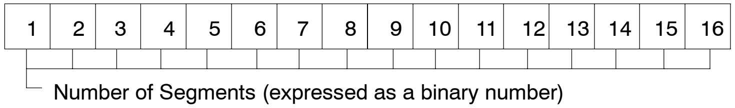
000 = RIO did not respond
001 = No response on loopback
010 = Failed loopback data check
011 = Timeout while awaiting a response
100 = RIO did not accept message

Word 5 Displays the controller's stop state conditions:

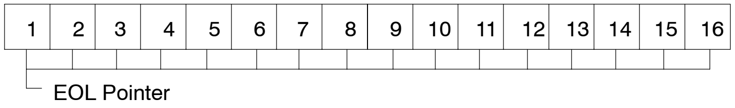
If the bit is set to 1, then the condition is TRUE.



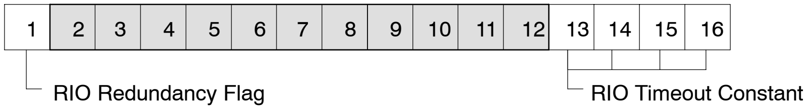
Word 6 Displays the number of logic segments:



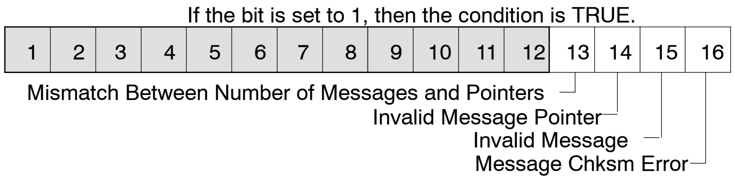
Word 7 Displays the end-of-logic (EOL) pointer:



Word 8 Holds a RIO redundancy flag and displays an RIO timeout constant:

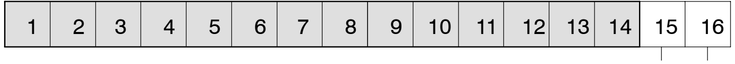


Word 9 Displays the ASCII message status:



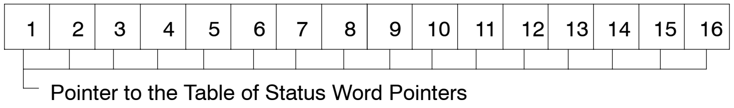
Word 10 Uses its two most significant bits to display the RUN load debug status:

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



Debug = 0 0
Run = 0 1
Load = 1 0

Word 11 Displays the address of the table of status word pointers:

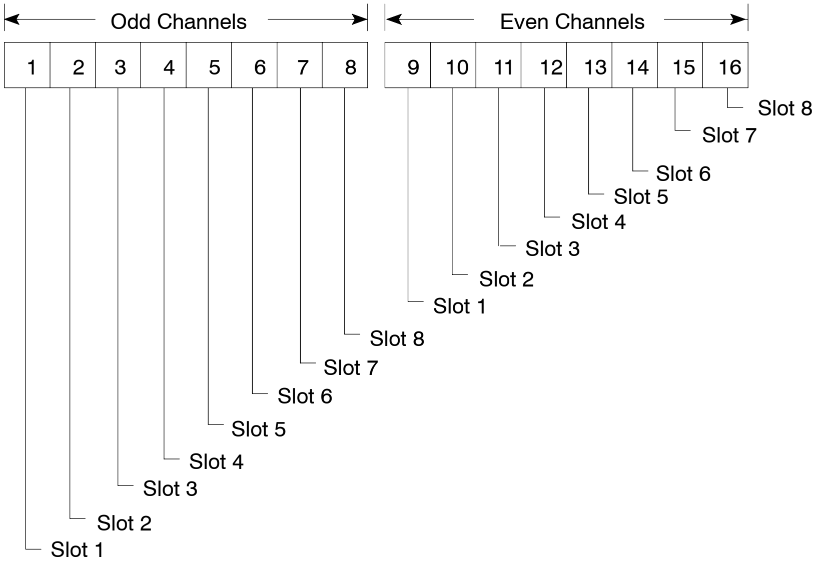


14.6 S901 I/O Module Health Status Words

Words 12 ... 43 display the health of the I/O modules in the odd and even channels:

12	Channel 1 Input	Channel 2 Input	0C
13	Channel 3 Input	Channel 4 Input	0D
14	Channel 5 Input	Channel 6 Input	0E
	” ” ” ” ” ” ” ” ”	” ” ” ” ” ” ” ”	”
26	Channel 29 Input	Channel 30 Input	1B
27	Channel 31 Input	Channel 32 Input	1C
28	Channel 1 Output	Channel 2 Output	1D
29	Channel 3 Output	Channel 4 Output	1E
30	Channel 5 Output	Channel 6 Output	1F
	” ” ” ” ” ” ” ” ”	” ” ” ” ” ” ” ”	”
42	Channel 29 Output	Channel 30 Output	2A
43	Channel 31 Output	Channel 32 Output	2B

Each of these 32 status words is organized as follows:



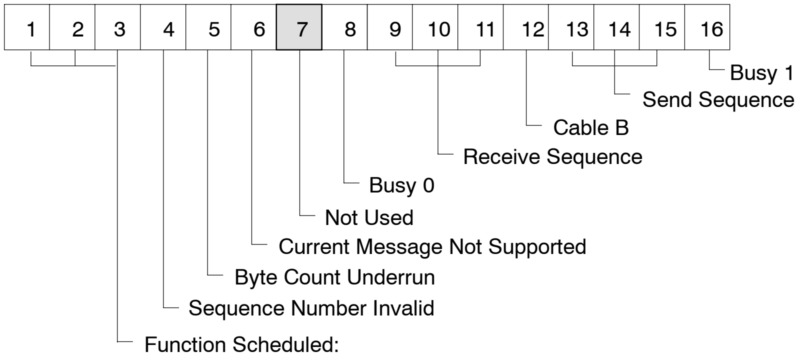
If a specified slot is inhibited in the traffic cop, the bit is 0. If the slot contains an input module or an input/output module, the bit is 1. If the slot contains an output module and the module's **COMM ACTIVE** LED is ON, the bit is 0; if slot contains an output module and the module's **COMM ACTIVE** LED is OFF, the bit is 1.

☞ **Note** These indicators are valid only when scan time > 30 ms.

14.7 S901 RIO Communication Status Words

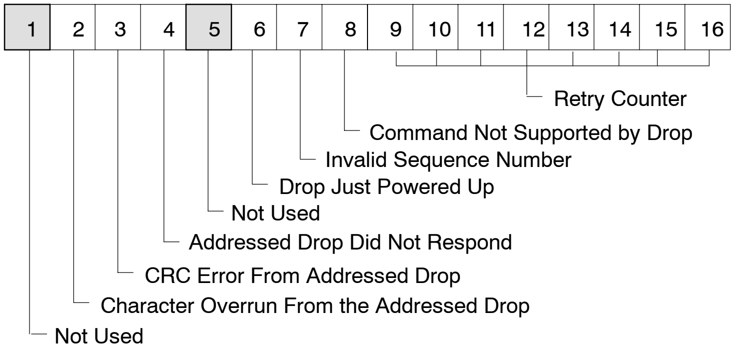
RIO system communication status is given in words 44 ... 75. Two words are used to describe each of up to 16 drops:

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



000 = Normal I/O
001 = Restart (Comm Reset)
010 = Restart (Application Reset)
011 is unassigned
100 = Inhibit
101 unassigned
110 unassigned
111 unassigned

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



14.8 The S908 Status Table

The 277 words in the S908 status table are organized in three sections—the first 11 words for controller status, the next 160 words for I/O module health, and the last 106 words for I/O communication health:

DECIMAL WORD		HEX WORD
1	Controller Status	01
2	Hot Standby Status	02
3	Controller Status	03
4	RIO Status	04
5	Controller Stop State	05
6	Number of Ladder Logic Segments	06
7	End-of-logic Pointer Address	07
8	RIO Redundancy and Timeout / Memory Sizing Word for Panel (in the 984-145 Compact Controller)	08
9	ASCII Message Status	09
10	Run/Load/Debug Status	0A
11	Not used	0B
12	Drop 1, Rack 1	0C
13	Drop 1, Rack 2	0D
14	Drop 1, Rack 3	0E
15	Drop 1, Rack 4	0F
16	Drop 1, Rack 5	10
17	Drop 2, Rack 1	11
18	Drop 2, Rack 2	12
	„ „ „ „ „ „ „ „ „	
170	Drop 32, Rack 4	AA
171	Drop 32, Rack 5	AB
172	S908 Startup Error Code	AC
173...175	Cable A Errors	AD...AF
176...178	Cable B Errors	B0...B2
179...181	Global Communication Errors	B3...B5
182...184	Drop 1 Errors / Health Status and Retry Counters (in the Compact 984 Controllers)	B6...B8
185...187	Drop 2 Errors	B9...BB
188...190	Drop 3 Errors	BC...BE
272...274	Drop 31 Errors	110...112
275...277	Drop 32 Errors	113...115

14.9 Accessing S908 Status Data with a Programming Panel

When accessing the status table from your programming panel, words 1 ... 11 are found in sequential memory locations 65 ... 6F (hex). The I/O health status table is kept in 160 sequential memory locations; the communication status table is kept in 106 sequential memory locations. The actual memory locations that hold these two tables will vary with different 984 mainframe models.

Use pointers to locate the first word in the I/O module health status table and the communication status table. The pointers are always found at the same locations in absolute memory:

- I/O module health pointer—location 46 (hex)
- I/O communication pointer—location 33 (hex)

If the most significant hex digit of the pointer is ≥ 8, add E8000 to the pointer as follows:

Pointer	**Address**
8*xxx*	f0*xxx*
9*xxx*	f1*xxx*
a*xxx*	f2*xxx*
b*xxx*	f3*xxx*
c*xxx*	f4*xxx*
d*xxx*	f5*xxx*
e*xxx*	f6*xxx*
f*xxx*	f7*xxx*

xxx = last three digits of the pointer become the last three digits of the address

For example, pointer B984 becomes address F3984

To find the address of an I/O health status word, subtract 0C (hex) from the status word number, then add the result to the I/O health pointer.

To find the address of a communication status word, subtract 0AC (hex) from the status word number, then add the result to the I/O communication pointer.

14.10 Accessing S908 Status Data with a P965 DAP

If you are accessing the status table with a P965 DAP, words 1 ... 11 can be found in absolute memory locations 300101 ... 300111 (decimal). The I/O health status table is kept in 160 sequential memory locations; the communication status table is kept in 106 sequential memory locations. The actual memory locations that hold these two tables will vary with different 984 controllers.

Use pointers to locate the first word in the I/O module health status table and the communication status table. The pointers are always found at the same locations in absolute memory:

- I/O module health pointer—location 300070
- I/O communication pointer—location 300051

Add 300000 to the pointer as follows:

Pointer	Address
xxxxx	3*xxxxxx*

where the last five digits (*xxxxx*) of the pointer become the last five digits of the address. For example, pointer 00984 becomes address 300984.

To find the address of an I/O health status word, subtract 12 from the status word number, then add the result to the I/O health status pointer.

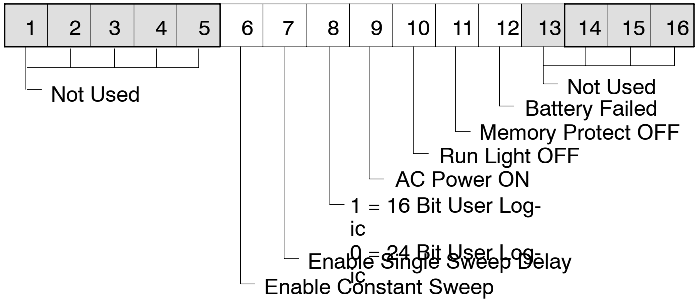
To find the address of a communication status word, subtract 172 from the status word number, then add the result to the I/O communication pointer.

14.11 S908 Controller Status Words

Words 1 ... 11 display the controller status words.

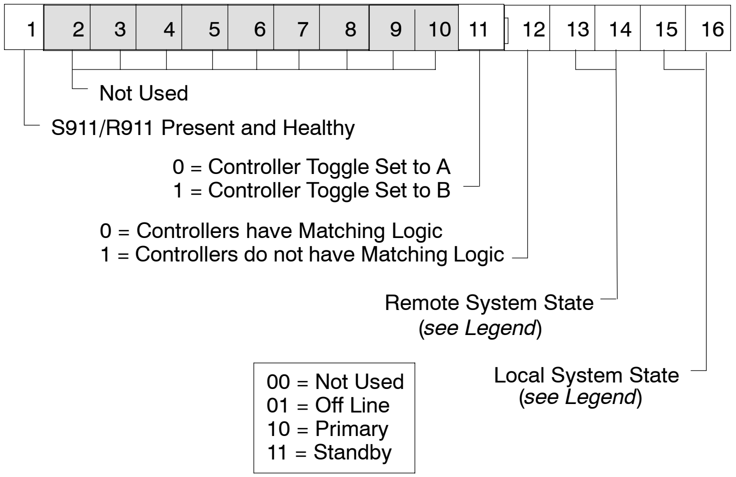
Word 1 Displays the following aspects of the controller's status:

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



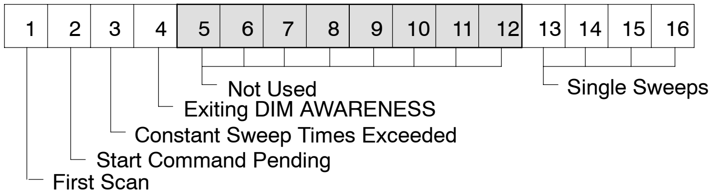
Word 2 Displays the Hot Standby status for 984 controllers that use S911/R911 Modules:

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



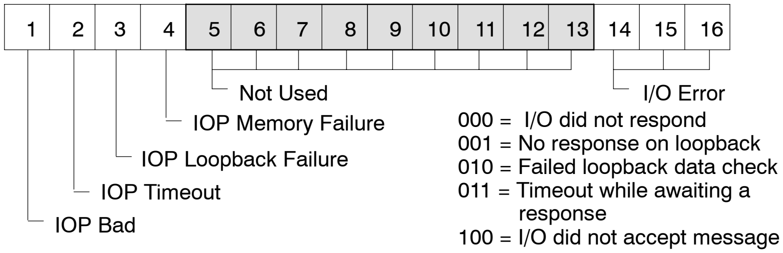
Word 3 Displays more aspects of the controller status:

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



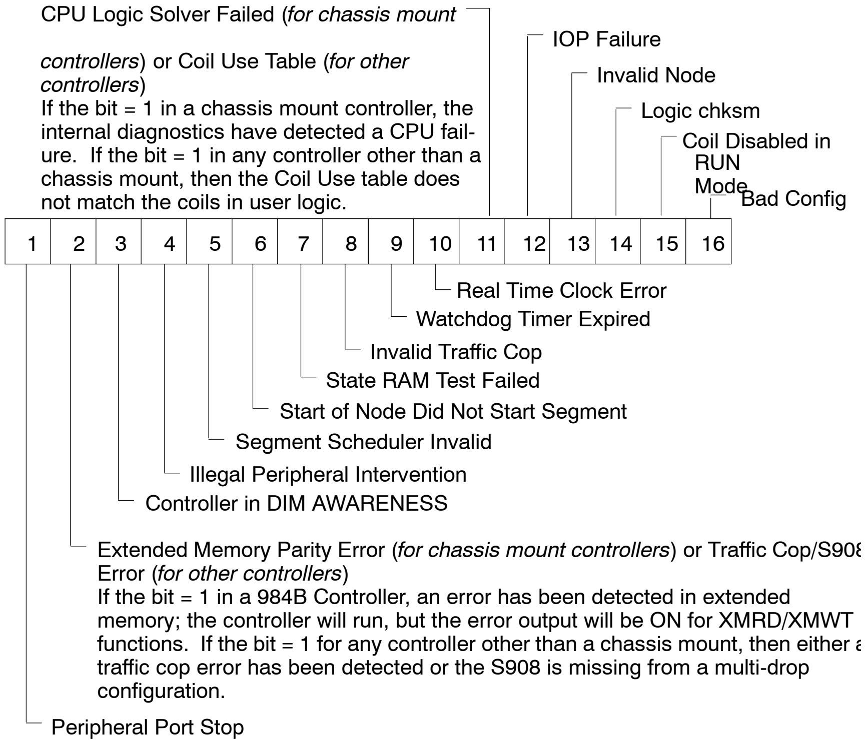
Word 4 Displays the status of the I/O processor in the 984 controller:

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



Word 5 Displays the controller’s stop state conditions:

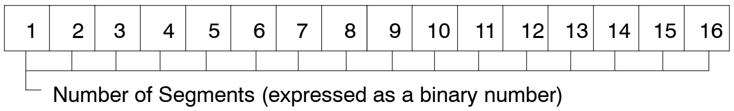
If the bit is set to 1, then the condition is TRUE.



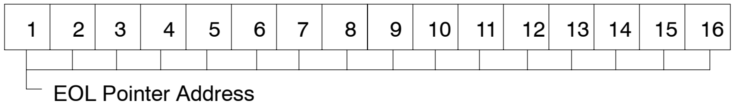
CPU Logic Solver Failed (*for chassis mount controllers*) or Coil Use Table (*for other controllers*)
If the bit = 1 in a chassis mount controller, the internal diagnostics have detected a CPU failure. If the bit = 1 in any controller other than a chassis mount, then the Coil Use table does not match the coils in user logic.

Extended Memory Parity Error (*for chassis mount controllers*) or Traffic Cop/S90
Error (*for other controllers*)
If the bit = 1 in a 984B Controller, an error has been detected in extended memory; the controller will run, but the error output will be ON for XMRD/XMWT functions. If the bit = 1 for any controller other than a chassis mount, then either traffic cop error has been detected or the S908 is missing from a multi-drop configuration.

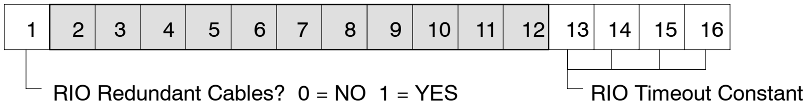
Word 6 Displays the number of segments in ladder logic; a binary number is shown:



Word 7 Displays the address of the end-of-logic (EOL) pointer:

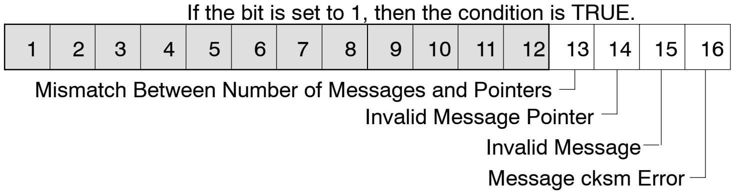


Word 8 *In controllers that support remote I/O*, word 8 uses its most significant bit to display whether or not redundant coaxial cables are run to the remote I/O drops, and it uses its four least significant bits to display the remote I/O timeout constant:

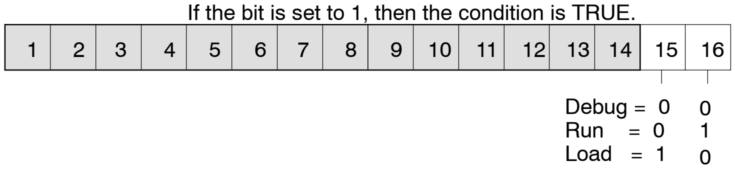


In the Compact 984-145 Controller, word 8 is used to store a numerical value that defines the upper limit of memory locations on page 0 where user logic can be placed. This value is not user-configurable and is used only by the programming panel.

Word 9 Uses its four least significant bits to display ASCII message status:



Word 10 Uses its two least significant bits to display RUN/LOAD/DEBUG status:



Word 11 is not used.

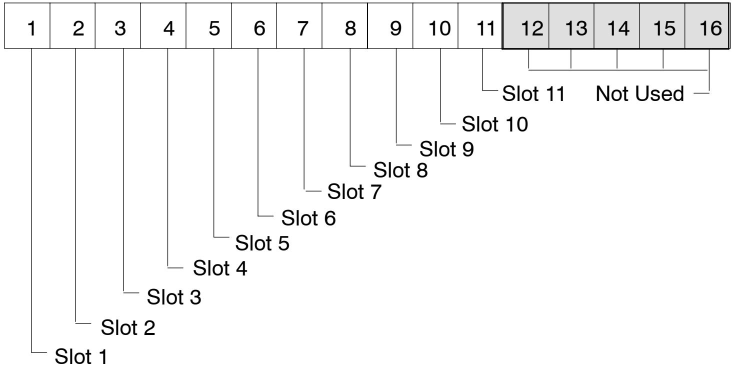
14.12 S908 I/O Module Health Status Words

Status words 12 ... 171 display I/O module health status:

12	Drop 1 Rack 1
13	Drop 1 Rack 2
14	Drop 1 Rack 3
15	Drop 1 Rack 4
16	Drop 1 Rack 5
17	Drop 2 Rack 1
18	Drop 2 Rack 2
	” ” ” ” ” ” ”
170	Drop 32 Rack 4
171	Drop 32 Rack 5

Five words are reserved for each of up to 32 drops, one word for each of up to five possible racks (I/O housings) in each drop. Each rack may contain up to 11 I/O modules; bits 1 ... 11 in each word represent the health of the associated I/O module in each rack.

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



Four conditions must be met before an I/O module can indicate good health:

- The slot must be traffic copped
- The slot must contain a module with the correct personality
- Valid communications must exist between the module and the RIO interface at remote drops
- Valid communications must exist between the RIO interface at each remote drop and the I/O processor in the controller

14.12.1 Converting from Word # to Drop and Rack

$$\frac{\textit{Word \#} - 12}{5} = \textit{Quotient} + \textit{Remainder}$$

where

Drop # = Quotient + 1
Rack # = Remainder + 1

14.12.2 Converting from Drop and Rack to Word

Word # = (*Drop #* x 5) + *Rack #* + 6

14.12.3 Status Words for the MMI Operator Panels

The status of the 32 Element Pushbutton Panels and PanelMate units on an RIO network can also be monitored with an I/O health status word. The Pushbutton Panels occupy slot 4 in an I/O rack and can be monitored at bit 4 of the appropriate status word. A PanelMate on RIO occupies slot 1 in rack 1 of the drop and can be monitored at bit 1 of the first status word for the drop.

Note The ASCII Keypad's communication status can be monitored with the error codes in the ASCII READ/WRIT blocks (see Section 13.5).

14.13 S908 I/O Communication Status Words

Status words 172 ... 277 contain the I/O system communication status. Words 172 ... 181 are global status words. Among the remaining 96 words, three words are dedicated to each of up to 32 drops, depending on the type of 984 controller you are using.

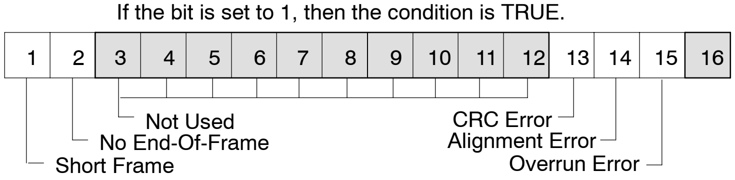
Word 172 *S908 Startup Error Code*. This word is always 0 when the system is running. If an error occurs, the controller does not start—it generates a stop state code of 10 (word 5):

Traffic Cop Validation Soft Error Codes

01	BADTCLEN	Traffic Cop length
02	BADLNKNUM	Remote I/O link number
03	BADNUMDPS	Number of drops in Traffic Cop
04	BADTCSUM	Traffic Cop checksum
10	BADDDLEN	Drop descriptor length
11	BADDRPNUM	I/O drop number
12	BADHUPTIM	Drop holdup time
13	BADASCNUM	ASCII port number
14	BADNUMODS	Number of modules in drop
15	PRECONDRP	Drop already configured
16	PRECONPRT	Port already configured
17	TOOMNYOUT	More than 1024 output points
18	TOOMNYINS	More than 1024 input points
20	BADSLTNUM	Module slot address
21	BADRCKNUM	Module rack address
22	BADOUTBC	Number of output bytes
23	BADINBC	Number of input bytes
25	BADRF1MAP	First reference number
26	BADRF2MAP	Second reference number
27	NOBYTES	No input or output bytes
28	BADDISMAP	Discrete not on 16-bit boundary
30	BADODDOUT	Unpaired odd output module
31	BADODDIN	Unpaired odd input module
32	BADODDREF	Unmatched odd module reference
33	BAD3X1XRF	1*x* reference after 3*x* register
34	BADDMYMOD	Dummy module reference already used
35	NOT3XDMY	3*x* module not a dummy
36	NOT4XDMY	4*x* module not a dummy
40	DMYREAL1X	Dummy, then real 1*x* module
41	REALDMY1X	Real, then dummy 1*x* module
42	DMYREAL3X	Dummy, then real 3*x* module
43	REALDMY3X	Real, then dummy 3*x* module

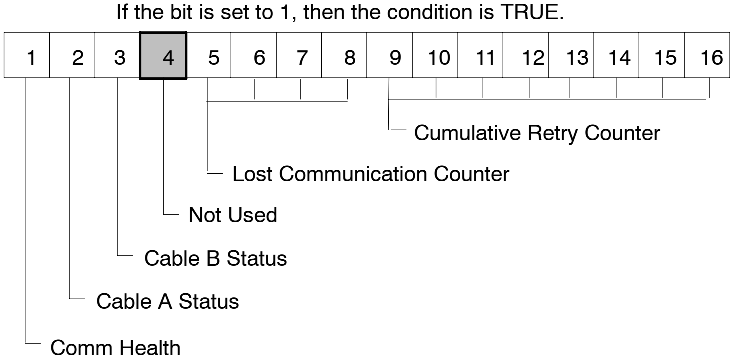
Words 173 ... 175 are Cable A error words:

Word 173	High byte (bits 1 ... 8): Framing error count. Low byte (bits 9 ... 16): DMA receiver overrun count.
Word 174	High byte: Receiver error count. Low byte: Bad drop reception count.
Word 175	Displays the last received LAN error code:



Words 176 ... 178 are Cable B error words:

Word 176	High byte: Framing error count. Low byte (bits 9 ... 16): DMA receiver overrun count.
Word 177	High byte: Receiver error count. Low byte: Bad drop reception count.
Word 178	*Last Received LAN Error Code*: see Word 175 above.
Word 179	Displays global communication status:



Word 180 *Global Cumulative Error Counter* (Cable A):
High byte (bits 1 ... 8): Detected error count.
Low byte (bits 9 ... 16): No response count.

Word 181 *Global Cumulative Error Counter* (Cable B):
High byte: Detected error count.
Low byte: No response count.

For controllers that support remote I/O, words 182 ... 277 are used to describe remote I/O drop status; three status words are used for each drop:

Words 182 ... 184 Assigned to drop 1

Words 185 ... 187 Assigned to drop 2

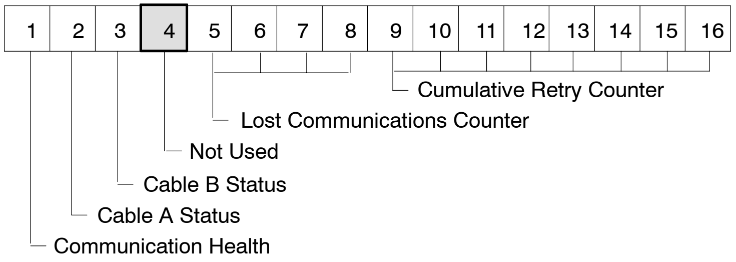
etc.

Words 275 ... 277 Assigned to drop 32

Each group of RIO drop status word is organized as follows:

First Word Displays communication status:

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

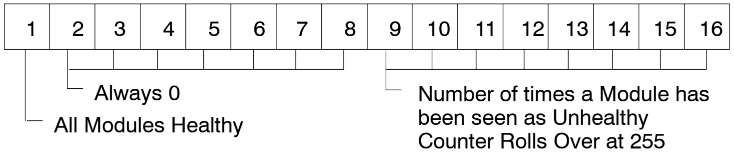


Second Word *Drop Cumulative Error Counter* (Cable A)
High byte (bits 1 ... 8): At least one error has occurred in words 173 ... 175
Low byte (bits 9 ... 16): No response count

Third Word *Drop Cumulative Error Counter* (Cable B)
High byte: At least one error has occurred in words 176 ... 178
Low byte: No response count

For any 984 controller where drop 1 is reserved for local I/O, status words 182 ... 184 are used as follows:

Word 182 Displays local drop status:



Word 183 Used as a 16 bit I/O bus error counter

Word 184 Used as a 16 bit I/O bus retry counter

14.13.1 Converting a Word # to a Drop # or Word

$$\frac{\textit{word \#} - 182}{3} = \textit{quotient} \text{ and } \textit{remainder}$$

quotient + 1 = *drop #*
remainder + 1 = *word*

14.13.2 Converting a Drop # or Word to a Word

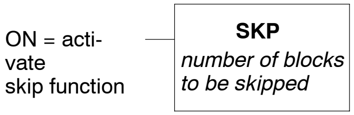
(*drop #* x 3) + *word* + 178 = *word #*

Chapter 15
Bypassing Networks with SKP

Warning **SKP is the most dangerous instruction in the 984 instruction set, and it should be used carefully. 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.**

15.1 SKP

With the SKP instruction, you can bypass networks in your ladder logic program and not solve the skipped logic. SKP functions allow you to reduce scan time and, in effect, establish subroutines in the logic. The SKP instruction is a one-node function block:



The node indicates that this is a SKP function and specifies the number of networks to be skipped—this number must include the network that contains the SKP instruction. The number can be

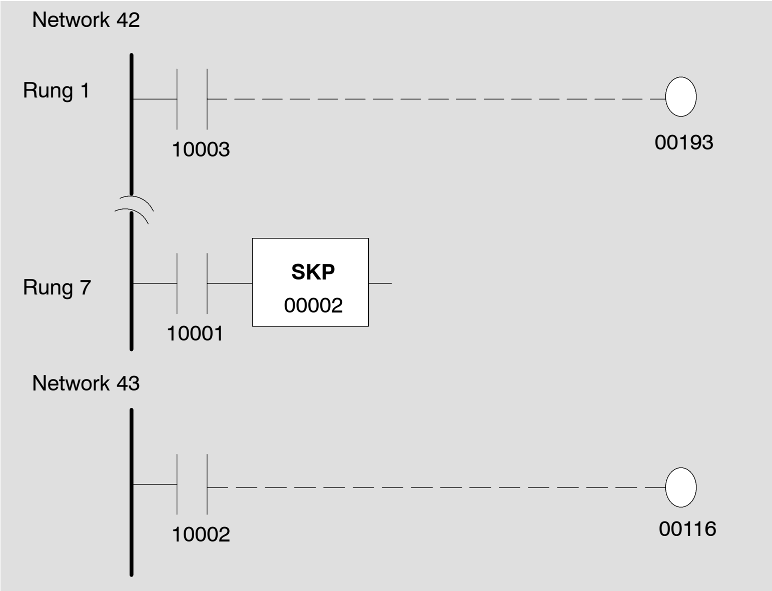
- A decimal ranging from 1 ... 999
- An input register (3*x*)
- A holding register (4*x*)

When the node is powered, SKP is performed on every scan. This causes the rest of the network containing the SKP block to be skipped (this counts as one network skipped); the CPU continues to skip networks until the total number of networks skipped equals the value specified in the function block.

A SKP operation cannot pass the boundary of a segment. No matter how many extra networks you schedule to be skipped, the instruction will stop if it reaches the end of a segment.

Note A SKP instruction can be activated only if you specify in the configurator editor that skips are allowed.

15.1.1 A Simple SKP Example



When 10001 is closed, the remainder of network 42 and all of network 43 are skipped. The power flow display for these two networks becomes invalid, and your system displays an information message to that effect.

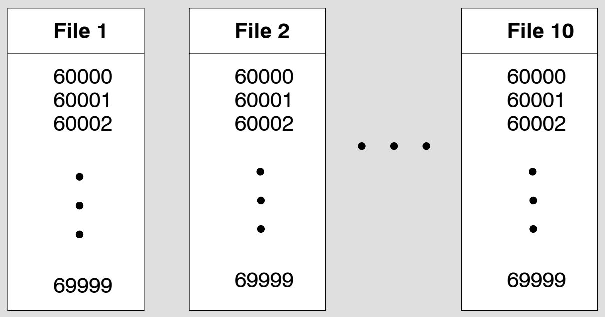
Coil 00193 is still controlled by contact 10003 because the solution of coil 00193 occurs before the SKP instruction.Coil 00116 will remain in whatever state it was in when network 43 was skipped.

Chapter 16
Extended Memory Capabilities

16.1 Extended Memory File Structure

The 984B chassis mount Controller provides an optional capability for supporting *extended memory*. Extended memory is used for massive data storage in a group of files made up of storage registers. These extended memory storage registers use 6*x* reference numbers on pages 1 ... 3 in system memory.

Extended memory provides up to ten files, and each file can contain as many as 10,000 registers ranging from 60000 ... 69999:



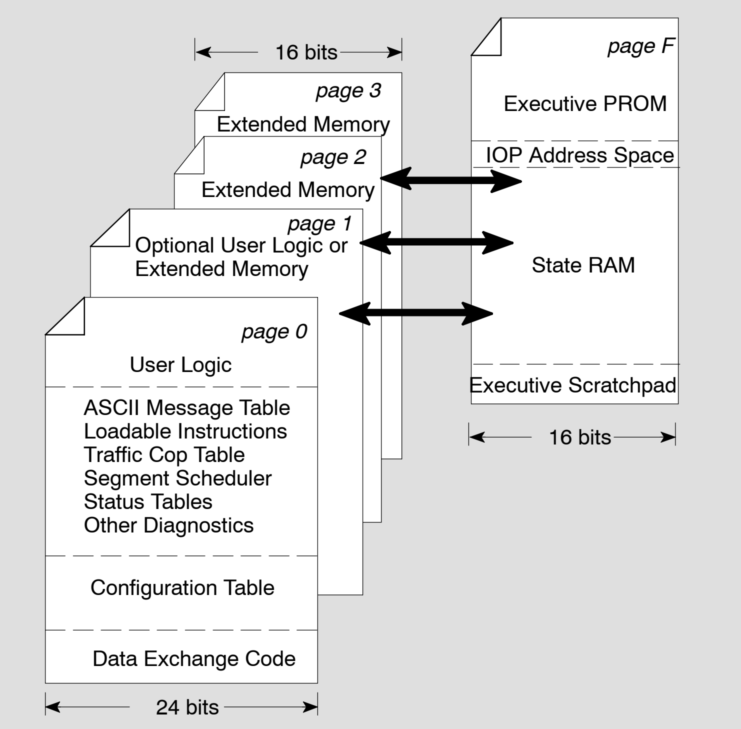
Extended Memory File Structure

Three optional sizes of extended memory are available: 32K words, 64K words, and 96K words. Each 6*x* register uses one word of extended memory. The total memory available may be up to 128K words, with either 32K words or 64K words allocated for user logic memory so that:

- A 984B with 32K words of memory has no extended memory
- A 984B with 64K words of memory may use all 64K for user logic or 32K of user logic and 32K words of extended memory
- A 984B with 96K words of memory may use 32K for user logic and 64K for extended memory or 64K for user logic and 32K for extended memory
- A 984 with 128K words of memory may use 32K for user logic and 96K for extended memory or 64K for user logic and 64K for extended memory

16.2 How Extended Memory Is Stored in User Memory

Extended Memory consists of a bank of memory registers located on pages 1 ... 3 in system memory; these registers may be used as mass storage area for 984 holding registers or as a buffer for input registers. You can store additional state RAM data not being used in a particular application here.



The 984B can be configured for either 32K or 64K words of user logic using the configurator editor in your panel software. If you use 64K, pages 0 and 1 (which contain 24 bit words) are used; if you choose 32K, only page 0 is used. If page 1 is not used for optional user logic in a 984B, it may be used for Extended Memory, along with pages 2 and 3.

☞ **Note** Pages 2 and 3 contain 16 bit words, as do all pages except pages 0 and 1 in a 24 bit machine.

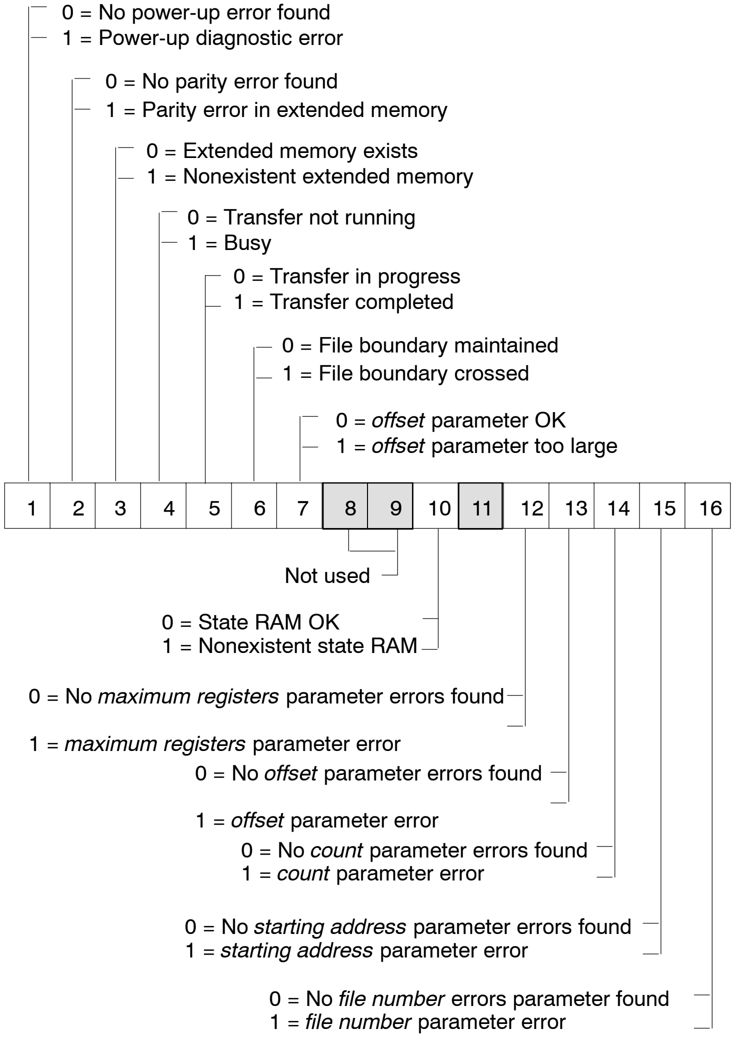
16.3 Extended Memory Control Table

Two additional three-node instructions are included in the 984B executive firmware to be used for manipulating extended memory files—XMWT for writing data into extended memory files and XMRD for reading data from extended memory to state RAM. Both these instructions use a table of six 4*x* holding registers called the *extended memory control table*.

Reference	Register Name	Description
4*x*	*status word*	Contains diagnostic information about extended memory (see illustration on next page)
4*x* + 1	*file number*	Specifies which of the extended memory files is currently in use (range: 1 ... 10)
4*x* + 2	*start address*	Specifies which 6*x* storage register in the current file is the starting address; 0 = 60000, 9999 = 69999
4*x* + 3	*count*	Specifies the number of registers to be read or written in a scan when the appropriate function block is powered; range: 0 ... 9999, not to exceed number specified in *maximum registers* (4*x* + 5)
4*x* + 4	*offset*	Keeps a running total of the number of registers transferred thus far
4*x* + 5	*maximum registers*	Specifies the maximum number of registers that may be transferred when the function block is powered (range: 0 ... 9999)

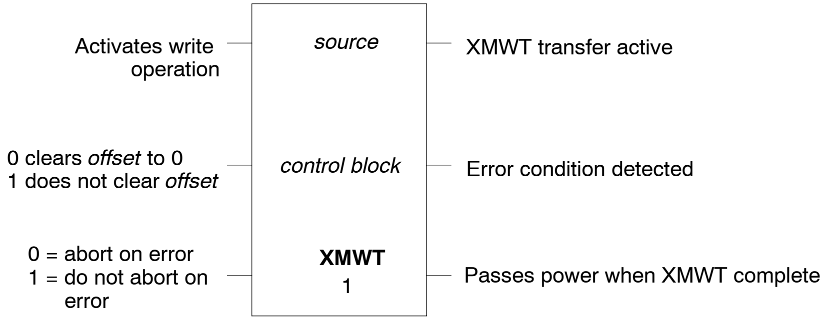
16.3.1 Format of the Extended Memory Status Word

The 16 bit values in the first word in the control table provide you with diagnostic information regarding extended memory:



16.4 Extended Memory Write Function

The XMWT instruction is used to write data from a block of input registers or holding registers in state RAM to a block of 6*x* registers in an extended memory file. It is a three-node function block:



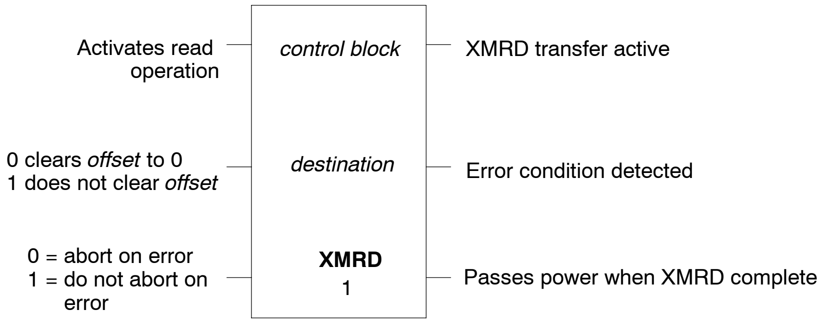
The top node may be a 3*x* input register or 4*x* holding register that specifies the first register in the block of registers to be written to extended memory.

The middle node is the first of six consecutive 4*x* registers to be used as the extended memory *control block* (as described in Section 16.3). If you are in multi-scan mode, these six registers should be unique to this function block.

The bottom node identifies the function as an extended memory write and always contains the constant value 1, which cannot be changed.

16.5 Extended Memory Read Function

The XMRD instruction is used to copy a table of 6*x* extended memory registers to a table of 4*x* holding registers in state RAM. XMRD is a three-node function block:



The top node is the first of six consecutive 4*x* registers to be used as the extended memory *control block* (as described in Section 16.3). If you are in multi-scan mode, these six registers should be unique to this function block.

The middle node is the first 4*x* holding register in a table of registers that receive the transferred data from the 6*x* extended memory storage registers.

The bottom node identifies the function as an extended memory read and always contains the constant value 1, which cannot be changed.

Chapter 17
Modbus Plus Master Function

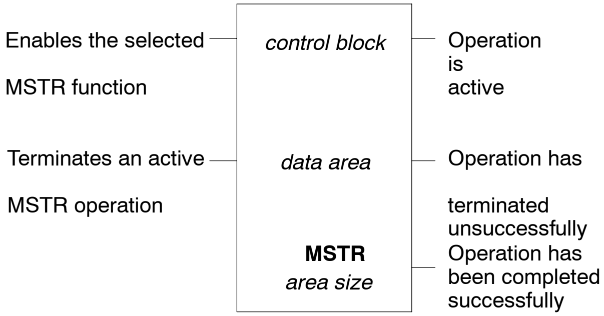
17.1 MSTR Block Overview

All 984 controllers that support a Modbus Plus communications capability have a special master (MSTR) instruction with which nodes on the network can initiate message transactions. The MSTR function allows you to initiate one of eight possible operations over the Modbus Plus network:

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

Up to four MSTR blocks may be simultaneously active in a ladder logic program. More than four MSTR blocks may be programmed to be enabled by the logic flow—as one active MSTR block releases the resources it has been using and becomes deactivated, the next MSTR function encountered in logic may be activated.

The MSTR instruction is a three-node function block:



The top node, which must be a 4*x* register, is the first of nine consecutive holding registers that form the MSTR *control block*:

4*x*	Identifies one of the eight MSTR operations
4*x* + 1	Displays error status
4*x* + 2	Displays length
4*x* + 3	Displays MSTR function-dependent information
4*x* + 4	The Routing 1 register, uses the bit value of the low byte to designate the address of the *destination* device; if you are using a controller with just one Mobbus Plus port, the value of the high byte should be set to 0:

high byte								*destination address*							
0	0	0	0	0	0	0	0	0	*x*	*x*	*x*	*x*	*x*	*x*	*x*
								binary value between 1 ... 64							

If you are using a controller with two Modbus Plus ports—e.g., using two S985 cards in a chassis mount controller—the value of the high byte for one port must be set to 0 and the high byte for the other port must be set to 1, leaving an offset of 256 between the *destination* node address and the register value:

high byte								*destination address*							
0	0	0	0	0	0	0	1	0	*x*	*x*	*x*	*x*	*x*	*x*	*x*
indicating a second MBP port								*binary value between 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

The middle node, which must also be a 4*x* register, designates the first register in the *data area*. For operations that provide the communication processor with data—such as a Write operation—the *data area* is the source of the data. For operations that acquire data from the communication processor—such as a Read operation—the *data area* is the destination of the data.

The bottom node indicates that this is an MSTR function and specifies the maximum number of registers in the *data area*; *area size* must be a constant value ranging from 1 ... 100.

17.2 MSTR Function Error Codes

If an error occurs during any one of the eight MSTR operations, a hexadecimal error code will be displayed in register 4*x* + 1 in the control block. The form of the code is Mmss, where

- M represents the major code
- m represents the minor code
- ss represents 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 equal to your own address
200A	Attempting to obtain more global data words than available
30ss*	Modbus slave exception response
4001	Inconsistent Modbus slave response
5001	Inconsistent network response
6mss**	Routing failure
07	Slave rejected long-duration program command

* The ss subfield in error code 30ss is:

ss Hex Value	Meaning
01	Slave device does not support the requested operation
02	Nonexistent slave device registers requested
03	Invalid data value requested
04	Unassigned
05	Slave has accepted long-duration program command
06	Function can't be performed now—a long-duration command in effect
08 ... 255	Unassigned

** The m subfield in error code 6mss is an index into the routing information indicating where an error has been detected—a value of 0 indicates the local node, a 2 the second device on the route, etc.

The ss subfield in error code 6mss is:

ss Hex Value	Meaning
01	No response received
02	Program access denied
03	Node offline and unable to communicate
04	Exception response received
05	Router node data paths busy
06	Slave device down
07	Bad destination address
08	Invalid node type in routing path
10	Slave has rejected the command
20	Initiated transaction forgotten by slave device
40	Unexpected master output path received
80	Unexpected response received

17.3 ***Read*** and ***Write*** MSTR Functions

An MSTR Write function transfers data from a *master source* device to a specified *slave destination* device on the network. An MSTR Read function transfers data from a specified *slave source* device to a *master destination* device on the network. Read and Write use one data master transaction path and may be completed over multiple scans.

17.3.1 Control Block Utilization

The contents of the nine registers in the top node of the MSTR block contain the following information when you implement a Read or Write function:

Control Block Register	**MSTR Function**	**Register Content**
4*x*	Operation type	1 = Write 2 = Read
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2	Length	Write = number of registers to be sent to slave Read = number of registers to be read from slave
4*x* + 3	Slave device data area	Specifies starting 4*x* register in the slave to be read from or written to (1 = 40001, 49 = 40049)
4*x* + 4, + 5, + 6, +7, +8	Routing 1, 2, 3, 4, 5	Designates the first through fifith routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

If you attempt to program the MSTR function to Read or Write its own station address, an error will be generated in the second register of the MSTR control block. It is possible to attempt a Read/Write operation to a nonexistent register in the slave device. The slave will detect this condition and report it—this may take several scans.

☞ **Note** You need to understand Modbus Plus routing path procedures before programming an MSTR block. A full discussion of routing path structures is given in ***Modbus Plus Network Planning and Installation Guide*** (GM-MBPL-001).

17.4 *Get Local Statistics* MSTR Function

The *Get local statistics* function obtains operational information related to the local node—where the MSTR function has been programmed. This operation takes one scan to complete and does not require a data master transaction path.

17.4.1 Control Block Utilization

The contents of the first four registers in the top node of the MSTR block are used when you implement a Get local statistics function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	3
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2	Length	Starting from *offset*, the number of words of statistics from the local processor's statistics table; the length must be > 0 ≤ the size of the data area
4*x* + 3	Offset	An offset value relative to the first available word in the local processor's statistics table—if the offset is specified as 1, the function obtains statistics starting with the second word in the table
4*x* + 4	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

See Section 17.10 for the listing of available network statistics.

17.5 *Clear Local Statistics* MSTR Function

The *Clear local statistics* function clears operational statistics relative to the local node—where the MSTR function has been programmed. This operation takes one scan to complete and does not require a data master transaction path.

17.5.1 Control Block Utilization

The contents of the first two registers in the top node of the MSTR block are used when you implement a Clear local statistics function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	4
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 4	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

See Section 17.10 for the listing of available network statistics.

17.6 *Write Global Data* MSTR Function

The *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 node gets the token. All nodes on the local network link can receive this data. This operation takes one scan to complete and does not require a data master transaction path.

17.6.1 Control Block Utilization

The contents of the first three registers in the top node of the MSTR block are used when you implement a Write global data function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	5
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2	Length	Specifies the number of registers from the data area to be sent to the comm processor; the value of the length must be ≤ 32 and must not exceed the size of the data area
4*x* + 4	Routing 1	If this is the second of two local nodes, set the high byte to a value of 1

17.7 *Read Global Data* MSTR Function

The *Read global data* function gets data from the comm processor in any node on the local network link that is providing global data. This operation may require multiple scans to complete if no global data are currently available from the requested node; if global data are currently available, the operation completes in a single scan. No master transaction path is required.

17.7.1 Control Block Utilization

The contents of the first five registers in the top node of the MSTR block are used when you implement a Read global data function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	6
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2	Length	Specifies the number of words of global data to be requested from the comm processor designated by the routing 1 parameter; the value of the length must be > 0 $\leq$ 32 and must not exceed the size of the data area
4*x* + 3	Available words	Contains the number of words available from the requested node; the value is automatically updated by internal software
4*x* + 4	Routing 1	The low byte specifies the address of the node whose global data are to be returned (a value between 1 ... 64); if this is the second of two local nodes, set the high byte to a value of 1

17.8 *Get Remote Statistics* MSTR Function

The *Get remote statistics* function obtains operational information relative to remote nodes on the network. This operation may require multiple scans to complete and does not require a master data transaction path.

17.8.1 Control Block Utilization

The contents of the nine registers in the top node of the MSTR block contain the following information when you implement a Get remote statistics function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	7
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2	Length	Starting from an *offset,* the number of words of statistics to be obtained from a remote node; the value of the length must be > 0 ≤ total number of statistics available (54) and must not exceed the size of the data area
4*x* + 3	Offset	Specifies an offset 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	Designates the first through fififth routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

The remote comm processor always returns its complete statistics table when a request is made, even if the request is for less than the full table. The MSTR function then copies only the amount of words you have requested to the designated 4*x* registers.

☞ **Note** You need to understand Modbus Plus routing path procedures before programming an MSTR block. A full discussion of routing path structures is given in ***Modbus Plus Network Planning and Installation Guide*** (GM-MBPL-001).

17.9 *Clear Remote Statistics* MSTR Function

The *Clear remote statistics* function clears operational statistics related to a remote network node from the data area in the local node. This operation may require multiple scans to complete and uses a single data master transaction path.

17.9.1 Control Block Utilization

The contents of seven registers in the top node of the MSTR block contain the following information when you implement a Clear remote statistics function:

Control Block Register	MSTR Function	Register Content
4*x*	Operation type	8
4*x* + 1	Error status	Displays a hex value indicating an MSTR error, when relevant (see 17.2)
4*x* + 2 and 4*x* + 3	Not used	
4*x* + 4, + 5, + 6, +7, +8	Routing 1, 2, 3, 4, 5	Designates the first through fifith routing path addresses, respectively; the last nonzero byte in the routing path is the destination device

☞ **Note** You need to understand Modbus Plus routing path procedures before programming an MSTR block. A full discussion of routing path structures is given in ***Modbus Plus Network Planning and Installation Guide*** (GM-MBPL-001).

See Section 17.10 for the listing of available network statistics.

17.10 Network Statistics

The following table presents statistics available on the Modbus Plus network. You may acquire this information by using the appropriate MSTR logic function or by using Modbus function code 8.

☞ **Note** When you issue the *Clear local* or *Clear remote statistics* functions, 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		Communications processor version. First release is version 1.00 and displays 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 status of this unit relative to the network:
	0	Monitor link operation
	32	Normal link operation
	64	Never getting token
	96	Sole station
	128	Duplicate station

continued on next page

Modbus Plus Network Statistics (continued)

Word	Byte	Meaning
05		Token pass counter; increments each time this station gets the token
06		Token rotation time in ms
07	LO	Data master failed during token ownership bit map
	HI	Program master failed during token ownership bit map
08	LO	Data master token owner work bit map
	HI	Program master token owner work bit map
09	LO	Data slave token owner work bit map
	HI	Program slave token owner work bit map
10	LO	Data master/get master response transfer request bit map
	HI	Data slave/get slave command transfer request bit map
11	LO	Program master/get master rsp transfer request bit map
	HI	Program slave/get slave command transfer request bit map
12	LO	Program master connect status bit map
	HI	Program slave automatic logout request bit map
13	LO	Pretransmit deferral error counter
	HI	Receive buffer DMA overrun error counter
14	LO	Repeated command received counter
	HI	No Try counter (nonexistent station)
15	LO	Receiver collision-abort error counter
	HI	Receiver alignment error counter
16	LO	Receiver CRC error counter
	HI	Bad packet-length error counter
17	LO	Bad link-address error counter
	HI	Transmit buffer DMA-underrun error counter
18	LO	Bad internal packet length error counter
	HI	Bad mac function code error counter
19	LO	Communication retry counter
	HI	Communication failed error counter
20	LO	Good receive packet success counter
	HI	No response received error counter
21	LO	Exception response received error counter
	HI	Unexpected path error counter

Modbus Plus Network Statistics (continued)

Word	Byte	Meaning
22	LO	Unexpected response error counter
	HI	Forgotten transaction error counter
23	LO	Active station table bit map, nodes 1 ... 8
	HI	Active station table bit map, nodes 9 ...16
24	LO	Active station table bit map, nodes 17 ... 24
	HI	Active station table bit map, nodes 25 ... 32
25	LO	Active station table bit map, nodes 33 ... 40
	HI	Active station table bit map, nodes 41 ... 48
26	LO	Active station table bit map, nodes 49 ... 56
	HI	Active station table bit map, nodes 57 ... 64
27	LO	Token station table bit map, nodes 1 ... 8
	HI	Token station table bit map, nodes 9 ... 16
28	LO	Token station table bit map, nodes 17 ... 24
	HI	Token station table bit map, nodes 25 ... 32
29	LO	Token station table bit map, nodes 33 ... 40
	HI	Token station table bit map, nodes 41 ... 48
30	LO	Token station table bit map, nodes 49 ... 56
	HI	Token station table bit map, nodes 57 ... 64
31	LO	Global data present table bit map, nodes 1 ... 8
	HI	Global data present table bit map, nodes 9 ... 16
32	LO	Global data present table bit map, nodes 17 ... 24
	HI	Global data present table bit map, nodes 25 ... 32
33	LO	Global data present table bit map, nodes 33 ... 40
	HI	Global data present table bit map, nodes 41 ... 48
34	LO	Global data present table bit ... map, nodes 49 ... 56
	HI	Global data present table bit map, nodes 57 ... 64
35	LO	Receive buffer in use bit map, buffer 1 ... 8
	HI	Receive buffer in use bit map, buffer 9 ... 16
36	LO	Receive buffer in use bit map, buffer 17 ... 24
	HI	Receive buffer in use bit map, buffer 25 ... 32
37	LO	Receive buffer in use bit map, buffer 33 ... 40
	HI	Station management command processed initiation counter

continued on next page

Modbus Plus Network Statistics (concluded)

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

Chapter 18
CKSM

984 slot mount and micro controllers that *do not* support Modbus Plus come with a standard checksum (CKSM) instruction. The CKSM instruction has the same opcode as the MSTR function and is not provided in executive firmware with the 984 controllers that support Modbus Plus.

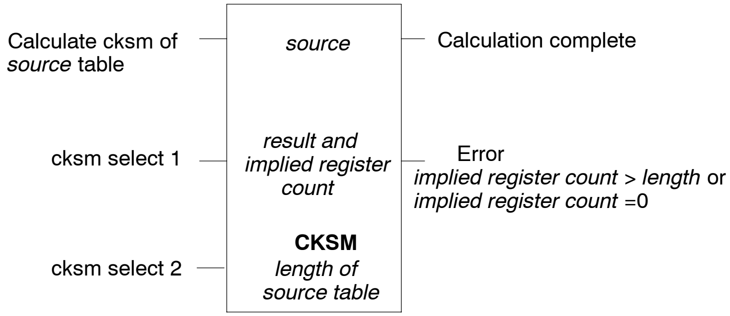
18.1 CKSM

CKSM allows you to program four types checksum calculations in ladder logic:

- Straight check
- Binary addition check
- Cyclical redundancy check (CRC-16)
- Longitudinal redundancy check (LRC)

All checksum algorithms handle both 8 bit and 16 bit data; if 8 bits are used, the high order byte in the register must be 0. In a straight checksum calculation, all bytes (high and low) are summed and the least significant eight bits are returned. A binary checksum calculation is a 16 bit sum of all registers. An LRC is a straight checksum that is then two's complemented. A CRC-16 calculation is a 16 bit cyclical checksum performed on the least significant bytes of the *source* registers.

The CKSM instruction is a three-node function block:



The top node contains the first 4*x* register in the *source* table. The checksum calculation is performed on the registers in this table.

The middle node contains two 4*x* registers—4*x* stores the result of the checksum calculation, and 4*x* + 1 specifies the number of registers selected from the *source* table used as input to the calculation.

The value in 4*x* + 1 must be ≤ *length of source table*

The bottom node identifies the block as CKSM and contains an integer value in the range 1 ... 255, specifying the number of 4*x* registers in the *source* table.

The three inputs to the block are used to indicate the type of checksum calculation to be performed:

CKSM Calculation	**Input Top**	**Mid**	**Bottom**
Straight Check	ON	OFF	ON
Binary Addition Check	ON	ON	ON
CRC-16	ON	ON	OFF
LRC	ON	OFF	OFF

Chapter 19
Ladder Logic Subroutines

19.1 Using Ladder Logic Subroutines

Several 984 instruction sets provide three standard function blocks in the EPROM firmware that allow you to set up ladder logic-based subroutines. The **JSR** function jumps from the regular (scheduled) logic to a subroutine; the **LAB** function labels the starting point of the subroutine; and the **RET** function returns you from the subroutine network to the regular (scheduled) user logic program.

19.1.1 The Value of Subroutines

Ladder logic subroutines allow you to save memory space in the user logic table in cases where you need to implement the same logic functions multiple times in a single scan. You need only create the logic once, store it in the logic segment reserved for subroutines, and call it from user logic with the JSR block as often as you need it within a scan.

Subroutines can also be helpful in reducing total scan time. Portions of logic that require only infrequent solution in logic scans can be placed in the subroutine segment and called from user logic only on those scans where it is needed.

19.1.2 Where to Store Subroutines in Ladder Logic

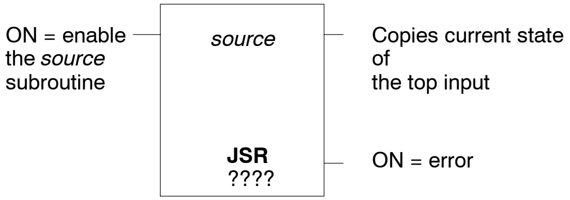
All ladder logic subroutines must be built in the *last* segment of user logic. This segment must be removed from the segment scheduler—it is not part of the regular order-of-solve table.

Note This means that you must specify at least one more segment than is required for regular user logic in the configuration table.

Controllers that support subroutines provide as many as 255 address locations for subroutine ladder logic. Each subroutine must start at the beginning of a network in the last logic segment. There is no set limit on the number of networks in the segment.

19.2 JSR

The JSR instruction causes the logic scan to jump to a specified subroutine in the last (unscheduled) segment of user logic. JSR is a two-node function block:



The top node contains a *source* that indicates the subroutine to which the logic scan is to jump. It may be specified as:

- A constant value useful in the range 1 ... 255
- A single holding register (4*x)* containing a value between 1 ... 255

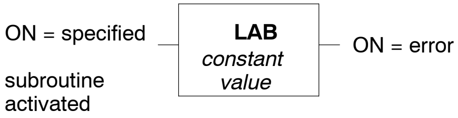
The bottom node indicates that this is a JSR function and contains a string of four question marks—you must insert the constant value 1 in this node.

Note You can use a JSR block anywhere in user logic, even within a subroutine. The process of calling one subroutine from another subroutine is called *nesting*. The system allows you to nest up to 100 subroutines—however, we recommend that you use no more than three nesting levels.

You may also perform a recursive form of nesting called *looping*, wherein the subroutine recalls itself.

19.3 LAB

The LAB instruction is used to label the starting point of a subroutine in the last (unscheduled) segment of user logic. This instruction must be programmed in row 1, column 1 of a network in the last (unscheduled) segment of user logic. LAB is a one-node function block:



The node indicates that this is a LAB function and contains a unique *constant value identifying* the subroutine you are about to run; it may range from 1 ... 255. If more than one subroutine network has the same LAB value, the network with the lowest number is used as the starting point for the subroutine.

Note The LAB block also functions as a default return from the subroutine in the preceding networks. If you have been executing a series of subroutine networks and you encounter a network that begins with a LAB block, the system assumes that the desired subroutine is finished, and it returns the logic scan to the node immediately following the most recently executed JSR block.

19.4 RET

The RET instruction may be used to conditionally return the logic scan to the node immediately following the most recently executed JSR block. This node can be implemented only from within a subroutine network—in the last (unscheduled) segment of user logic. RET is a one-node function block:

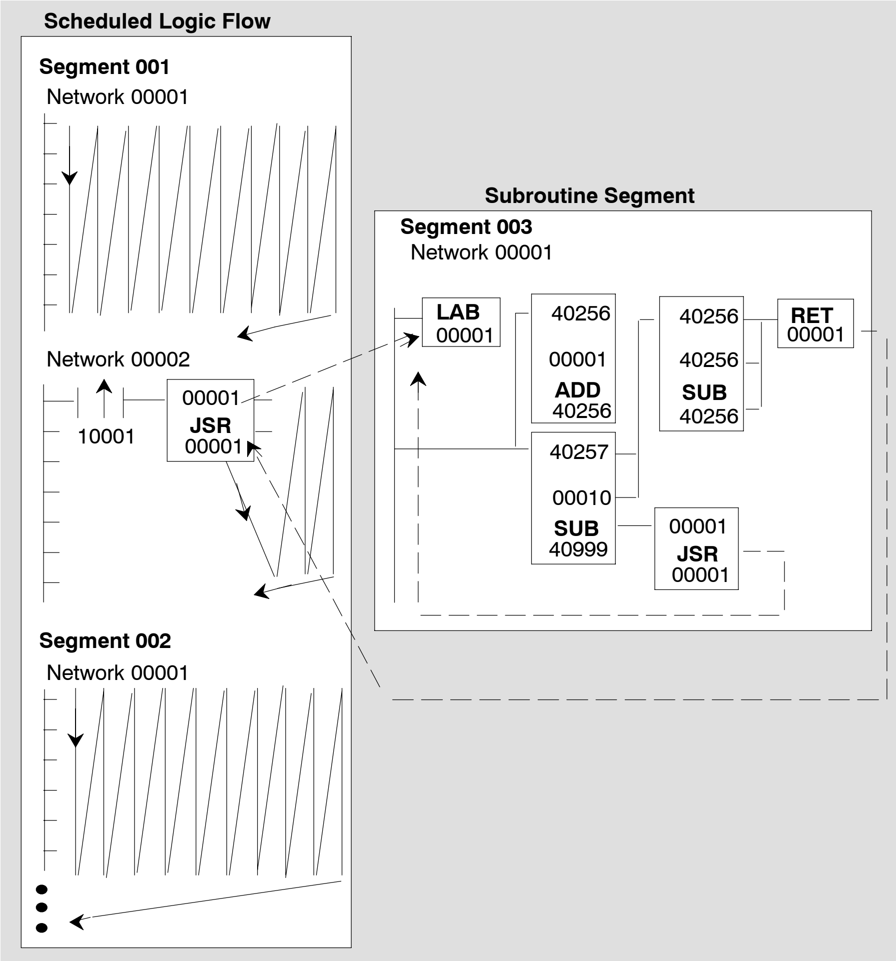
The bottom node indicates that this is a RET function and contains the constant value 00001.

When the ENABLE input is energized, the RET block returns the logic scan to the node immediately following the most recently executed JSR block.

If a subroutine does not contain a RET block, either a LAB block or the end-of-logic (whichever comes first) serves as the default return from the subroutine.

19.5 A Subroutine Example

The example below shows a series of three user logic networks, the last of which is used for an up-counting subroutine. Segment 3 has been removed from the order-of-solve table in the segment scheduler:



When input 10001 to the JSR block in network 2 of segment 1 transitions from OFF to ON, the logic scan jumps to subroutine #1 in network 1 of segment 3.

The subroutine will internally loop on itself ten times, counted by the ADD block. The first nine loops end with the JSR block in the subroutine (network 1 of segment 3) sending the scan back to the LAB block. Upon completion of the tenth loop, the RET block sends the logic scan back to the scheduled logic at the JSR node in network 2 of segment 1.

19.6 Some Cautionary Notes About Subroutines

You should always keep your subroutine logic as straightforward as possible for debugging purposes. The power flow displayed on your programming panel is invalid in the subroutine networks and is therefore more difficult to troubleshoot.

☞ **Note** We recommend that you debug your ladder logic programs before making them subroutines.

For transitionals to work properly within a subroutine, the subroutine must be executed at the appropriate time to see the state change. To use a negative transitional within the subroutine, the subroutine must be called once when the contact is ON, then called again on the scan when the contact is turned OFF. To use a positive transitional within a subroutine, the subroutine must be called while the contact is OFF, then called again on the scan when the contact is turned ON.

Counters also work on a state change basis—when the top input transitions from OFF to ON. Timers do not function properly from within a subroutine unless that subroutine is executed on every scan.

☞ **Note** Multiple scan functions do not function from within a subroutine.

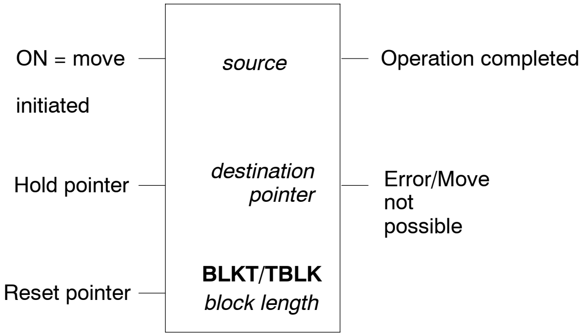
⚠ **Caution We strongly recommend that you *do not* control real-world outputs from within a ladder logic subroutine. Control of such coils would be possible only when the subroutine was executed.**

Chapter 20
984 Enhanced Instructions

- ❐ Moving Blocks to Tables and Tables to Blocks
- ❐ Capabilities of the EMTH Block
- ❐ Double Precision Math Functions
- ❐ Integer Math Functions
- ❐ Floating Point Arithmetic Functions
- ❐ A Closed Loop Control System
- ❐ The PID2 Block
- ❐ Top Node Values
- ❐ Middle Node Values
- ❐ PID2 Error Codes
- ❐ Process Square Root

20.1 Moving Blocks to Tables and Tables to Blocks

The block-to-table (BLKT) and table-to-block (TBLK) instructions can be thought of as functions that combine the R→T/T→R instructions with the BLKM instruction. BLKT moves large quantities of holding registers from a fixed-source block to a destination block within a table; TBLK moves a large number of consecutive registers from a table to a fixed-destination block. A BLKT or a TBLK function is accomplished in one scan. They are both three-node function blocks:



The top node—*source*—must be the first 4*x* holding register in the block to be moved.

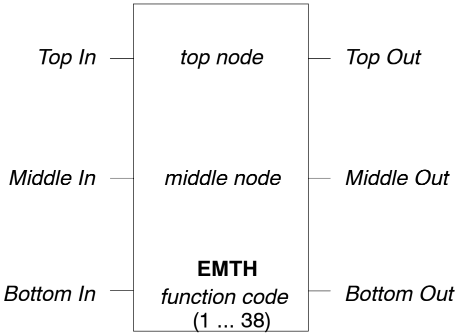
The middle node is the *destination pointer*; it is a movable 4*x* pointer that indicates the first register in the destination block (or table). The destination block itself begins with register 4*x* + 1 and runs to the end of the *block length* specified in the bottom node.

The bottom node indicates that this is a BLKT or TBLK function and specifies a number of 4*x* registers in a destination block within the table. The range is from 1 ... 100; the overall size of the destination table is a function of the number of 4*x* registers currently available.

Warning BLKT is a powerful function. If your logic does not confine the pointer to a desired range, all the registers in your 984 controller may be corrupted by the data in the *source* node.

20.2 Capabilities of the EMTH Block

EMTH provides you with double-precision math capabilities, additional integer math capabilities such as square root and logarithm calculations, and a set of floating point (FP) arithmetic functions. In all, the block allows you to select 38 extended math functions using a code number in the bottom node. EMTH is a three-node function block:



The top node requires two consecutive registers, usually 4*x* holding registers but, in the integer math cases, either 4*x* or 3*x* registers.

The middle node requires either two, four, or six consecutive registers, depending on the function you are implementing. Use 4*x* holding registers.

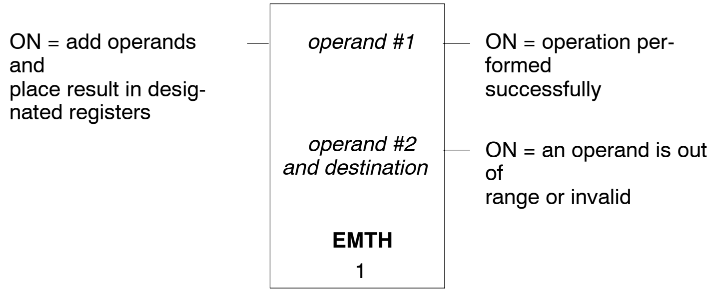
The bottom node identifies the block as the EMTH function and provides a functional selection mechanism for the block. Enter a constant value in the range 1 ... 38 to indicate the extended math function you want to employ.

Inputs to and outputs from the EMTH block may be ACTIVE or INACTIVE, depending on the function called in the bottom node.

EMTH Functions	Code	Active Inputs	Active Outputs
Double Precision Math			
Addition	01	Top only	Top, Middle
Subtraction	02	Top only	Top, Middle, Bottom
Multiplication	03	Top only	Top, Middle
Division	04	Top, Middle	Top, Middle, Bottom
Integer Math			
Square Root	05	Top only	Top, Middle
Process Square Root	06	Top only	Top, Middle
Logarithm	07	Top only	Top, Middle
Antilogarithm	08	Top only	Top, Middle
Floating Point Arithmetic			
Integer-to-FP Conversion	09	Top only	Top only
Integer + FP	10	Top only	Top only
Integer – FP	11	Top only	Top only
Integer x FP	12	Top only	Top only
Integer ÷ FP	13	Top only	Top only
FP – Integer	14	Top only	Top only
FP ÷ Integer	15	Top only	Top only
Integer-FP Comparison	16	Top only	Top only
FP-to-Integer Conversion	17	Top only	Top, Bottom
Addition	18	Top only	Top only
Subtraction	19	Top only	Top only
Multiplication	20	Top only	Top only
Division	21	Top only	Top only
Comparison	22	Top only	Top, Middle, Bottom
Square Root	23	Top only	Top only
Change Sign	24	Top only	Top only
Load Value of π	25	Top only	Top only
Sine in Radians	26	Top only	Top only
Cosine in Radians	27	Top only	Top only
Tangent in Radians	28	Top only	Top only
Arcsine in Radians	29	Top only	Top only
Arccosine in Radians	30	Top only	Top only
Arctangent in Radians	31	Top only	Top only
Radians to Degrees	32	Top only	Top only
Degrees to Radians	33	Top only	Top only
FP to an Integer Power	34	Top only	Top only
Exponential Function	35	Top only	Top only
Natural Log	36	Top only	Top only
Common Log	37	Top only	Top only
Report Errors	38	Top only	Top, Middle

20.3 Double Precision Math Functions

Double Precision Addition

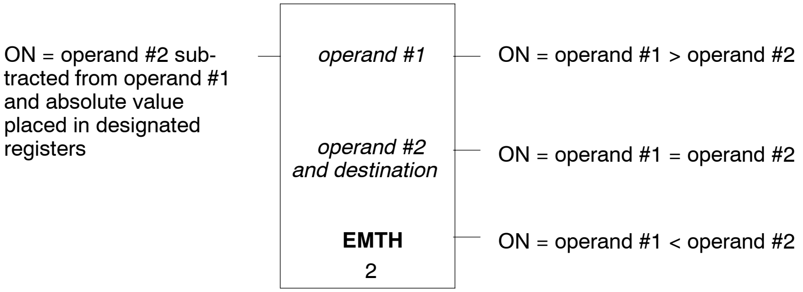


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2 indicates whether an overflow condition exists (1 = overflow)
- 4*x* + 3 and 4*x* + 4 hold the double precision addition result
- 4*x* + 5 is not used in this calculation but must exist in state RAM

Double Precision Subtraction

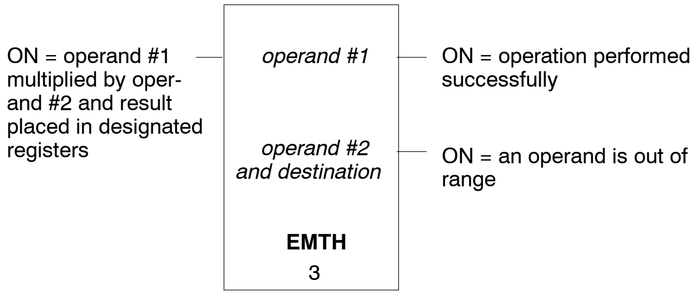


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2 and 4*x* + 3 hold the double precision subtraction result
- 4*x* + 4 indicates whether or not the operands are in the valid range (1 = out of range and 0 = in range)
- 4*x* + 5 is not used in this calculation but must exist in state RAM

Double Precision Multiplication

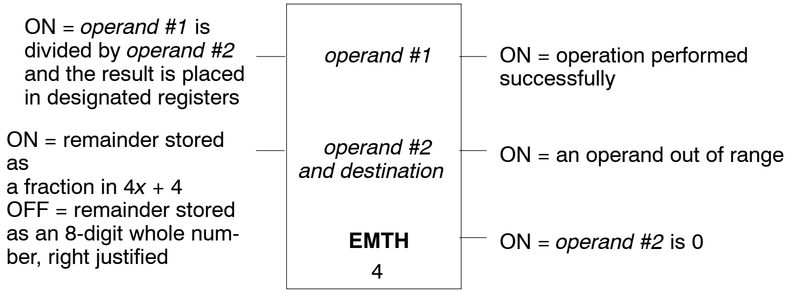


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2, 4*x* + 3, 4*x* + 4, and 4*x* + 5 hold the double precision multiplication result

Double Precision Division



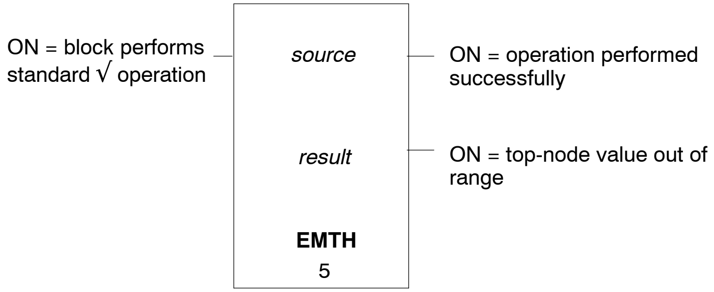
The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999 (Since division by 0 is illegal, a 0 value causes an error—an error trapping routine sets the remaining middle-node registers to 0000 and turns the bottom output ON.)
- 4*x* + 2 and 4*x* + 3 hold an eight-digit result, the quotient
- 4*x* + 4 and 4*x* + 5 hold the remainder—if the remainder is expressed in whole numbers, it is eight digits long and both registers are used; if the remainder is expressed as a decimal, it is four digits long and only register 4*x* + 4 is used

20.4 Integer Math Functions

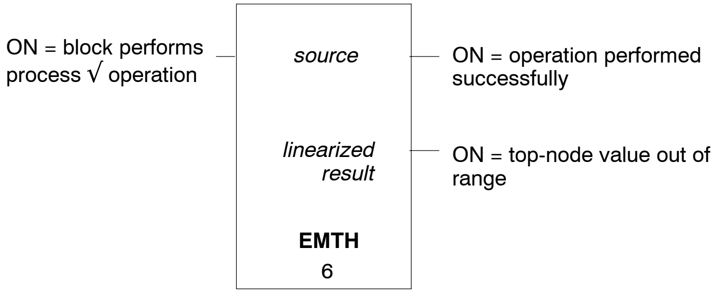
Square Root



The top node comprises either two consecutive 4*x* holding registers or one 3*x* input register. If the *source* value is five to eight digits long in the range 10,000 ... 99,999,99, it is stored in the two consecutive 4*x* registers. If the *source* is less than five digits long, in the range 0 ... 9,999, it is stored in register 4*x* + 1. If you specify a 3*x* register in the top node, the square root calculation is done on only register 3*x*; a second register is implied but not used.

The middle node comprises two consecutive 4*x* registers, where the *result* of the standard square root operation is stored. Data are stored in a fixed-decimal format: 1234.5600. where register 4*x* stores the most significant data, to the left of the first decimal point, and register 4*x* + 1 stores the four-digit value to the right of the first decimal point. Numbers after the second decimal point are truncated; no roundoff calculations are performed.

Process Square Root



The process square root function implements the standard square root function and tailors it for closed loop analog control applications. It takes the result of the standard square root operation, multiplies it by 63.9922—the square root of 4095—and stores that *linearized result* in the middle-node registers. In order to generate values that have meaning, the value entered in the top-node 4*x* or 3*x* register must not exceed 4095. Process square root linearizes signals from differential pressure flow transmitters so that they may be used as inputs in PID2 operations (see Section 20.8).

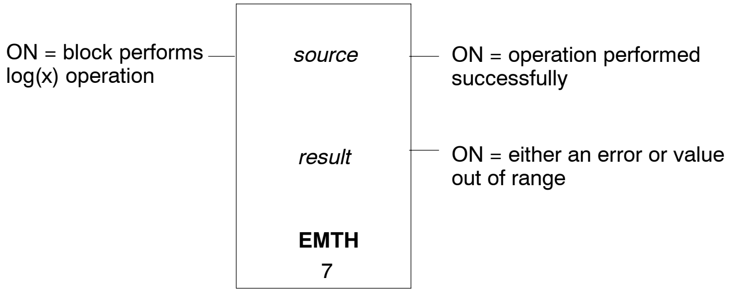
For example, if a value of 2000 is in a 30300 top node, then:

$$\sqrt{2000} = 0044.72$$

which is then multiplied by 63.9922, yielding a *result* of 2861.63. This *result* is placed in registers 40030 and 40031 in the middle node:

40030 = 2861
40031 = 6300

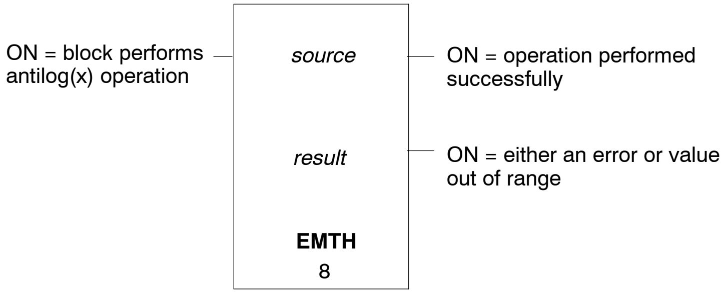
Logarithm (base 10)



The top node comprises either two consecutive 4*x* holding registers or one 3*x* input register. If the *source* to be logged is five to eight digits long in the range 10,000 ... 99,999,99, it is stored in the two consecutive 4*x* registers. If the *source* is less than five digits long, in the range 0 ... 9,999, it is stored in register 4*x* + 1. If you specify a 3*x* register in the top node, the log calculation is done on only register 3*x*; a second register is implied but not used.

The middle node contains a single 4*x* holding register where the *result* is stored. The *result* is expressed in a fixed decimal format 1.234, and is truncated after the third decimal position. The largest number that can be calculated is 7.999, which is stored in the register as value 7999.

Antilogarithm (base 10)



The top node is a single 4*x* holding register or 3*x* input register. The *source* value stored here is in the fixed decimal format 1.234 and must be in the range 0 ... 7.999; the largest antilog value that can be calculated is 99770006.

The *result* is stored in two consecutive 4*x* holding registers in the middle node, in the fixed decimal format 12345678, where the most significant bits are in 4*x* and the least significant bits are in 4*x* + 1.

20.5 Floating Point Arithmetic Functions

To make use of the FP capability, the standard four-digit integer values used in standard 984 instructions must be converted to the IEEE floating point format. All calculations are then performed in FP format, and the results must be converted back to integer format.

20.5.1 The IEEE Floating Point Standard

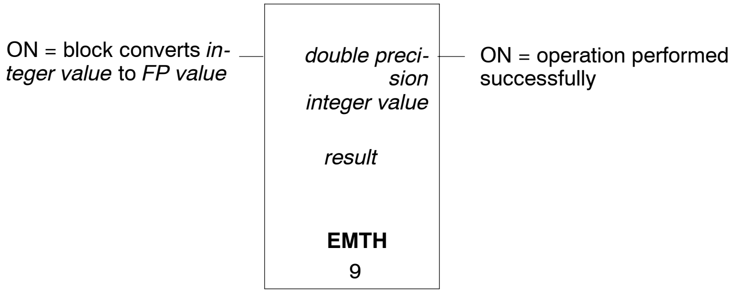
EMTH floating point functions require values in 32-bit IEEE floating point format. Each value has two registers assigned to it—the eight most significant bits representing the exponent and the other 23 bits (plus one assumed bit) representing the mantissa and the sign of the value. It is virtually impossible to recognize an FP representation on the programming panel. Therefore, all numbers should be converted back to integer format before you attempt to read them.

20.5.2 Dealing with Negative Floating Point Numbers

Standard 984 integer math does not handle negative numbers explicitly. The only way to identify negative values is by noting that the SUB function block has turned the bottom output ON.

If such a negative number is being converted to floating point, perform the Integer-to-FP conversion (EMTH function #9), then use the Change Sign function (EMTH function #24) to make it negative prior to any other FP calculations.

Integer-to-FP Conversion



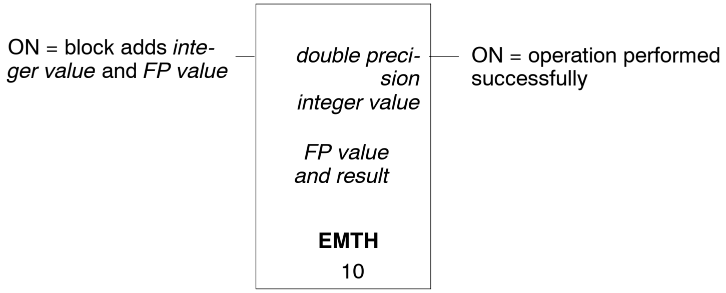
The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* to be converted to 32-bit FP format.

☞ **Note** If an invalid integer value (*value* > 9999) is placed in either of the two top-node registers, the FP conversion will be performed but an error will be reported and logged in EMTH function #38. The *result* of the conversion may not be correct.

The middle node contains four consecutive 4*x* registers—4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 are used to store the *result* of the FP conversion.

☞ **Note** If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

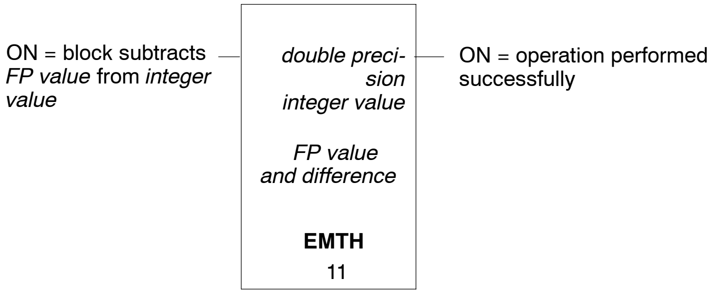
Integer + FP



The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* to be added to a FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the FP number to be added in the operation, and 4*x* + 2 and 4*x* + 3 contain the FP sum of the operation.

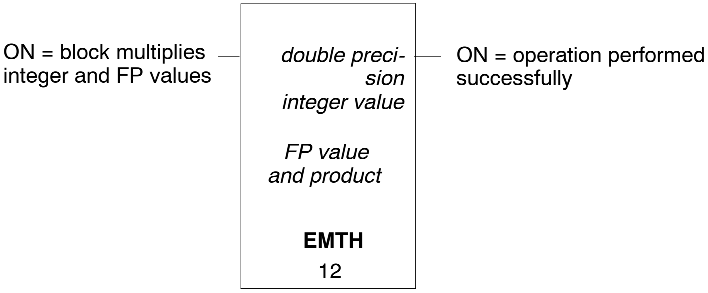
Integer - FP



The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* from which an FP number is to be subtracted.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the FP number that is subtracted from the integer value in the top node, and 4*x* + 2 and 4*x* + 3 contain the FP difference of the operation.

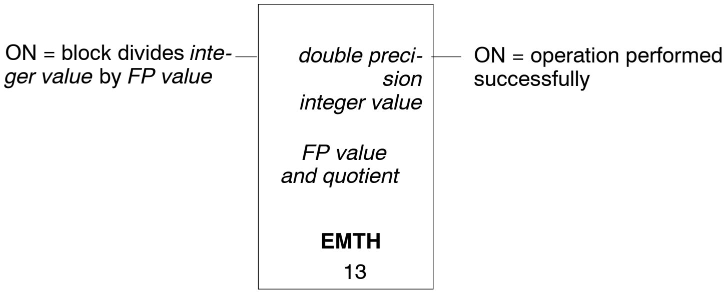
Integer x FP



The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* to be multiplied by an FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the FP number that multiplies the integer value in the top node, and 4*x* + 2 and 4*x* + 3 contain the FP product of the operation.

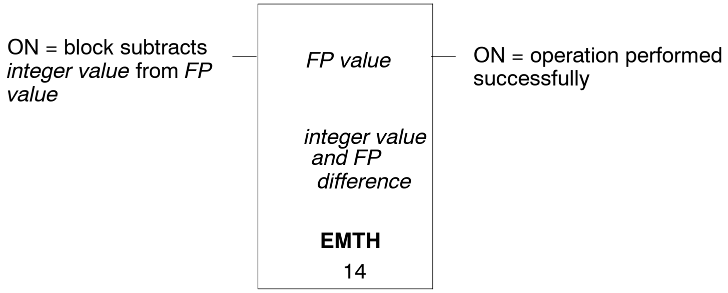
Integer : FP



The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* to be divided by an FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the FP number that divides the integer value in the top node, and 4*x* + 2 and 4*x* + 3 contain the FP quotient of the operation.

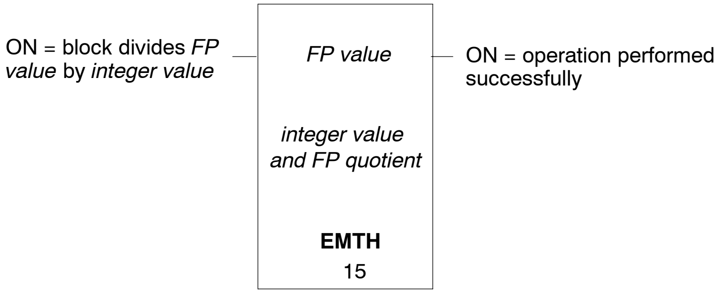
FP - Integer



The top node comprises two consecutive 4*x* registers that contain an FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the integer value to be subtracted from the FP value in the top node, and 4*x* + 2 and 4*x* + 3 contain the FP difference of the operation.

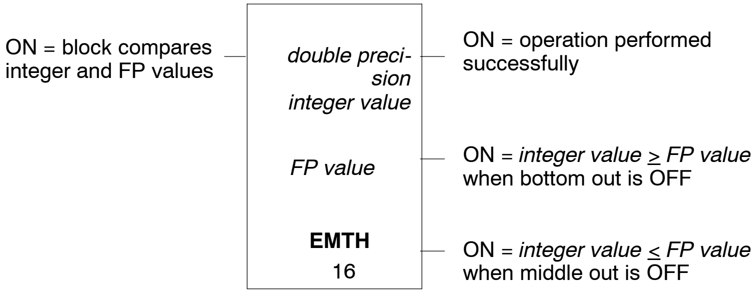
FP ÷ Integer



The top node comprises two consecutive 4*x* registers that contain an FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain the integer value that divides the FP value in the top node, and 4*x* + 2 and 4*x* + 3 contain the FP quotient of the operation.

Integer-FP Comparison



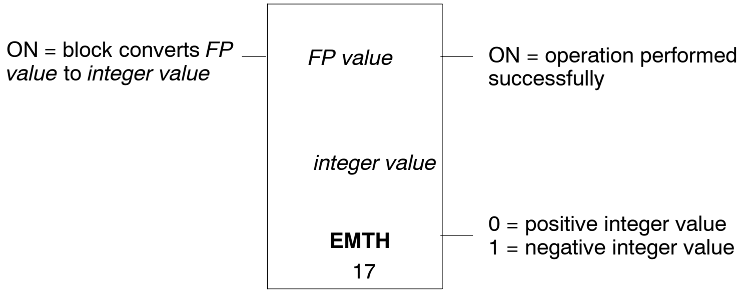
Middle Output	Bottom Output	Value Relationship
ON	OFF	I > FP
OFF	ON	I < FP
ON	ON	I = FP

The top node comprises two consecutive 4*x* registers that contain a *double precision integer value* to be compared with an FP number.

The middle node comprises four consecutive 4*x* registers—4*x* and 4*x* + 1 contain an FP value to be compared with the integer value in the top node, and the other two nodes are not used.

The result of the comparison is displayed by the state of the middle and bottom outputs.

FP-to-Integer Conversion



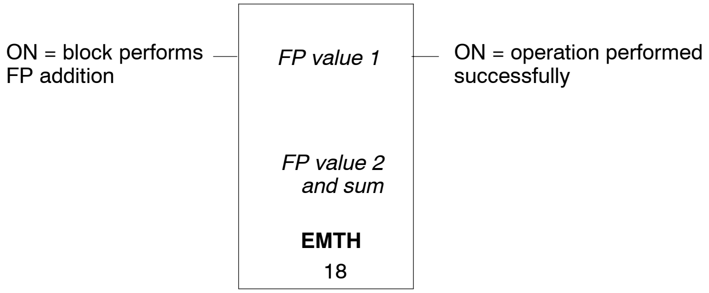
The top node comprises two consecutive 4*x* registers that contain an *FP value* in 32-bit FP format.

The middle node contains four consecutive 4*x* registers—4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the integer result of the conversion. This value should be the largest *integer value* possible that is ≤ the *FP value*—for example, the *FP value* 3.5 is converted to the *integer value* 3, while the *FP value* -3.5 is converted to the integer value -4.

Note If the resultant integer is too large for 984 double precision integer format (> 99,999,999), the conversion still occurs but an error is logged in EMTH function #38.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

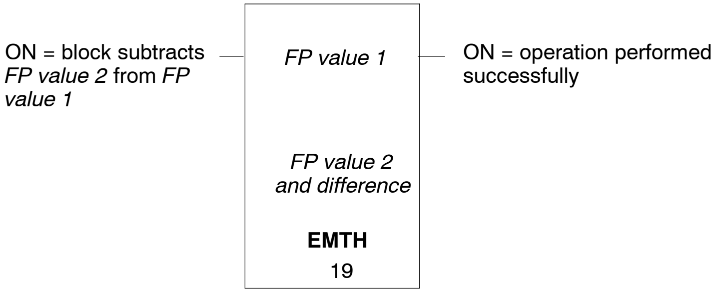
FP Addition



The top node comprises two consecutive 4*x* registers that contain one *FP value.*

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 contain a second *FP value*; 4*x* + 2 and 4*x* + 3 contain the *FP sum* of the addition.

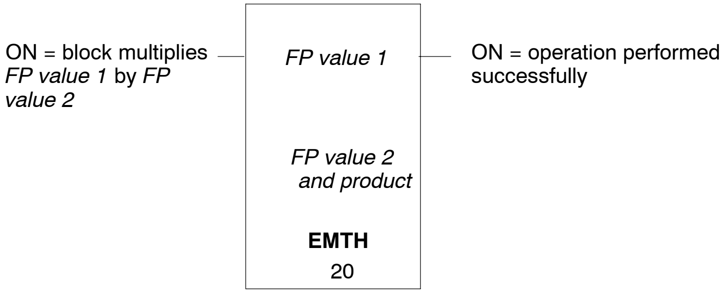
FP Subtraction



The top node comprises two consecutive 4*x* registers that contain one *FP value.*

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 contain a second *FP value*, which will be subtracted from the top-node value; 4*x* + 2 and 4*x* + 3 contain the *FP difference* of the subtraction.

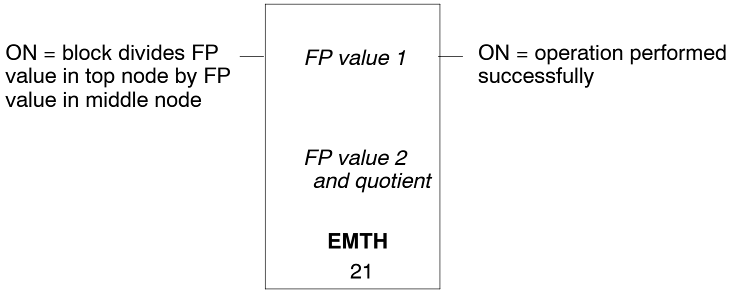
FP Multiplication



The top node comprises two consecutive 4*x* registers that contain one *FP value*, which will be multiplied by the middle-node value.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 contain a second *FP value*; 4*x* + 2 and 4*x* + 3 contain the FP product.

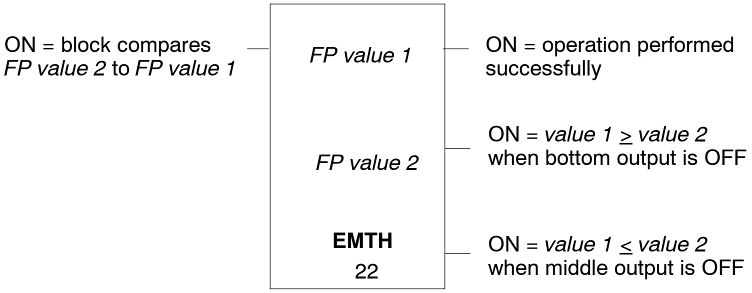
FP Division



The top node comprises two consecutive 4*x* registers that contain one *FP value*, which will be divided by the middle-node value.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 contain the second *FP value*; 4*x* + 2 and 4*x* + 3 contain the FP quotient.

FP Comparison

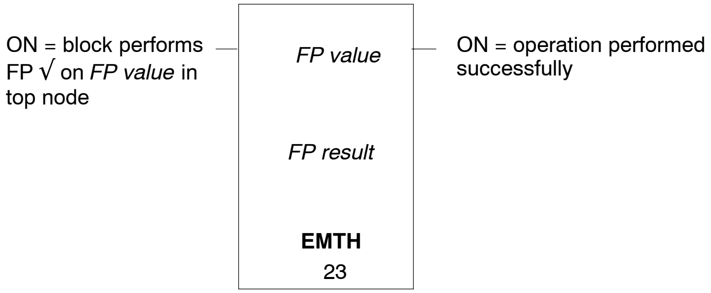


Middle Output	Bottom Output	Value Relationship
ON	OFF	#1 > #2
OFF	ON	#1 < #2
ON	ON	#1 = #2

The top node comprises two consecutive 4*x* registers that contain one *FP value.*

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 contain the second *FP value*, which will be compared to the top-node *value*; 4*x* + 2 and 4*x* + 3 are not used.

FP Square Root

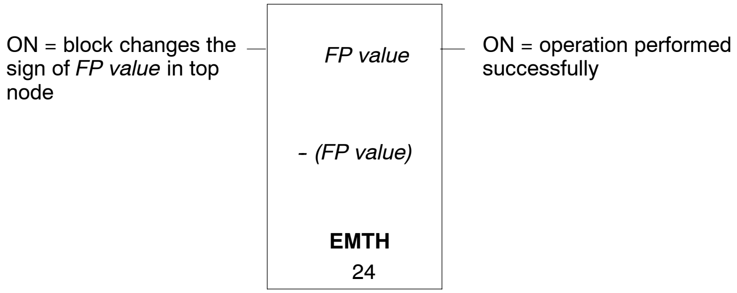


The top node comprises two consecutive 4*x* registers that contain an *FP value.*

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the result of the FP square root operation.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

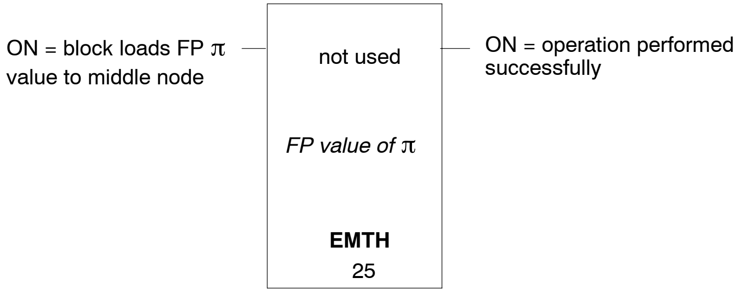
FP Change Sign



The top node comprises two consecutive 4*x* registers that contain an *FP value.*

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the negative of the top node *FP value*.

Load FP Value of π

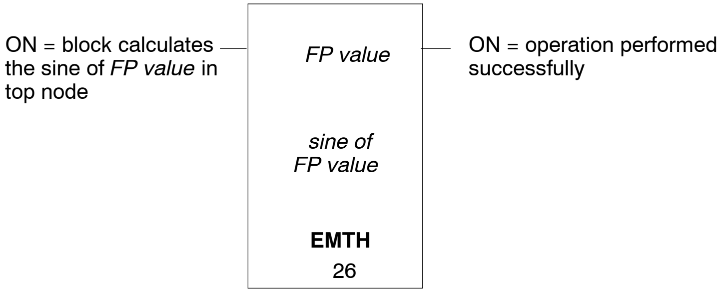


The top node contains two consecutive 4*x* registers that are not used.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the *FP value of* π.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since these registers must be assigned but are not used.

FP Sine of an Angle in Radians



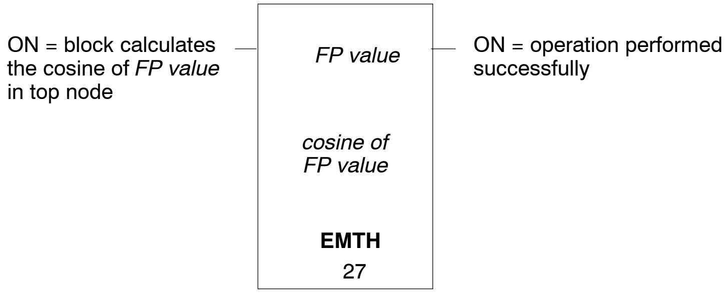
The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the value of an angle in radians. The magnitude of this value must be < 65536.0; if not:

- The sine is not computed
- An invalid result is returned
- An error is flagged in EMTH function #38

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the sine of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Cosine of an Angle in Radians



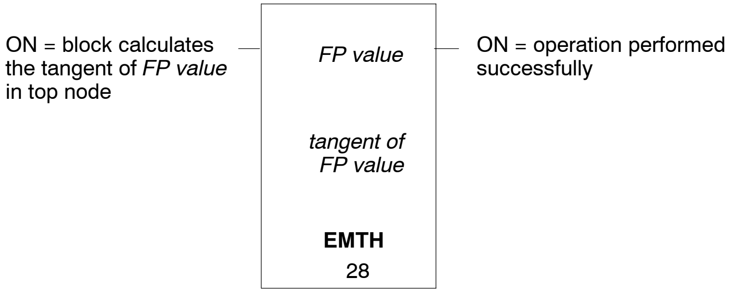
The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the value of an angle in radians. The magnitude of this value must be < 65536.0; if not:

- The cosine is not computed
- An invalid result is returned
- An error is flagged in EMTH function #38

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the cosine of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Tangent of an Angle in Radians



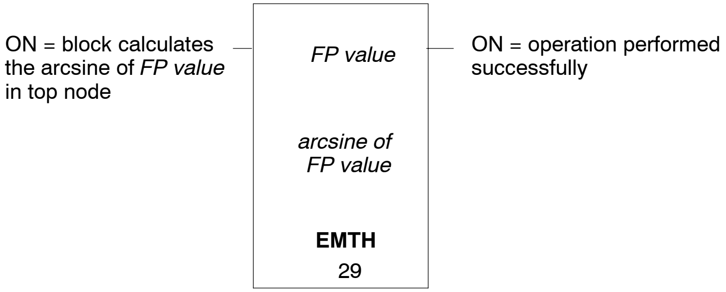
The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the value of an angle in radians. The magnitude of this value must be < 65536.0; if not:

- The tangent is not computed
- An invalid result is returned
- An error is flagged in EMTH function #38

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the tangent of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Arcsine of an Angle in Radians



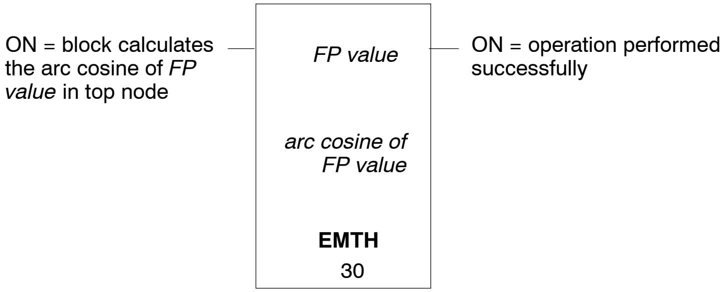
The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the sine of an angle between -π/2 ... π/2 radians. This value—the sine of an angle—must be in the range of -1.0 ... +1.0; if not:

- The arcsine is not computed
- An invalid result is returned
- An error is flagged in EMTH function #38

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the arcsine in radians of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Arc Cosine of an Angle in Radians



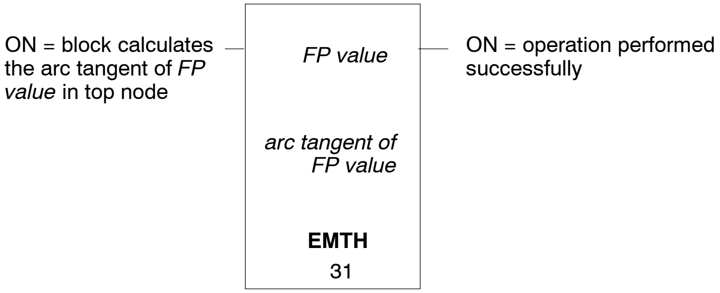
The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the cosine of an angle between 0 ... π radians. This value must be in the range of -1.0 ... +1.0; if not:

- The arc cosine is not computed
- An invalid result is returned
- An error is flagged in EMTH function #38

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the arc cosine in radians of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Arc Tangent of an Angle in Radians

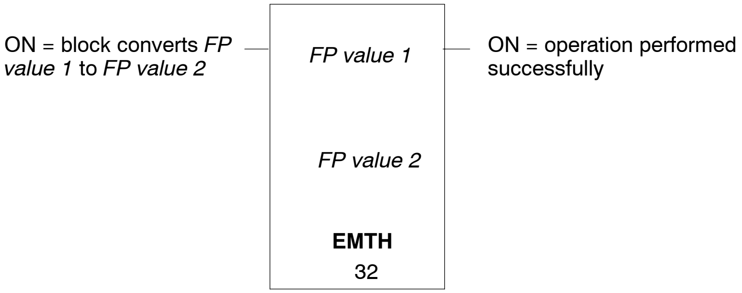


The top node comprises two consecutive 4*x* registers that contain an *FP value* indicating the tangent of an angle between -π/2 ... π/2 radians. Any valid FP value is allowed.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the arc tangent in radians of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Conversion of Radians to Degrees

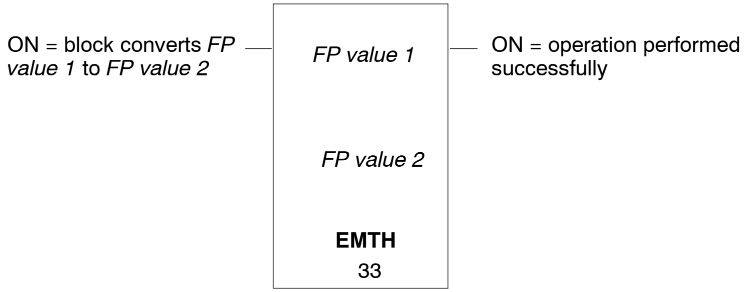


The top node comprises two consecutive 4*x* registers that contain an FP representation of the value of an angle in radians.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the FP representation of the top-node value converted to degrees.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Conversion of Degrees to Radians

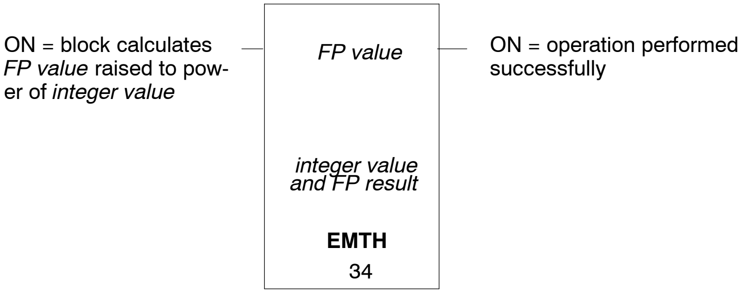


The top node comprises two consecutive 4*x* registers that contain an FP representation of the value of an angle in degrees.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the FP representation of the top-node value converted to radians.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

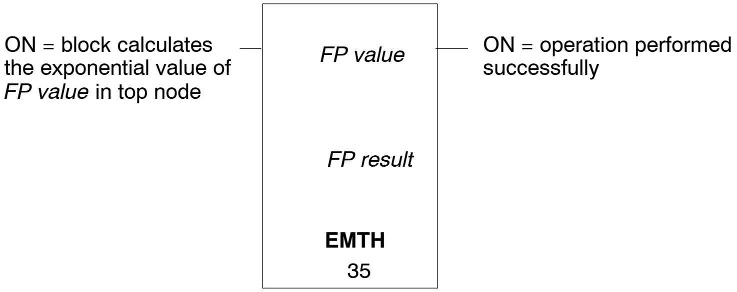
FP Number Raised to an Integer Power



The top node comprises two consecutive 4*x* registers that contain a *floating point value*.

The middle node contains four 4*x* registers—register 4*x* must be 0, register 4*x* + 1 contains an *integer value*; 4*x* + 2 and 4*x* + 3 contain the *FP result* of the *FP value* being raised to the power of the *integer value*.

FP Exponential Function

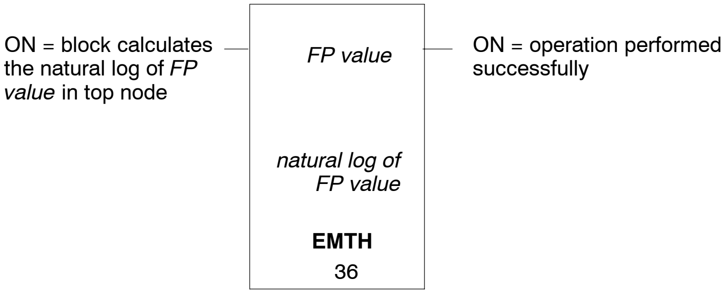


The top node comprises two consecutive 4*x* registers that contain an *FP value* in the range -87.34 ... +88.72. If the *value* is out of range, the *result* will either be 0 or the maximum value, but no error will be flagged.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the IEEE floating point format of the value in the top node.

☞ **Note** If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Natural Logarithm

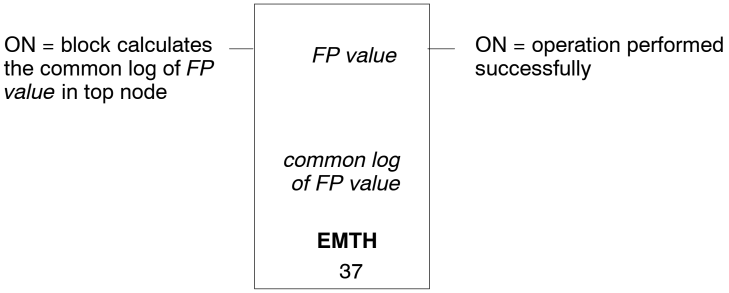


The top node comprises two consecutive 4*x* registers that contain an *FP value* > 0. If the *value* $\leq$ 0, an invalid result will be returned in the middle node and an error will be logged in EMTH function #38.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the natural logarithm of the *FP value* in the top node.

☞ **Note** If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Common Logarithm

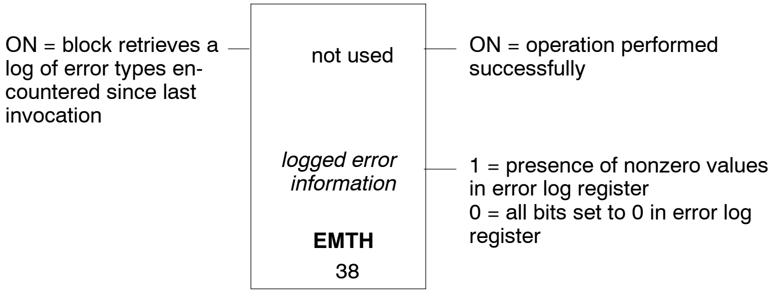


The top node comprises two consecutive 4*x* registers that contain an *FP value* > 0. If the *value* $\leq$ 0, an invalid result will be returned in the middle node and an error will be logged in EMTH function #38.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 and 4*x* + 3 contain the common logarithm of the *FP value* in the top node.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since the first two middle-node registers are not used.

FP Error Report Log

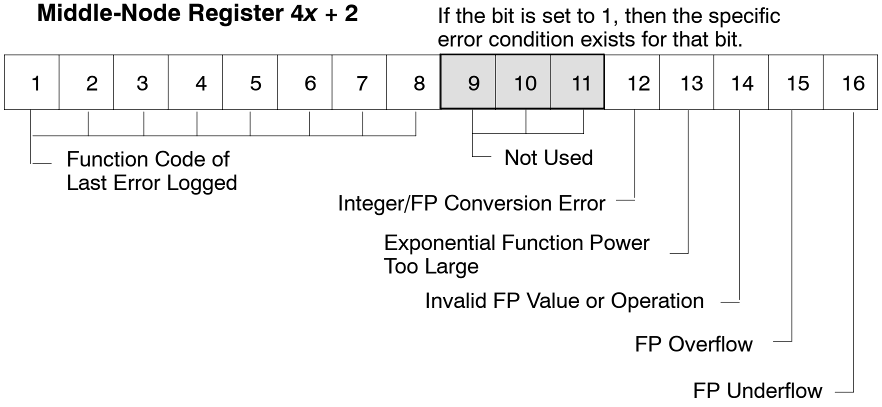


The top node requires the assignment of two consecutive 4*x* registers, but they are not used in the operation.

The middle node contains four consecutive 4*x* registers—registers 4*x* and 4*x* + 1 are not used; 4*x* + 2 contains the error log data, and 4*x* + 3 is set to 0.

Note If you want to preserve registers, you may make registers 4*x* and 4*x* + 1 in the middle node = 4*x* and 4*x* + 1 in the top node, since these registers must be assigned but are not used.

Middle-Node Register 4*x* + 2

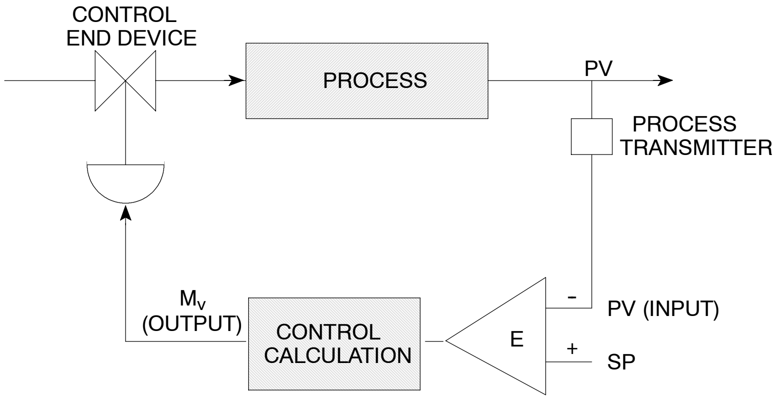


20.6 A Closed Loop Control System

An analog closed loop control system is one in which the deviation from an ideal process condition is measured, analyzed, and adjusted in an attempt to obtain (and maintain) zero error in the process condition. Provided with the Enhanced Instruction Set is a proportional-integral-derivative function block called PID2, which allows you to establish closed loop (or *negative feedback*) control in ladder logic.

20.6.1 Set Point and Process Variable

The desired (zero error) control point, which you will define in the PID2 block, is called the *set point* (SP). The conditional measurement taken against SP is called the process variable (PV). The difference between the SP and the PV is the *deviation* or *error* (E). E is fed into a control calculation that produces a *manipulated variable* (M_v) used to adjust the process so that PV = SP (and, therefore, E = 0).



20.6.2 Proportional Control

With proportional-only control (P), you can calculate the manipulated variable by multiplying error by a proportional constant, K_1, then adding a bias:

$M_v = K_1E + \text{bias}$

However, process conditions in most applications are changed by other system variables so that the bias does not remain constant; the result is offset error, where PV is constantly offset from the SP. This condition limits the capability of proportional-only control.

20.6.3 Proportional-Integral Control

To eliminate this offset error without forcing you to manually change the bias, an integral function can be added to the control equation:

$$M_v = K_1(E + K_2 \int_0^t E\Delta t)$$

Proportional-integral control (PI) eliminates offset by integrating E as a function of time. K_1 is the integral constant expressed as rep/min. As long as $E \neq 0$, the integrator increases (or decreases) its value, adjusting M_v. This continues until the offset error is eliminated.

20.6.4 Proportional-Integral-Derivative Control

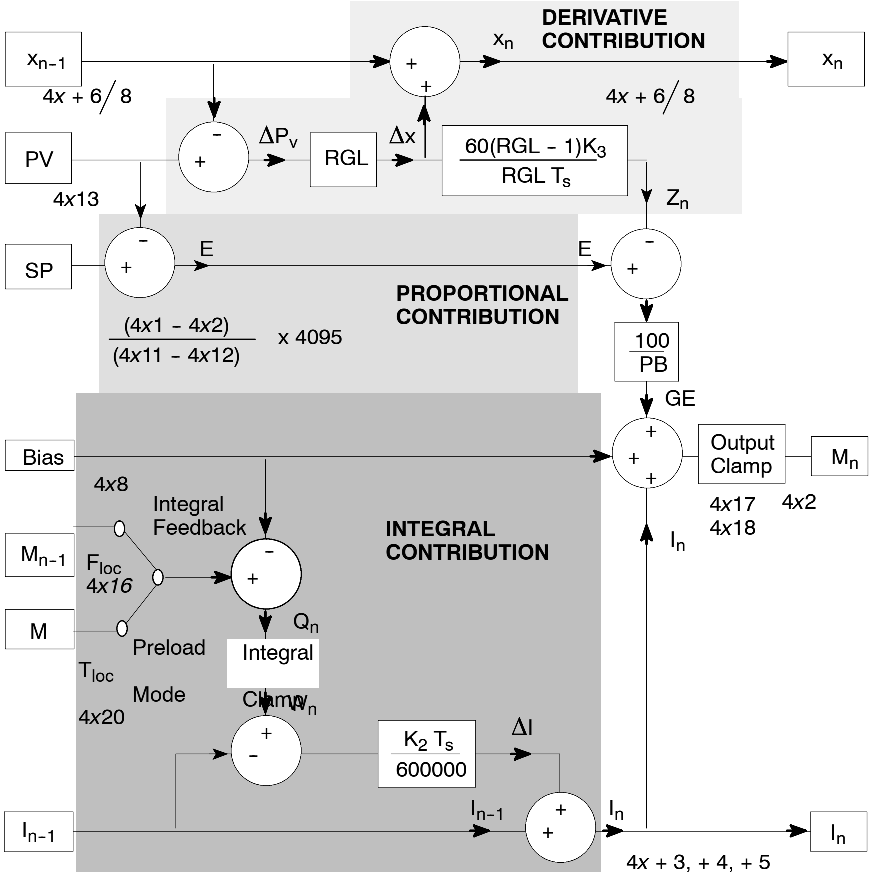
You may want to add derivative functionality to the control equation to minimize the effects of frequent load changes or to override the integral function in order to get to the SP condition more quickly:

$$M_v = K_1(E + K_2 \int_0^t E\Delta t + K_3 \frac{\Delta PV}{\Delta t})$$

Proportional-integral-derivative (PID) control can be used to save energy in the process or as a safety valve in the event of a sudden, unexpected change in process flow. K_3 is the derivative time constant expressed as min. ΔPV is the change in the process variable over a time period of Δt.

20.7 The PID2 Algorithm

Modicon's algorithm for PID2 tunes the closed loop operation in a manner similar to traditional pneumatic and analog electronic loop controllers. It uses a rate gain limiting (RGL) filter on the PV as it is used for the derivative term only, thereby filtering out higher-frequency PV noise sources (random and process generated).



PID2 Algorithm Block Diagram

where:

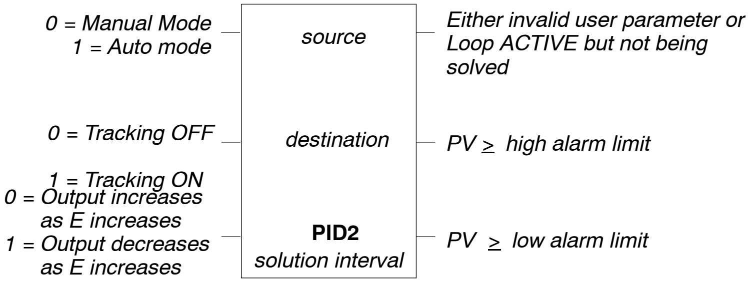
E = error, expressed in raw analog units
SP = set point, in the range 0 ... 4095
PV = process variable, in the range 0 ... 4095
x = filtered PV
K_2 = integral mode gain constant, expressed in 0.01 min^{-1}
K_3 = derivative mode gain constant, expressed in hundredths of a minute
RGL = rate gain limiting filter constant, in the range 2 ... 30
T_s = solution time, expressed in hundredths of a second
PB = proportional band, in the range 5 ... 500%
$bias$ = loop output bias factor, in the range 0 ... 4095
M = loop output
GE = gross error, the proportional-derivative contribution to the loop output
Z = derivative mode contribution to GE
Q_n = unbiased loop output
F = feedback value, in the range 0 ... 4095
I = integral mode contribution to the loop output
I_{low} = anti-reset-windup low SP, in the range 0 ... 4095
I_{high} = anti-reset-windup high SP, in the range 0 ... 4095

$$\mathrm{K}_1 = \frac{100}{\mathrm{PB}}$$

☞ **Note** The integral mode contribution calculation actually integrates the difference of the output and the integral sum—this is effectively the same as integrating the error.

20.8 PID2

The PID2 instruction implements an algorithm that performs proportional-integral-derivative operations. PID2 is a three-node function block:



The top *source* node indicates the first of 21 consecutive holding registers ranging from 4*x*0 ... 4*x*20. The contents of registers 4*x*5, 4*x*6, 4*x*7, and 4*x*8 in the top node determine whether the operation will be P, PI, or PID:

Function	**4*x*5**	**4*x*6**	**4*x*7**	**4*x*8**
P	✓			✓
PI	✓	✓		
PID	✓	✓	✓	

✓ = A non-zero value within the permissible range

The middle node contains nine additional holding registers, 4*x* ... 4*x* + 8, which are used by the PID2 block for calculations. You do not need to load anything into these registers.

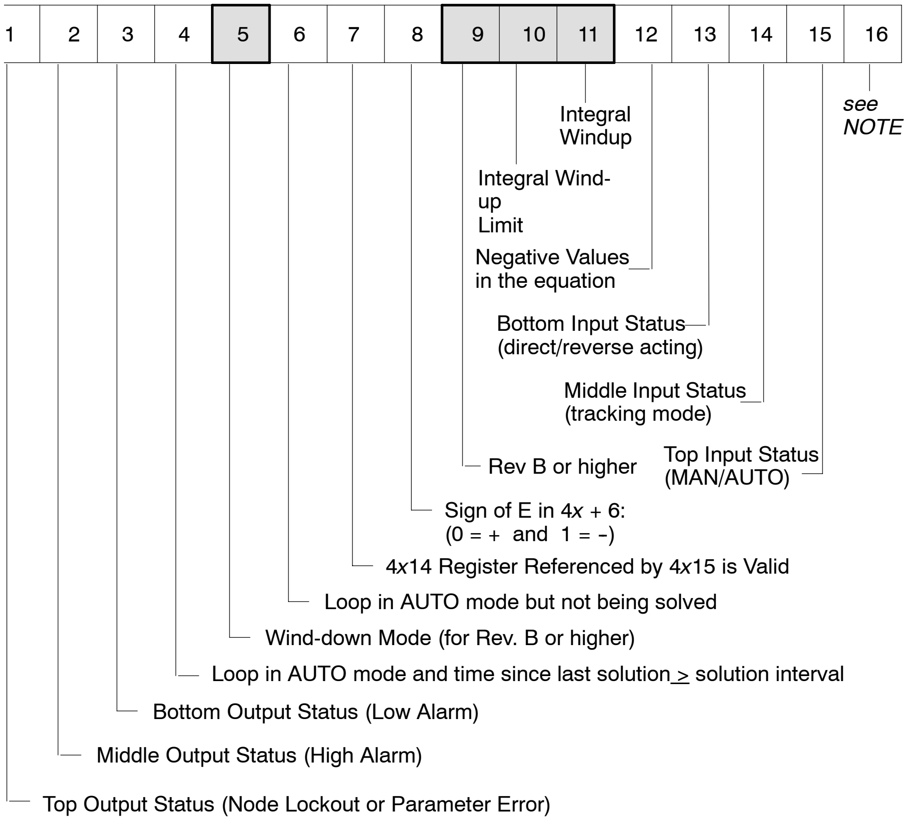
The bottom node indicates that this is a PID2 function and contains a number ranging from 1 ... 255, indicating how often the function should be performed. The number represents a time value in tenths of a second—for example, the number 17 indicates that the PID function should be performed every 1.7 s.

Top Node Register	Function
4*x*0	**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: $\text{Scaled PV} = \frac{4x13}{4095} \text{ x } (4x11 - 4x12) + 4x12$ Truncate the resulting number at the decimal point—discard all digits to the right of the decimal point, and do not round off
4*x*1	**SP**: You must specify the set point in engineering units; the value must be > 4*x*11 and > 4*x*12
4*x*2	**M_V**: Loaded by the block every time the loop is solved; it is clamped to a range of 0 ... 4095, making the output compatible with an analog output module; the manipulated variable register may be used for further CPU calculations such as cascaded loops
4*x*3	**High Alarm Limit**: Load a value in this register to specify a high alarm for PV (at or above SP); enter the value in engineering units within the range specified by 4*x*11 and 4*x*12
4*x*4	**Low Alarm Limit**: Load a value in this register to specify a low alarm for PV (at or below SP); enter the value in engineering units within the range specified by 4*x*11 and 4*x*12
4*x*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
4*x*6	**Reset Time Constant**: Load this register to add integral action to the calculation; enter a value between 0000 ... 9999 to represent a range of 00.00 ... 99.99 repeats/min; the larger the number, the larger the integral contribution; a value < 9999 or > 0000 stops the PID2 calculation
4*x*7	**Rate Time Constant**: Load this register to add derivative action to the calculation; enter a value between 0000 ... 9999 to represent a range of 00.00 ... 99.99 repeats/min; the larger the number, the larger the derivative contribution; a value < 9999 or > 0000 stops the PID2 calculation
4*x*8	**Bias**: Load this register to add a bias to the output; the value must be between 000 .... 4095, and added directly to M_V

Register	Function
4*x*9	**High Integral Windup Limit**: Load this register with the upper limit of the output value (between 0 ... 4095) where the *anti-reset windup* takes effect; the updating of the integral sum is stopped if it goes above this value—this is normally 4095
4*x*10	**Low Integral Windup Limit**: Load this register with the lower limit of the output value (between 0 ... 4095) where the anti-reset windup takes effect—this 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 range must be given 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 range must be given as a positive integer between 0 ... 9998, and it must be less than the value in register 4*x*11; it corresponds to a raw analog input value of 0
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 entry is determined by discarding the most significant digit in the register where the controller will count the loops solved/scan—e.g., if the controller does the count in register 41236, load 1236 into 4*x*14; the same value must be loaded into the 4*x*14 register in every PID2 block in the logic program
4*x*15	**Maximum Number of Loops Solved In a Scan**: If register 4*x*14 contains a non-zero value, you may load a value in this register to limit the number of loops to be solved in one scan
4*x*16	**Pointer To Reset Feedback Input**: The value you load in this register points to the holding register that contains the value of feedback (F); drop the 4 from the feedback register and enter the remaining four digits in register 4*x*16; integration calculations depend on the F value being connected to M_V—i.e., as the PID2 output varies from 0 ... 4095, so should F vary from 0 ... 4095
4*x*17	**Output Clamp—High**: The value entered in this register determines the upper limit of M_V—this is normally 4095

Register	Function
4*x*18	**Output Clamp—Low**: The value entered in this register determines the lower limit of M_V—this is normally 0
4*x*19	**Rate Gain Limit (RGL) Constant**: The value entered in this register determines the effective degree of derivative filtering; the range 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; drop the 4 from the tracking register and enter the remaining four digits in register 4*x*20; the value in the T register is connected to the input of the integral lag whenever the auto bit and track bit are both true

Middle Node Register	**Function**
4*x*	**Loop Status Register**: Twelve of the 16 bits in this register are used to define loop status:



NOTE: Bit 16 is set after initial startup or installation of the loop. If you clear the bit, the following actions take place in one scan:

- The loop status register is reset
- The current value in the real-time clock is stored in register 4*x* + 1
- Registers 4*x* + 3, 4*x* + 4, and 4*x* + 5 are set to zero
- The value (4*x*13 x 8) is stored in register 4*x* + 6
- Registers 4*x* + 7 and 4*x* + 8 are cleared

4*x* + 1	**Error (E) Status Bits**: This register displays PID2 error codes as described in previous table

Middle Node Register	Function
4*x* + 2	**Loop Timer Register**: This register 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 the register is the *elapsed* time; if elapsed time $\geq$ *solution interval* (10 times the value given in the bottom node of the PID2 block), then the loop should be solved in this scan
4*x* + 3	**For Internal Use**: Integral (integer portion)
4*x* + 4	**For Internal Use**: Integral—fraction 1
4*x* + 5	**For Internal Use**: Integral—fraction 2
4*x* + 6	**P_V x 8 (Filtered)**: This register stores the result of the filtered analog input (from register 4*x*14) multiplied by 8; this value is useful in derivative control operations
4*x* + 7	**Absolute Value of E**: This register, which is updated after each loop solution, contains the absolute value of (SP - PV); bit 8 in register 4*x* + 1 indicates the sign of E
4*x* + 8	**For Internal Use**: Current solution interval

PID2 Error Codes
(Displayed in Middle Node Register 4*x* + 1)

Code	Explanation	Check These Registers
0000	No errors, all validations OK	None
0001	Scaled SP above 9999	4*x*1
0002	High alarm above 9999	4*x*3
0003	Low alarm above 9999	4*x*4
0004	Proportional band below 5	4*x*5
0005	Proportional band above 500	4*x*5
0006	Reset above 99.99 r/min	4*x*6
0007	Rate above 99.99 min	4*x*7
0008	Bias above 4095	4*x*8
0009	High integral limit above 4095	4*x*9
0010	Low integral limit above 4095	4*x*10
0011	High engineering unit scale above 9999	4*x*11
0012	Low engineering unit scale above 9999	4*x*12
0013	High E.U. below low E.U.	4*x*11 and 4*x*12
0014	Scaled SP above high E.U.	4*x*1 and 4*x*11
0015	Scaled SP below low E.U.	4*x*1 and 4*x*12
0016*	Maximum loops/scan > 9999	4*x*15
0017	Reset feedback pointer out of range	4*x*16
0018	High output clamp above 4095	4*x*17
0019	Low output clamp above 4095	4*x*18
0020	Low output clamp above high output clamp	4*x*17 and 4*x*18
0021	RGL below 2	4*x*19
0022	RGL above 30	4*x*19
0023**	Track F pointer out of range	4*x*20 and middle input ON
0024**	Track F pointer is zero	4*x*20 and middle input ON
0025*	Node locked out (short of scan time)	None

NOTE: If lockout occurs often and the parameters are all valid, increase the maximum number of loops/scan. Lockout may also occur if the counting registers in use are not cleared as required.

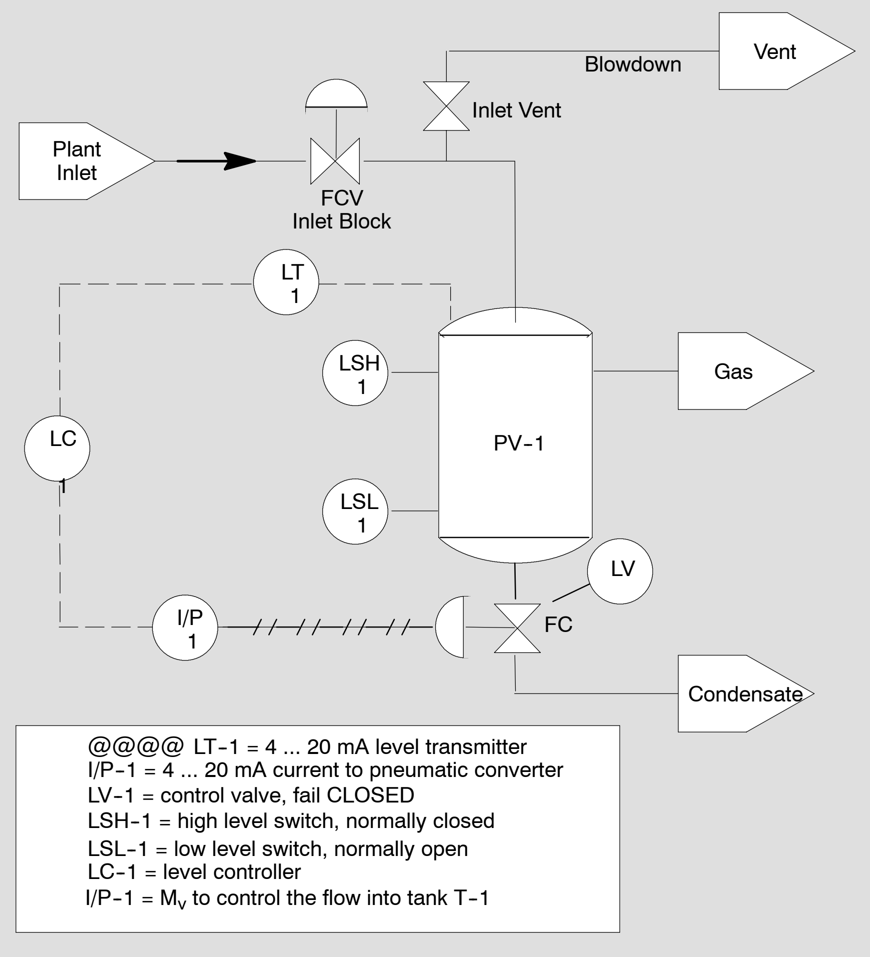
0026*	Loop counter pointer is zero	4*x*14 and 4*x*15
0027	Loop counter pointer out of range	4*x*14 and 4*x*15

* Activated by maximum loop feature—i.e., only if 4*x*15 p 0.

** Activated only if the track feature is ON—i.e., the middle input of the PID2 block is receiving power while in AUTO mode.

20.9 A Level Control Example

Here is a simplified P&I diagram for an inlet separator in a gas processing plant. There is a two-phase inlet stream—liquid and gas.

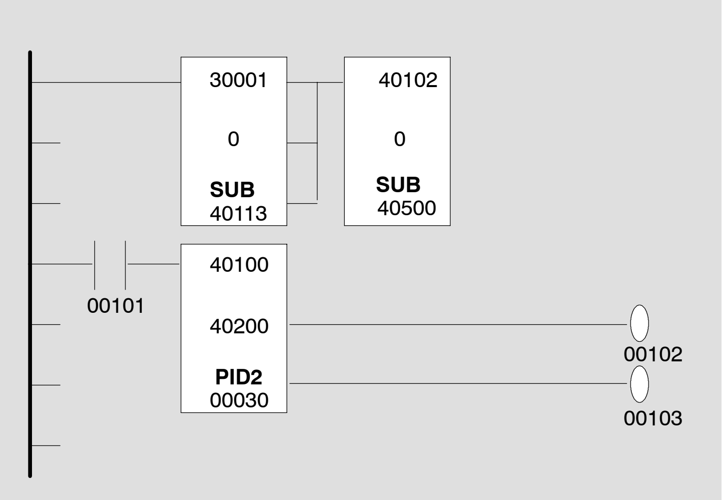


@@@@ LT-1 = 4 ... 20 mA level transmitter
I/P-1 = 4 ... 20 mA current to pneumatic converter
LV-1 = control valve, fail CLOSED
LSH-1 = high level switch, normally closed
LSL-1 = low level switch, normally open
LC-1 = level controller
I/P-1 = M_V to control the flow into tank T-1

20.10 Ladder Logic for the PID2 Example

The liquid is dumped from the tank to maintain a constant level. The control objective is to maintain a constant level in the separator. The phases must be separated before processing; separation is the role of the inlet separator, PV-1. If the level controller, LSH-1, fails to perform its job, the inlet separator could fill, causing liquids to get into the gas stream; this could severely damage devices such as gas compressors.

The level is controlled by device LC-1, a 984 controller connected to an analog input module; I/P-1 is connected to an analog output module. We can implement the control loop with the following 984 ladder logic:



The first SUB block is used to move the analog input from LT-1 to the PID2 analog input register, 40113. The second SUB block is used to move the PID2 output M_V to the traffic copped output I/P-1. Coil 00101 is used to change the loop from AUTO to MANUAL mode, if desired. For AUTO mode, it should be ON.

Specify the set point in mm for input scaling (EU). The full input range will be 0 ... 4000 mm (for 0 ... 4095 raw analog). Specify the register content of the top node in the PID2 block as follows:

40100 = Scaled PV (mm); PID2 writes this.

40101 = 2000 Scaled SP (mm). Set this to 2000 mm (half full) initially.

40102 = 0000 Loop output (0 ... 4095). PID2 writes this; keep it set to 0 just to be safe

40103 = 3500 Alarm High Set Point (mm). If the level rises above 3500 mm, coil 00102 goes ON.

40104 = 1000 Alarm Low Set Point (mm). If the level drops below 1000 mm, coil 00103 goes ON.

40105 = 0100 PB (%). The actual value used here depends on the process dynamics.

40106 = 0500 Integral constant (5.00 repeats/min). This actual value used here depends on the process dynamics.

40107 = 0000 Rate time constant (per minute). Setting this to 0 turns off the derivative mode.

40108 = 0000 Bias (0 ... 4095). This is set to 0, since we have a integral term.

40109 = 4095 High windup limit (0 ... 4095). Normally set to the maximum.

40110 = 0000 Low windup limit (0 ... 4095). Normally set to the minimum.

40111 = 4000 High engineering range (mm). The scaled value of the process variable when the raw input is at 4095.

40112 = 0000 Low engineering range (mm). The scaled value of the process variable when the raw input is at 0.

40113 = Raw analog measure (0 ... 4095). A copy of the input from the analog input module register (30001) copied by the first SUB block in the ladder logic.

40114 = 0000 Offset to loop counter register. Zero disables this feature. Normally, this is not used.

40115 = 0000 Max loops solved per scan—*see 40114*.

40116 = 0102 Pointer to reset feed back. If you leave this as zero, the PID2 function automatically supplies a pointer to the loop output register. If the actual output (40500) could be changed from the value supplied by PID2, then this register should be set to 500 (40500) to calculate the integral properly.

40117 = 4095 Output clamp high (0 ... 4095). Normally set to maximum.

40118 = 0000 Output clamp low (0 ... 4095). Normally set to minimum.

40119 = 0015 Rate Game Limit Constant (2 ... 30). Normally set to about 15. The actual value depends on how noisy the input signal is. Since we are not using derivative mode, this has no effect on the PID2 function.

40120 = 0000 Pointer to track input. Used only if the PRELOAD feature is used. If the PRELOAD is not used, this is normally 0.

The values in the registers in the 40200 destination block are all set by the PID2 block.

Chapter 21
984 Loadable Instructions

21.1 Loadable Software Packages for 984 Controllers

Two types of software loadable functions are available for 984 programmable controllers—function blocks that support optional controller modules, such as the coprocessing and Hot Standby capabilities, and function blocks that support special application or programming requirements, such as drum sequencing and the event/alarm recording system (EARS).

21.1.1 Loadable Support for Controller Option Modules

Loadable Functions	**Part Number***	**Controller Option Module**	**Controller Types Supported**
HSBY	SW-AP9X-RXA SW-AP98-RXA	AM-R911-000 AS-S911-800	chassis mounts 984-680/685/780/785 slot mounts, host based
CALL	SW-AP9X-CXB	AM-C986-004	chassis mounts
MBUS/PEER	SW-AP9X-AXA SW-AP98-AXA	AM-S975-100 AM-S975-820	chassis mounts 984-685/780/785 slot mounts, host based
MSTR**	SW-AP9X-MBP	AM-S985-0*x*0	chassis mounts

* When the X in the above software part numbers is a T, the medium is a P190 tape; when the X is a D, the software media are 5.25 in and 3.5 in diskettes.

** The MSTR function that is a loadable for the chassis mount controllers is functionally identical to the MSTR block provided in firmware for the 984-385/485/685/785 Controllers.

21.1.2 Other 984 Loadable Functions

Loadable Functions	Part Number*	Software Capability	Controller Types Supported
DRUM/ICMP	SW-SA*x*9-001 SW-AP98-S*x*A	Sequence control	chassis mounts slot mounts, host based
FN*xx*	SW-AP98-GDA	Custom loadable	slot mounts, host based
Loadables Library**	SW-AP9*x*-D*x*A	includes MATH, DMTH, TBLK, BLKT, CKSM, and PID2	chassis mounts
PID2**	SW-AP9*x*-2*x*a	PID2 closed loop control software	chassis mounts
EARS	SW-AP9D-EDA	Event/alarm recording system	All 984 controllers

* When the *x* in the above software part numbers is a T, the medium is a P190 tape; when the *x* is a D, the software media are 5.25 in and 3.5 in diskettes.

** TBLK, BLKT, CKSM, and PID2 are functionally identical to those instructions of the same name provided in firmware for the 984-385/485/685/785 Controllers.

This chapter describes all the loadable functions that support option modules except MSTR, which is described in Chapter 17.

It also describes the sequence control loadables (DRUM and ICMP), the EARS function block, and the custom loadable function block model (FN*xx*).

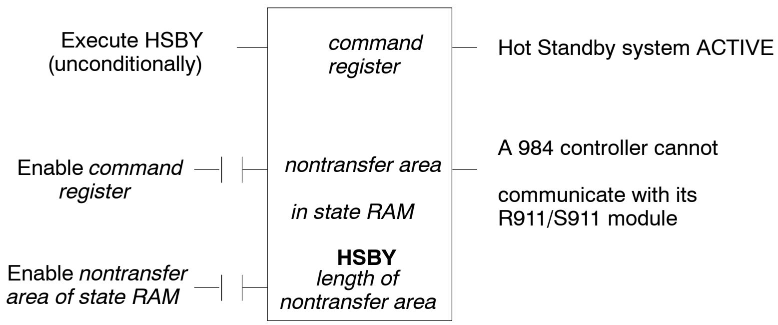
The MATH and DMTH functions—which do double precision math, square root, log, and antilog functions similar to those in EMTH (see Chapter 20)—are also described here. For descriptions of TBLK, BLKT, and PID2, refer to Chapter 20; for a description of the CKSM function, refer to Chapter 18.

21.2 The 984 Hot Standby Loadable

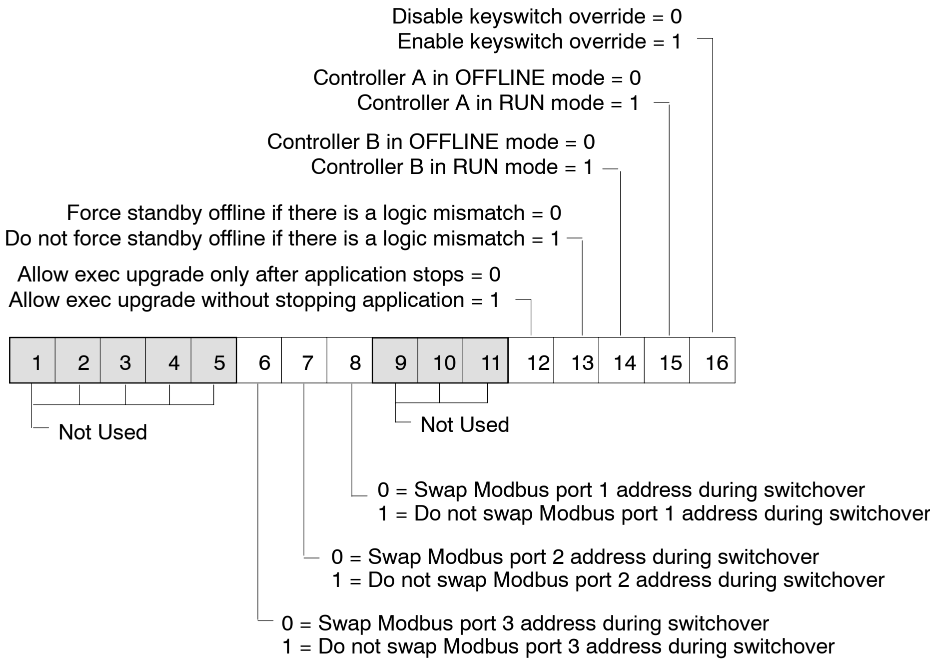
HSBY is a loadable DX function that manages a Hot Standby control system. This function block must be placed in network 1 of segment 1 in the application logic for both the primary and standby controllers. This function allows you to program a *nontransfer area* in system state RAM—an area that protects a serial group of registers in the standby controller from being modified by the primary controller.

Through the HSBY instruction you can access two registers—a *command* register and a *status* register—that allow you to monitor and control Hot Standby operations. The status register is the third register in the nontransfer area you specify.

HSBY is a three-node function block:



The top node contains a 4*x* holding register used as the HSBY command register; eight bits in this register may be configured and controlled via your panel software:



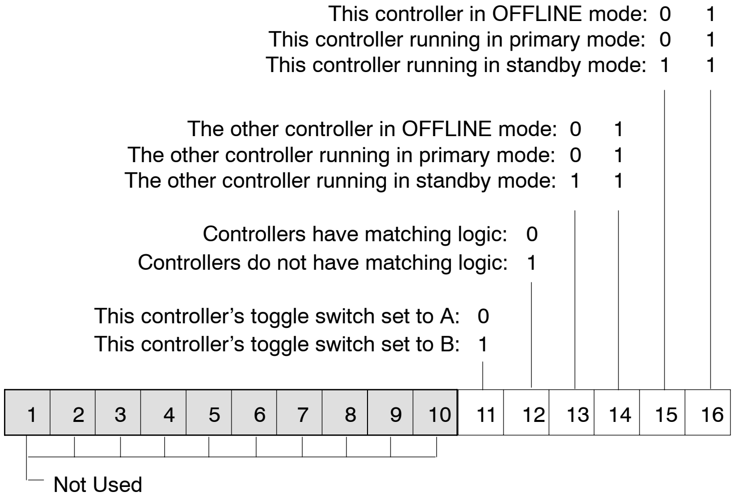
The middle node is a 4*x* register that is the first register in the *nontransfer area* in state RAM. The first three registers in the nontransfer area are special registers: 4*x* and 4*x* + 1 are the *reverse transfer registers* for passing information from the standby to the primary controller, and 4*x* + 2 is the HSBY *status register*. The total size of the nontransfer area is specified in the bottom node.

The bottom node indicates that this is an HSBY function and defines the size of the nontransfer area in state RAM. The nontransfer area must contain at least four registers. In a 16 bit CPU, the size may range from 4 ... 255 registers; in 24 bit CPUs, the size may range from 4 ... 8000 registers.

21.3 The HSBY Status Register

The HSBY status register—register 4*x* + 2 in the nontransfer area specified in the middle node of the block—contains six bits that describe the current status of the primary and standby controllers:

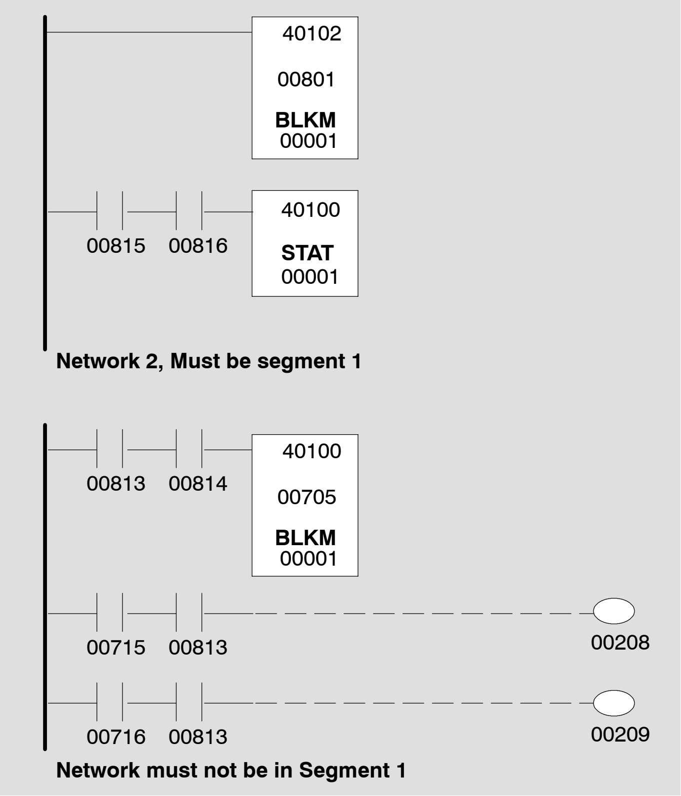
- The combined states of bits 15 and 16 tells you whether the controller you are attached to is in primary, standby, or OFFLINE mode
- The combined states of bits 13 and 14 tell you whether the other controller in the Hot Standby system is in primary, standby, or OFFLINE mode
- Bit 12 tells you whether both controllers are using identical application logic programs
- Bit 11 tells you whether the R911/S911 module in the controller you are attached to has its toggle switch set to position A or position B



The HSBY Status Register—Register 4*x* + 2 in Nontransfer Area

21.4 An HSBY Reverse Transfer Example

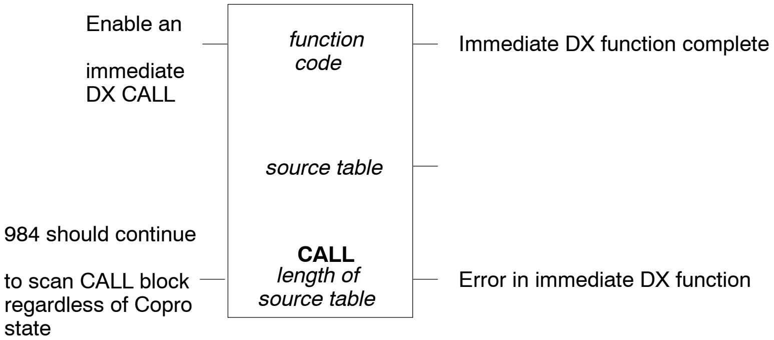
The two networks below are for a primary controller that monitors two fault lamps and a reverse transfer that sends status data from the standby controller to the primary controller. The first network must be network 2 of segment 1; the second network must *not* be in segment 1.



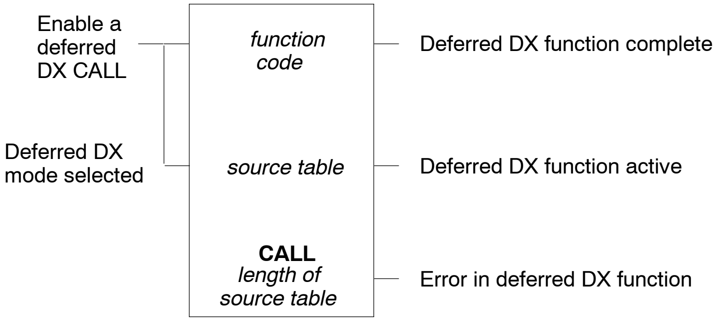
The first BLKM function transfers the HSBY status register (40102) to internal coils (00801). The STAT block, which is enabled if the other controller is in standby mode, sends one status register word from the standby controller to a reverse transfer register (40100) in the primary controller.

21.5 CALL Blocks for the 984 Coprocessors

A CALL instruction activates an immediate or deferred DX function from a library of functions defined by function codes. The Copro copies the data and function code into its local memory, processes the data, and copies the results back to Controller memory (see Section 2.4). CALL is a three-node function block:



An Immediate DX CALL Block



A Deferred DX CALL Block

The top node specifies as a constant or in a 4*x* holding register containing a *function code* to be executed. The codes fall into two ranges: numbers 0 ... 499 are available for *user-definable* DXs, and numbers 500 ... 9999 are *system* DXs provided by Modicon:

System Immediate DX Functions

Name	Code	Function
f_config	500	Obtain Copro configuration data
f_2md_fl	501	Convert a two-register long integer to 64-bit floating point
f_fl_2md	502	Convert floating point to two-register long integer
f_4md_fl	503	Convert a four-register long integer to floating point
f_fl_4md	504	Convert floating point to four-register long integer
f_1md_fl	505	Convert a one-register long integer to floating point
f_fl_1md	506	Convert floating point to one-register long integer
f_exp	507	Exponential function
f_log	508	Natural logarithm
f_log10	509	Base 10 logarithm
f_pow	510	Raise to a power
f_sqrt	511	Square root
f_cos	512	Cosine
f_sin	513	Sine
f_tan	514	Tangent
f_atan	515	Arc tangent x
f_atan2	516	Arc tangent y/x
f_asin	517	Arc sine
f_acos	518	Arc cosine
f_add	519	Add
f_sub	520	Subtract
f_mult	521	Multiply
f_div	522	Divide
f_deg_rad	523	Convert degrees to radians
f_rad_deg	524	Convert radians to degrees
f_swap	525	Swap byte positions within a register
f_comp	526	Floating point compare
f_dbwrite	527	Write Copro register database from 984
f_dbread	528	Read Copro register database from 984

System Deferred DX Functions

Name	Code	Function
f_config	500	Obtain Copro configuration data (not used but must be present)
f_d_dbwr	501	Write Copro register database from 984
f_d_dbrd	502	Read Copro register database from 984
f_dgets	515	Issue dgets() on comm line
f_dputs	516	Issue dputs() on comm line
f_sprintf	518	Generate a character string
f_sscanf	519	Interpret a character string
f_egets	520	IEEE-488 gets() function
f_eputs	521	IEEE-488 puts() function.
f_ectl	522	IEEE-488 error control function

A CALL block runs a deferred DX when the middle input is enabled and an immediate DX when no middle input is programmed.

The 4*x* register in the middle node is the first in a block of registers to be passed to the Copro for processing; the number of registers in the block is defined in the bottom node.

21.6 MBUS and PEER

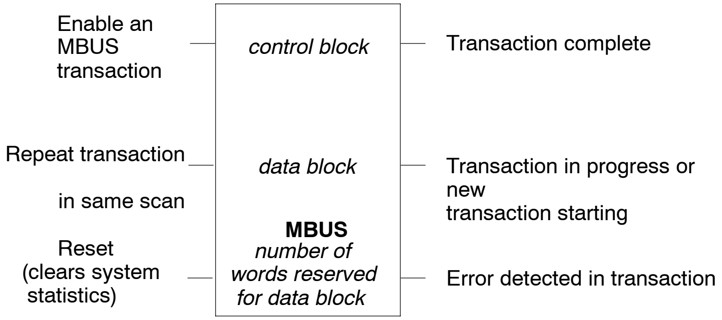
The S975 Modbus II Interface option modules use two loadable function blocks—MBUS and PEER. MBUS is always used to initiate a single transaction with another device on the Modbus II network; PEER may initiate identical message transactions with as many as 16 devices on Modbus II at one time. In an MBUS transaction, you are able to read or write discrete or register data; in a PEER transaction, you may only write register data.

Controllers on a Modbus II network can handle up to 16 transactions simultaneously. Transactions include incoming (unsolicited) messages as well as outgoing (MBUS/PEER) messages. Thus, the number of MBUS/PEER message initiations a controller can manage at any time is (16 – *# of incoming messages)*.

A transaction cannot be initiated unless the S975 has enough resources for the entire transaction to be performed. Once a transaction has been initiated, it runs until a reply is received, an error is detected, or a timeout occurs. A second transaction cannot be started in the same scan that the previous transaction completes unless the middle input is ON; a second transaction cannot be initiated by the same MBUS/PEER block until the first transaction has completed.

21.6.1 MBUS

MBUS is a three-node function block:



The top node is the first of seven 4*x* registers in the MBUS control block:

Control Block Register	Function
4*x*	Address of destination device (range: 0 ... 246)
4*x* + 1	Not used
4*x* + 2	Function code for requested action: 01 *Read discretes* 02 *Read registers* 03 *Write discrete outputs* 04 *Write register outputs* 255 *Get system statistics* (see Section 21.7)
4*x* + 3	Discrete or register reference type: 0 Discrete output (0*x*) 1 Discrete input (1*x*) 3 Input register (3*x*) 4 Holding register (4*x*)
4*x* + 4	Reference number—e.g., if you placed a 4 in register 4*x* + 3 and you place a 23 in this register, the reference will be holding register 40023
4*x* + 5	Number of words of discrete or register references to be read or written; the length limits are: *Read register* 251 registers *Write register* 249 registers *Read coils* 7848 discretes *Write coils* 7800 discretes
4*x* + 6	Time allowed for a transaction to be completed before an error is declared; expressed as a multiple of 10 ms—e.g., *100* indicates 1000 ms; the default timeout is 250 ms

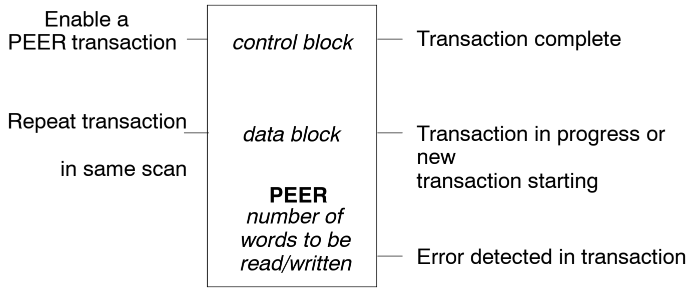
The middle node is the first 4*x* register in a *data block* to be transmitted or received in the MBUS transaction.

The number of words reserved for the data block is entered as a constant value in the bottom node. This number does not imply a data transaction length, but it can restrict the maximum allowable number of register or discrete references to be read or written in the transaction. The maximum number of words that may be used in the specified transaction is:

- 251 for reading registers (one register/word)
- 249 for writing registers (one register/word)
- 490 for reading discretes using 24 bit CPUs: 255 for reading discretes using 16 bit CPUs (up to 16 discretes/word)
- 487 for writing discretes using 24 bit CPUs; 255 for reading discretes using 16 bit CPUs (up to 16 discretes/word)

21.6.2 PEER

PEER is a three-node function block that writes 4*x* registers to multiple nodes on the network (up to 16):



The top node is the first of 19 4*x* registers in the PEER control block:

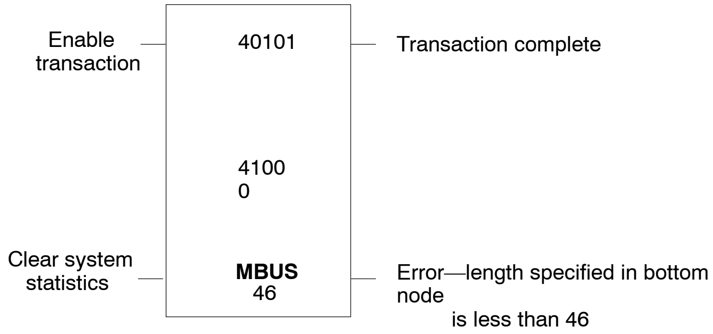
Control Block Register	Function
4*x*	Indicates the status of the transactions at each device, the leftmost bit being the status of device #1 and the rightmost bit the status of device #16: 0 = OK, 1 = transaction error
4*x* + 1	Defines the reference to the first 4*x* register to be written to in the receiving device; a 0 in this field is an invalid value and will produce an error (the bottom output will go ON)
4*x* + 2	Time allowed for a transaction to be completed before an error is declared; expressed as a multiple of 10 ms—e.g., *100* indicates 1000 ms; the default timeout is 250 ms
4*x* + 3	The Modbus port 3 address of the first of the receiving devices; address range: 1 ... 255 (0 = no transaction requested)
4*x* + 4	The Modbus port 3 address of the second of the receiving devices; address range: 1 ... 255 (0 = no transaction requested)
•	•
•	•
•	•
4*x* + 18	The Modbus port 3 address of the 16th of the receiving devices (address range: 1 ... 255)

The middle node is the first 4*x* register in a *data block* to be transmitted by the PEER function.

The bottom node contains a constant value defining the number of holding registers to be written, starting with the 4*x* register defined in the middle node; the range is 1 ... 249.

21.7 The MBUS *Get Statistics* Function

Function code 255 in register 4*x* + 2 in the MBUS control block allows you to obtain a copy of the Modbus II local statistics, which stores errors and system conditions in a series of 46 consecutive locations. When using MBUS for a *get statistics* operation, set the constant value in the bottom node to 46; any value less than 46 will return an error (the bottom output will go ON), and any value greater than 46 will reserve extra registers that cannot be used. For example:



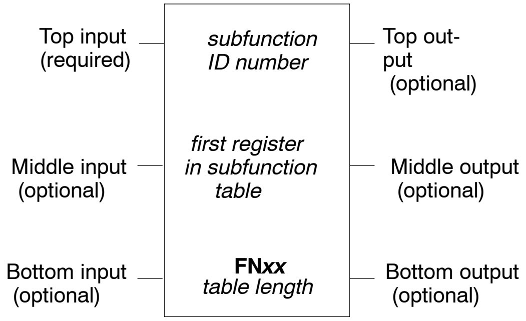
Register 40101 is the first register in the MBUS control block, making register 40103 the control register that defines the MBUS function code. By entering a value of 255 in register 40103, you implement a *get statistics* function. Registers 41000 ... 41045 are then filled with the following system statistics:

Type of Statistic	Counter Register	Type of Information
Token bus controller (TBC)	41000	Number of tokens passed by this station
	41001	Number of tokens sent by this station
	41002	Number of time the TBC has failed to pass token and has not found a successor
	41003	Number of times the station has had to look for a new successor
Software-maintained receive statistics	41004	TBC-detected error frames
	41005	Invalid request with response frames
	41006	Applications message too long
	41007	Media access control (MAC) address out of range
	41008	Duplicate application frames
	41009	Unsupported logical link control (LLC) message types
	41010	Unsupported LLC address

Type of Statistic	Counter Register	Type of Information
TBC-maintained error counters	41011	Receive noise bursts (no start delimiter)
	41012	Frame check sequence errors
	41013	E-bit error in end delimiter
	41014	Fragmented frames received (start delimiter not followed by end delimiter)
	41015	Receive frames too long
	41016	Discarded frames because there is no receive buffer
	41017	Receive overruns
	41018	Token pass failures
Software-maintained transmit errors	41019	Retries on request with response frames
	41020	All retries performed and no response received from unit
Software-maintained receive errors	41021	Bad transmit request
	41022	Negative transmit confirmation
User logic transaction errors	41023	Message sent but no application response
	41024	Invalid MBUS/PEER logic
Manufacturing message format standard (MMFS) errors	41025	Command not executable
	41026	Data not available
	41027	Device not available
	41028	Function not implemented
	41029	Request not recognized
	41030	Syntax error
	41031	Unspecified error
	41032	Data request out of bounds
	41033	Request contains invalid 984 address
	41034	Request contains invalid data type
	41035	None of the above
Background statistics	41036	Invalid MBUS/PEER request
	41037	Number of unsupported MMFS message types received
	41038	Unexpected response or response received after timeout
	41039	Duplicate application responses received
	41040	Response from unspecified device
	41041	Number of responses buffered to be processed (in the least significant byte); Number of MBUS/PEER requests to be processed (in the most significant byte)
	41042	Number of received requests to be processed (in the least significant byte); Number of transactions in process (in the most significant byte)
	41043	S975 scan time in 10 microsecond increments
Software revision	41044	Version level of fixed software (PROMs): major version number in most significant byte; minor version number in least significant byte
	41045	Version of loadable software(EEPROMs): major version number in most significant byte; minor version number in least significant byte

21.8 Designing Custom Loadable Functions

Modicon offers a custom loadable software package (SW-AP98-GDA) that allows you to design your own function blocks for operation with slot mount controllers. The operational unit for the custom loadable support software is a three-node block, FN*xx*; the package allows you to create up to 99 unique FN*xx* blocks. Within each block, you may design a large number of subfunctions—up to 8192.



The top node may be either a 4*x* holding register or a constant value; it is used to identify a *subfunction ID number*. Valid ID numbers range from 0 ... 9999, and as many as 8192 different subfunctions may be designed within a block. When multiple subfunctions are designed within an FN*xx* block, each subfunction within the block must have a unique ID number, but those numbers do not have to be consecutive.

The middle node is the first 4*x* register in a table of registers to be used by the subfunction. The table may be used to pass data to the subfunction and store results. The table format may be customized for your requirements, and each subfunction developed within the function block may have its own format.

The bottom node defines the function number, which may range from FN01 ... FN99, and uses a constant value to define the number of 4*x* registers in the subfunction table—the *table length* range may be from 1 ... 255 in a 16 bit CPU and from 1 ... 999 in a 24 bit CPU.

21.8.1 Programming Considerations

21.8.1.1 Programming Environment

This development package is for experienced C or Assembly Language programmers, and the development environment is outside the standard ladder logic programming environment. Custom loadable function blocks may be developed on IBM-AT or compatible computers running MS-DOS, Rev. 3.2 or greater. The resulting blocks may be downloaded to a standard disk-based programming panel and used in ladder logic programs.

21.8.1.2 Creating a Subfunction Library

Each subfunction built into an FN*xx* loadable block is comparable to a standard three-node DX function and requires a certain amount of user logic memory upon installation. A large number of subfunctions can be written and stored in a subfunction library in the development environment, and the size of this library can be far in excess of available memory in the target controller. Only particular subfunctions for immediate use can be pulled from the library and compiled in the FN*xx* function as it is built. The controller needs only enough extra memory to support the installed subfunctions.

21.8.1.3 Naming Subfunctions

In addition to an individual ID number, each subfunction in a customized function block is assigned a name by the programmer. The name may contain from one to four alphabetical characters, either upper or lower case. The programmer creates a separate file—the subfunction list file—where a subfunction ID number is linked to each subfunction name, and the name can be used by utility tools to access and display the subfunction and its specific characteristics.

21.8.1.4 Assigning Opcodes to Functions

Each FN*xx* function must be assigned an opcode that is in the valid range of Modicon opcodes and that is not used by any other function block currently installed in the programmable controller (see Chapter 6). If you have designed multiple custom loadable functions but intend to download only some of them together at any one time, then you need only assign as many unique opcodes as there are custom functions downloaded at any one time. However, you must inform the user how to change opcodes using the *lodutil* utility as one function is withdrawn and replaced by another. The fact that you are able to create so many subfunctions within one function allows you to work around the finite limit of available opcodes.

21.9 Sequential Control Functions

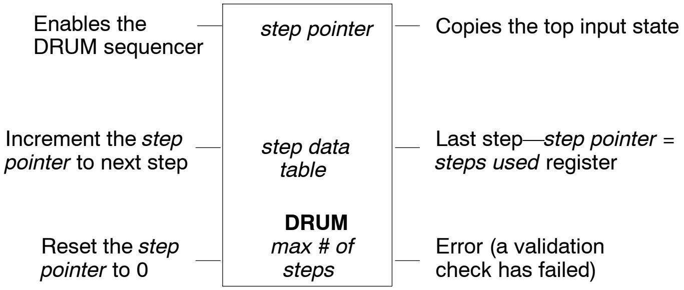
Modicon provides a drum sequencer software package, for use with 984 chassis mount controllers, which can be used in sequential control applications where simultaneous control of multiple devices—e.g., motors, valves, solenoids—at different steps in a process is required. The package consists of two loadable instructions—DRUM and ICMP—along with a DOS-based user interface. The DRUM instruction uses software to emulate a Tenor drum in ladder logic. The ICMP instruction is an *input compare* function used with DRUM to verify the correct operation of each step in the drum sequence.

21.9.1 DRUM

The DRUM function operates on a table of 4*x* registers containing data representing the desired status of 16 outputs for each step in a sequence. The number of these registers associated with a DRUM block is dependent upon the number of steps required in the sequence.

You may pre-allocate registers used to store data for each step in the sequence, thereby allowing you to add future sequencer steps without having to modify application logic.

DRUM blocks incorporate an output mask that allows you to selectively mask bits in the register data before writing it to coils. This is particularly useful when all physical sequencer outputs are not contiguous on the output module. Masked bits are not altered by the DRUM instruction, and may be used by logic unrelated to the sequencer. DRUM is a three-node function block:



The top node contains one 4*x* register used to hold the current step number. The maximum number of steps allowed is specified in the bottom node. The value in this register is referenced by the DRUM instruction each time it is solved. If the middle input to the block is ON, the contents of the register in the top node are incremented to the next step in the sequence before the block is solved.

The middle node contains the first 4*x* register in an implied register table of step data information; the first six registers in the table hold constant and variable data required to solve the block:

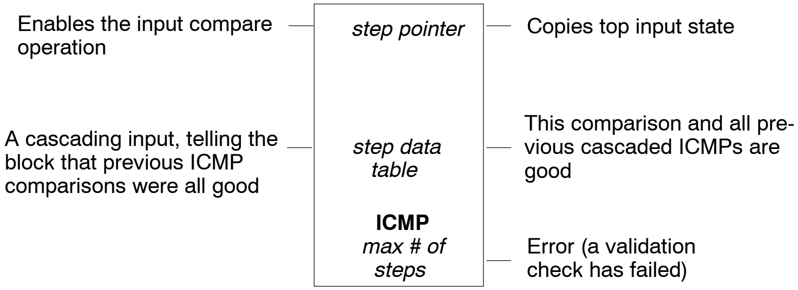
Reference	**Register Name**	**Description**
4*x*	*masked output data*	Loaded by DRUM each time the block is solved; contains the contents of the *current step data* regiater masked with the *output mask* register
4*x* + 1	*current step data*	Loaded by DRUM each time the block is solved; contains data from the *step pointer*; causes the block logic to automatically calculate register offsets when accessing step data in the *step data table*
4*x* + 2	*output mask*	Loaded by user before using the block, DRUM will not alter *output mask* contents during logic solve; contains a mask to be applied to the data for each sequencer step
4*x* + 3	*machine ID number*	Identifies DRUM/ICMP blocks belonging to a specific machine configuration; value range: 0 ... 9999 (0 = block not configured); all blocks belonging to same machine configuration have the same *machine ID number*
4*x* + 4	*profile ID number*	Identifies profile data currently loaded to the sequencer; value range: 0 ... 9999 (0 = block not configured); all blocks with the same *machine ID number* must have the same *profile ID number*
4*x* + 5	*steps used*	Loaded by user before using the block, DRUM will not alter *steps used* contents during logic solve; contains between 1 ... 255 for 16 bit CPUs and 1 ... 999 for 24 bit CPUs, specifying the actual number of steps to be solved; the number must be $\leq$ the *table length* in the bottom node of the DRUM block

The remaining registers contain data for each step in the sequence.

The bottom node contains a constant value used to calculate the maximum number of registers allocated to the *step data table*; the number may range from 1 ... 255 in 16 bit CPUs and 1 .. 999 in 24 bit CPUs. The maximum number of registers is the specified constant + 6. The specified constant must be $\geq$ the value placed in the *steps used* register in the middle node.

21.9.2 ICMP

ICMP (input compare) provides logic for verifying the correct operation of each step processed by a DRUM block. Errors detected by ICMP may be used to trigger additional error-correction logic or to shut down the system. ICMP and DRUM are synchronized through the use of a common *step pointer* register. As the pointer increments, ICMP moves through its data table in lock step with DRUM. As ICMP moves through each new step, it compares—bit for bit—the live input data to the expected status of each point in its data table. ICMP is a three-node function block:



The top node contains one 4*x* register used to hold the current step number value. The value is referenced by ICMP each time the instruction is solved; the value in this register must be controlled externally by a DRUM function or by other user logic. The same register must be used in the top node of all ICMP and DRUM blocks that are to be solved as a single sequencer.

The middle node contains the first 4*x* register in an implied register table of step data information; the first eight registers in the table hold constant and variable data required to solve the block:

Reference	**Register Name**	**Description**
4*x*	*raw input data*	Loaded by user from a group of sequential inputs to be used by ICMP for current step
4*x* + 1	*current step data*	Loaded by ICMP each time the block is solved; contains a copy of data in the *step pointer*; causes the block logic to automatically calculate register offsets when accessing step data in the *step data table*
4x + 2	*input mask*	Loaded by user before using the block; contains a mask to be ANDed with *raw input data* for each step—masked bits will not be compared; masked data are put in the *masked input data* register

Reference	Register Name	Description
4*x* + 3	*masked input data*	Loaded by ICMP each time the block is solved; contains the result of the ANDed *input mask* and *raw input data*
4*x* + 4	*compare status*	Loaded by ICMP each time the block is solved; contains the result of an XOR of the *masked input data* and the *current step data*; unmasked inputs that are not in the correct logical state cause the associated register bit to go to 1—non-zero bits cause a miscompare, and middle output will not go ON
4*x* + 5	*machine ID number*	Identifies DRUM/ICMP blocks belonging to a specific machine configuration; value range: 0 ... 9999 (0 = block not configured); all blocks belonging to same machine configuration have the same *machine ID number*
4*x* + 6	*profile ID number*	Identifies profile data currently loaded to the sequencer; value range: 0 ... 9999 (0 = block not configured); all blocks with the same *machine ID number* must have the same *profile ID number*
4*x* + 7	*steps used*	Loaded by user before using the block, DRUM will not alter *steps used* contents during logic solve; contains between 1 ... 255 for 16 bit CPUs and 1 ... 999 for 24 bit CPUs, specifying the actual number of steps to be solved; the number must be $\leq$ the *table length* in the bottom node of the ICMP block

The remaining registers contain data for each step in the sequence.

The bottom node contains a constant value used to calculate the maximum number of registers allocated to the *step data table*; the number may range from 1 ... 255 in 16 bit CPUs and 1 .. 999 in 24 bit CPUs. The maximum number of registers is the specified constant + 8. The specified constant must be $\geq$ the value placed in the *steps used* register in the middle node.

21.9.3 Cascaded DRUM/ICMP Blocks

A series of DRUM and/or ICMP blocks may be cascaded to simulate a mechanical drum up to 512 bits wide. Programming the same 4*x* register reference into the top node of each related block causes them to cascade and step as a grouped unit without the need of any additional application logic. All DRUM/ICMP blocks with the same register reference in the top node are automatically synchronized. The must also have the same constant value in the bottom node, and must be set to use the same value in the *steps used* register in the middle node.

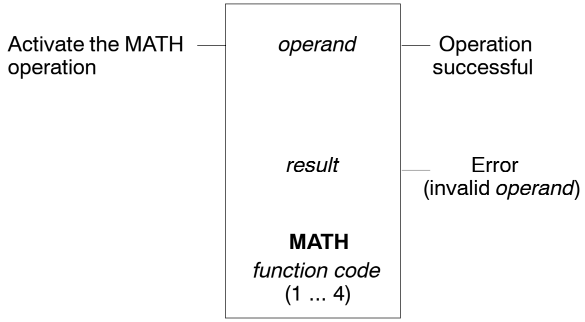
21.10 Extended Math Loadables

Included in the loadables library provided for chassis mount controllers are two extended math instructions—MATH and DMTH—which provide you with double precision math, square root, process square root, log, and antilog functions comparable to those in the EMTH instruction (Section 20.2).

> **Note** The BLKM, TBLK, PID2 functions included in the loadables library are functionally identical to the functions of the same names described in Chapter 20. The CKSM function in the loadables library is functionally identical to the function described in Chapter 18.

21.10.1 MATH

The MATH function performs any one of four integer math operations. MATH is a three-node function block:



The top node requires either two consecutive 4*x* registers or one 3*x* register. The selected operation is performed on the value held in the register(s). The four different operation types (as specified by code number in the bottom node) each has specific limits on the *operand* value allowed in the register(s):

- For integer square root functions, the value stored in each register cannot exceed 9999, permitting a maximum stored value of 99,999,999 in the 4*x* registers and a maximum stored value of 9,999 in the 3*x* register

- For process square root functions, the value in the 3*x* or 4*x* register must be $\leq$ 4095; thus only one register is used
- For logarithm functions, the value stored in each register cannot exceed 9999, permitting a maximum stored value of 99,999,999 in the 4*x* registers and a maximum stored value of 9,999 in the 3*x* register; the register value must not be less than 1
- For antilogarithm functions, the value stored in the 3*x* or 4*x* register must be in the range 0 ... 7999 (a maximum value of 7.999 with an implied decimal point)

The middle node is the first of two consecutive 4*x* holding registers. The *result* of the operation is stored in these two registers.

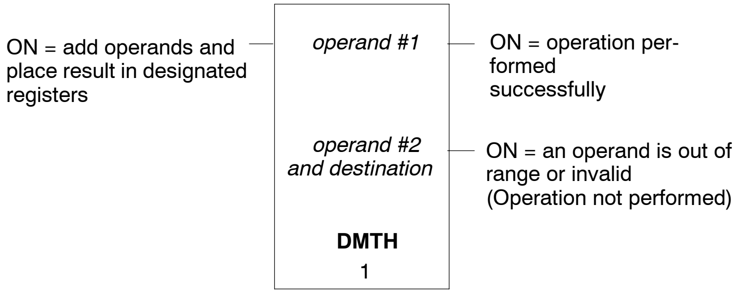
The bottom node provides the functional selection mechanism for the block. Enter a constant value in the range 1 ... 4 to indicate the integer math function you want to employ:

Code Number	Math Function
1	decimal square root
2	process square root
3	logarithm
4	antilogarithm

21.10.2 DMTH

The DMTH function performs any one of four double precision math operations. DMTH is a three-node function block with input and output lines that vary depending on the selected operation:

Double Precision Addition

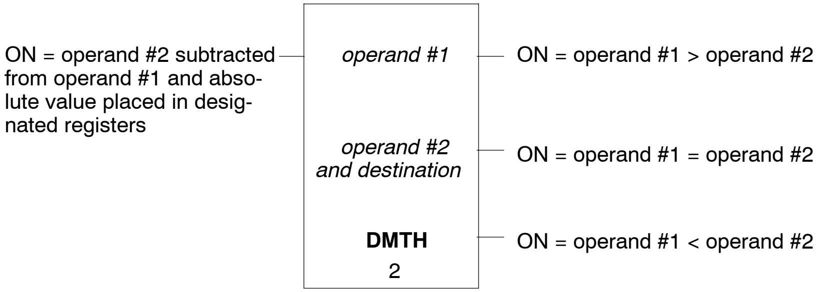


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2 indicates whether an overflow condition exists (1 = overflow)
- 4*x* + 3 and 4*x* + 4 hold the double precision addition result
- 4*x* + 5 is not used in this calculation but must exist in state RAM

Double Precision Subtraction

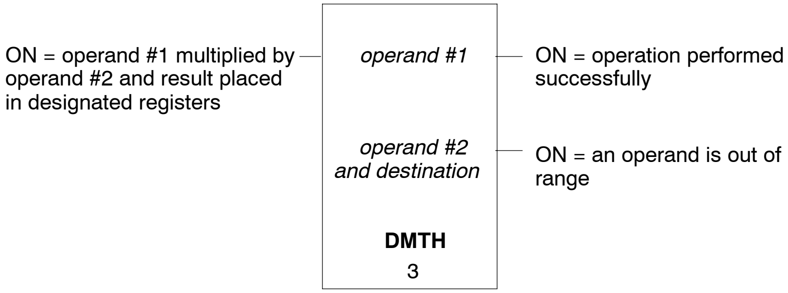


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2 and 4*x* + 3 hold the double precision subtraction result
- 4*x* + 4 indicates whether the operands are in the valid range (1 = out of range and 0 = in range)
- 4*x* + 5 is not used in this calculation but must exist in state RAM

Double Precision Multiplication

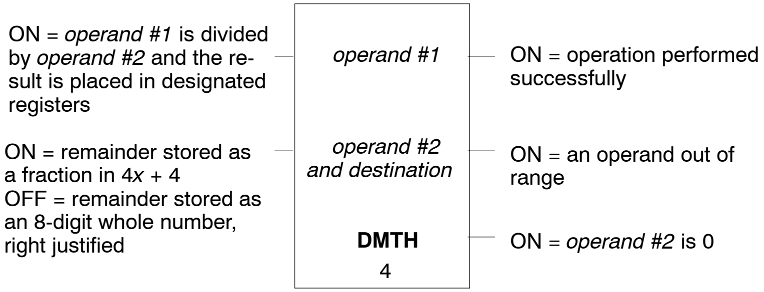


The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999
- 4*x* + 2, 4*x* + 3, 4*x* + 4, and 4*x* + 5 hold the double precision multiplication result

Double Precision Division



The top node comprises two consecutive 4*x* registers; each register holds a value in the range 0000 ... 9999 for a combined value range of up to 99,999,999.

The middle node comprises six consecutive 4*x* registers:

- 4*x* and 4*x* + 1 hold the second operand value, in the range 0 ... 99,999,999 (Since division by 0 is illegal, a 0 value causes an error—an error trapping routine sets the remaining middle-node registers to 0000 and turns the bottom output ON.)
- 4*x* + 2 and 4*x* + 3 hold an eight-digit result, the quotient

- $4x + 4$ and $4x + 5$ hold the remainder—if the remainder is expressed in whole numbers, it is eight digits long and both registers are used; if the remainder is expressed as a decimal, it is four digits long and only register $4x + 4$ is used.

21.11 The EARS Loadable

The EARS block is loaded to a 984 controller being used in an alarm/event recording system. An EARS system requires that the 984 work in conjunction with a man-machine interface (MMI) host device that runs a special off-line software package. The controller monitors a specified group of events for any changes in state and logs change data into a buffer; the data are then removed by the host over a high speed network such as Modbus II or Modbus Plus. The two devices comply with a defined handshake protocol that ensures that all data detected by the 984 controller are accurately represented in the host.

21.11.1 984 Functions in an Event/Alarm Recording System

When a 984 controller is employed in an EARS environment, it is set up to maintain and monitor two tables of 4*x* registers, one containing the *current* state of a set of user-defined events and one containing the *history* of the most recent state of these events. Event states are stored as bit representations in the 4*x* registers—a bit value of 1 signifying an ON state and a bit value of 0 signifying an OFF state. Each table can contain up to 62 registers, allowing you to monitor the states of up to 992 events.

When the controller detects a change between the current state bit and the history bit for an event, the EARS function block prepares a two-word message and places it in a circular buffer where they can be off-loaded to a host MMI. This message contains:

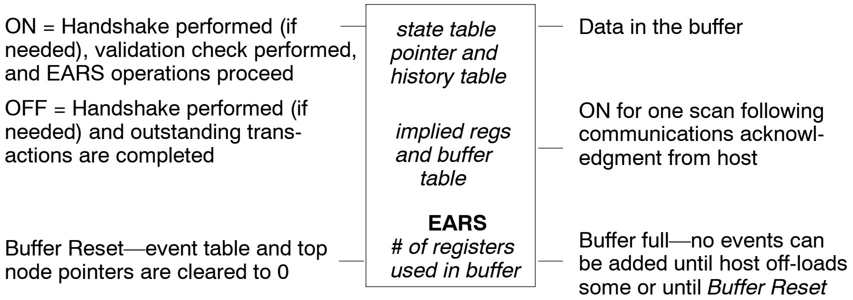
- A time stamp representing the time span from midnight to 24:00 hours in tenths of a second
- A transition flag indicating that the event is either a positive or negative transition with respect to the event state
- A number indicating which event has occurred

21.11.2 Host↔Controller Interaction

The host MMI device must be able to read and write 984 data registers via the Modbus protocol. A handshake protocol maintains integrity between the host and the circular buffer running in the 984; this enables the the host to receive events asynchronously from the buffer at a speed suitable to the host while the controller detects event changes and load the buffer at its faster scan rate.

21.11.3 The EARS Block

EARS is a three-node function block:



ON = Handshake performed (if needed), validation check performed, and EARS operations proceed

OFF = Handshake performed (if needed) and outstanding transactions are completed

Buffer Reset—event table and top node pointers are cleared to 0

Data in the buffer

ON for one scan following communications acknowledgment from host

Buffer full—no events can be added until host off-loads some or until *Buffer Reset*

The top node contains the first of 64 consecutive 4*x* registers. The first two of these registers contain values that specify the location and size of the current state table. The the remaining 62 registers are available to contain the history table:

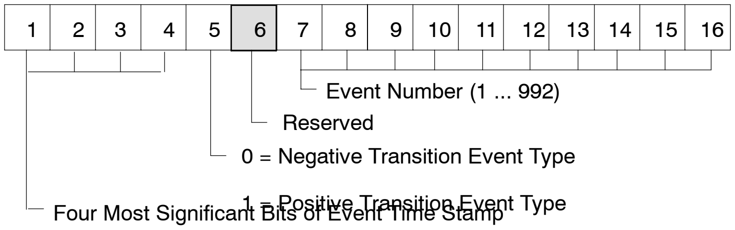
- 4*x* is the indirect pointer to the current state table—e.g., if the register contains a value of 5, then the state table begins at register 40005
- 4*x* + 1 contains a value in the range 1 ... 62 that specifies the number of registers in the current state table
- 4*x* + 2 is the first register of the history table, and the remaining registers allocated to the top node may be used in the table as required; the history table can provide monitoring for as many as 992 contiguous events (if 16 bits in all the 62 available registers are used)

If all 62 registers are not required for the history table, the extra registers may be used elsewhere in the program for other purposes, but they will still be found (by a Modbus search) in the top node of the EARS block.

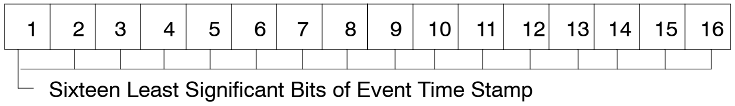
The middle node contains the first in another series of consecutive 4*x* registers. The first five registers are implied, and the rest contain the circular buffer. The circular buffer uses an even number of registers in the range 2 ... 100:

- 4*x* contains a value that defines the maximum number of registers the circular buffer may occupy
- 4*x* + 1 contains the *Q_take* pointer—the pointer to the next register where the host will go to remove data
- The low byte of register 4*x* + 2 contains the *Q_put* pointer—the pointer to the register in the circular buffer where the EARS block will begin to place the next state-change data; the high byte of register 4*x* + 2 contains the last transaction number received
- 4*x* + 3 contains the *Q+count*—a value indicating the number of words currently in the circular buffer
- 4*x* + 4 contains status/error codes
- 4*x* + 5 is the first register in the circular buffer where event-change data are stored; each detected change in event status produces two consecutive registers of information:

Event Data Register 1



Event Data Register 2



The time stamp is encoded in 20 bits as a binary weighted value that represents the time in an increment of 0.1 s starting from midnight of the day on which the status change was detected:

1 hour = 3,600 seconds = 36,000 tenths of a second, and
24 hours = 86,400 seconds = 864,000 tenths of a second

The following table shows binary weighted values for the time stamp, where *n* is the relative bit position in the 20-bit time scheme:

Event Data Register 1				**Event Data Register 2**															
19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2^n	*n*	2^n	*n*	2^n	*n*
1	0	256	8	65536	16
2	1	512	9	131072	17
4	2	1024	10	262144	18
8	3	2048	11	524288	19
16	4	4096	12		
32	5	8192	13		
64	6	16384	14		
128	7	32768	15		

☞ **Note** The real time clock in the chassis mount controllers has a tenth-of-a-second resolution, but the other 984s have real time clock chips resolve only to a second. An algorithm is used in EARS to provide a best estimate of tenth-of-a-second resolution—it is accurate in the relative time intervals between events, but it may vary slightly from the real time clock.

The bottom node displays an even constant value in the range 2 .... 100, which represents the actual number of registers allocated for the circular buffer. Each event requires two registers for data storage—therefore, if you wish to trap up to 25 events at any given time in the buffer, assign a value of 50 in the bottom node.

Index