At the end of this document you will find links to products related to this catalog. You can go directly to our shop by clicking HERE. HERE

Cat. No. W438-E1-01

SYSMAC

CX-Programmer

Ver. 5.0

WS02-CXPC1-E-V50

CS1-H, CJ1-H, CJ1M CPU Units

OPERATION MANUAL

Function Blocks

OMRON

CX-Programmer

Ver. 5.0
WS02-CXPC1-E-V50

CS1-H, CJ1-H, CJ1M CPU Units

Operation Manual
Function Blocks

Produced July 2004

Notice:

OMRON products are manufactured for use according to proper procedures by a qualified operator and only for the purposes described in this manual.

The following conventions are used to indicate and classify precautions in this manual. Always heed the information provided with them. Failure to heed precautions can result in injury to people or damage to property.

⚠ DANGER Indicates an imminently hazardous situation which, if not avoided, will result in death or serious injury.

⚠ WARNING Indicates a potentially hazardous situation which, if not avoided, could result in death or serious injury.

⚠ Caution Indicates a potentially hazardous situation which, if not avoided, may result in minor or moderate injury, or property damage.

OMRON Product References

All OMRON products are capitalized in this manual. The word “Unit” is also capitalized when it refers to an OMRON product, regardless of whether or not it appears in the proper name of the product.

The abbreviation “Ch,” which appears in some displays and on some OMRON products, often means “word” and is abbreviated “Wd” in documentation in this sense.

The abbreviation “PLC” means Programmable Controller. “PC” is used, however, in some Programming Device displays to mean Programmable Controller.

Visual Aids

The following headings appear in the left column of the manual to help you locate different types of information.

Note Indicates information of particular interest for efficient and convenient operation of the product.

1,2,3... 1. Indicates lists of one sort or another, such as procedures, checklists, etc.

© OMRON, 2004

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, mechanical, electronic, photocopying, recording, or otherwise, without the prior written permission of OMRON.

No patent liability is assumed with respect to the use of the information contained herein. Moreover, because OMRON is constantly striving to improve its high-quality products, the information contained in this manual is subject to change without notice. Every precaution has been taken in the preparation of this manual. Nevertheless, OMRON assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained in this publication.

TABLE OF CONTENTS

PRECAUTIONS xi

1 Intended Audience xii
2 General Precautions xii
3 Safety Precautions xii
4 Application Precautions xiii

SECTION 1
Introduction 1

1-1 Introducing the Function Blocks 2
1-2 Function Blocks 7
1-3 Variables 13
1-4 Converting Function Block Definitions to Library Files 16
1-5 Usage Procedures 17

SECTION 2
Specifications 19

2-1 Function Block Specifications 21
2-2 Instance Specifications 30
2-3 Restrictions on Function Blocks 37
2-4 Function Block Applications Guidelines 42
2-5 Precautions for Instructions with Operands Specifying the First or Last of Multiple Words 49
2-6 Instruction Support and Operand Restrictions 52
2-7 CPU Unit Function Block Specifications 104
2-8 Number of Function Block Program Steps and Instance Execution Time 108

SECTION 3
Creating Function Blocks 111

3-1 Procedural Flow 112
3-2 Procedures 114

Appendices

A Data Types 137
B Structured Text (ST Language) Specifications 139
C External Variables 161

Index 163

Revision History 165

TABLE OF CONTENTS

About this Manual:

This manual describes the function blocks and related functionality of the CX-Programmer Ver. 5.0 used together with CS1-H, CJ1-H, and CJ1M CPU Units with unit version 3.0 or later, and includes the sections described on the next page. The CX-Programmer Ver. 5.0 is software that enables the personal computer to be used as a function block programming device, and can be used only for SYSMAC CS-series and CJ-series CPU Units that support function blocks.

The CX-Programmer Ver. 5.0 function block functions have been enhanced. This manual describes only CX-Programmer Ver. 5.0 operations that are related to functions blocks. For operations not related to function blocks, refer to the *CX-Programmer Operation Manual* (enclosed, Cat. No. W437). This manual also provides only information related to function blocks for the CS1-H, CJ1-H, and CJ1M CPU Units. For other information, refer to the CS/CJ-series manuals.

Please read this manual and related manuals carefully and be sure you understand the information provided before attempting to install or operate the CX-Programmer Ver. 5.0 or the CS1-H, CJ1-H, or CJ1M CPU Units. Be sure to read the precautions provided in the following section.

Manuals Related to the CX-Programmer Ver. 5.0

Name	Cat. No.	Contents
SYSMAC WS02-CXPC1-E-V50 CX-Programmer Ver. 5.0 Operation Manual Function Blocks (CS1G-CPU□□H, CS1H-CPU□□H, CJ1G-CPU□□H, CJ1H-CPU□□H, CJ1M-CPU□□ CPU Units)	W438	Describes the functionality unique to the CX-Programmer Ver. 5.0 and CS/CJ-series CPU Units with unit version 3.0 or later based on function blocks. Functionality that is the same as that of the CX-Programmer is described in W437 (enclosed).
SYSMAC WS02-CXPC1-E-V50 CX-Programmer Operation Manual	W437	Provides information on how to use the CX-Programmer for all functionality except for function blocks.

Manuals Related to the CS1-H, CJ1-H, CJ1M CPU Units

Name	Cat. No.	Contents
SYSMAC CS Series CS1G/H-CPU□□-EV1, CS1G/H-CPU□□H Programmable Controllers Operation Manual	W339	Provides an outline of and describes the design, installation, maintenance, and other basic operations for the CS-series PLCs. The following information is included: An overview and features The system configuration Installation and wiring I/O memory allocation Troubleshooting Use this manual together with the W394.
SYSMAC CJ Series CJ1G-CPU□□, CJ1G/H-CPU□□H, CJ1G-CPU□□P, CJ1M-CPU□□ Programmable Controllers Operation Manual	W393	Provides an outline of and describes the design, installation, maintenance, and other basic operations for the CJ-series PLCs. The following information is included: An overview and features The system configuration Installation and wiring I/O memory allocation Troubleshooting Use this manual together with the W394.

Name	Cat. No.	Contents
SYSMAC CS/CJ Series CS1G/H-CPU□□-EV1, CS1G/H-CPU□□H, CJ1G-CPU□□, CJ1G/H-CPU□□H, CJ1G-CPU□□P, CJ1M-CPU□□ Programmable Controllers Programming Manual	W394	Describes programming and other methods to use the functions of the CS/CJ-series PLCs. The following information is included: Programming Tasks File memory Other functions Use this manual together with the W339 or W393.
SYSMAC CS/CJ Series CS1G/H-CPU□□-EV1, CS1G/H-CPU□□H, CJ1G-CPU□□, CJ1G/H-CPU□□H, CJ1G-CPU□□P, CJ1M-CPU□□ Programmable Controllers Instructions Reference Manual	W340	Describes the ladder diagram programming instructions supported by CS/CJ-series PLCs. When programming, use this manual together with the *Operation Manual* (CS1: W339 or CJ1: W393) and *Programming Manual* (W394).
SYSMAC CS/CJ Series CS1G/H-CPU□□-EV1, CS1G/H-CPU□□H, CS1W-SCB21-V1/41-V1, CS1W-SCU21/41, CJ1G-CPU□□, CJ1G/H-CPU□□H, CJ1G-CPU□□P, CJ1M-CPU□□, CJ1W-SCU21-V1/41-V1 Communications Commands Reference Manual	W342	Describes the communications commands that can be addressed to CS/CJ-series CPU Units. The following information is included: C-series (Host Link) commands FINS commands Note: This manual describes commands that can be sent to the CPU Unit without regard for the communications path, which can be through a serial communications port on the CPU Unit, a communications port on a Serial Communications Unit/Board, or a port on any other Communications Unit.

Overview of Contents

Precautions provides general precautions for using the CX-Programmer Ver. 5.0.

Section 1 introduces the function block functionality of the CX-Programmer and explains the features that are not contained in the non-function block version of CX-Programmer.

Section 2 provides specifications for reference when using function blocks, including specifications on function blocks, instances, and compatible PLCs, as well as usage precautions and guidelines.

Section 3 describes the procedures for creating function blocks on the CX-Programmer.

The ***Appendices*** provide information on data types, structure text specifications, and external variables.

⚠ WARNING Failure to read and understand the information provided in this manual may result in personal injury or death, damage to the product, or product failure. Please read each section in its entirety and be sure you understand the information provided in the section and related sections before attempting any of the procedures or operations given.

PRECAUTIONS

This section provides general precautions for using the CX-Programmer Ver. 5.0 and the Programmable Logic Controller.

The information contained in this section is important for the safe and reliable application of the CX-Programmer Ver. 5.0 and Programmable Controller. You must read this section and understand the information contained before attempting to set up or operate the CX-Programmer Ver. 5.0 and Programmable Controller.

1 Intended Audience xii
2 General Precautions xii
3 Safety Precautions xii
4 Application Precautions xiii

1 Intended Audience

This manual is intended for the following personnel, who must also have knowledge of electrical systems (an electrical engineer or the equivalent).

- Personnel in charge of installing FA systems.
- Personnel in charge of designing FA systems.
- Personnel in charge of managing FA systems and facilities.

2 General Precautions

The user must operate the product according to the performance specifications described in the operation manuals.

Before using the product under conditions which are not described in the manual or applying the product to nuclear control systems, railroad systems, aviation systems, vehicles, combustion systems, medical equipment, amusement machines, safety equipment, and other systems, machines, and equipment that may have a serious influence on lives and property if used improperly, consult your OMRON representative.

Make sure that the ratings and performance characteristics of the product are sufficient for the systems, machines, and equipment, and be sure to provide the systems, machines, and equipment with double safety mechanisms.

This manual provides information for programming and operating the product. Be sure to read this manual before attempting to use the product and keep this manual close at hand for reference during operation.

⚠ WARNING It is extremely important that a PLC and all PLC Units be used for the specified purpose and under the specified conditions, especially in applications that can directly or indirectly affect human life. You must consult with your OMRON representative before applying a PLC System to the above-mentioned applications.

3 Safety Precautions

⚠ WARNING Confirm safety sufficiently before transferring I/O memory area status from the CX-Programmer Ver. 5.0 to the actual CPU Unit. The devices connected to Output Units may malfunction, regardless of the operating mode of the CPU Unit. Caution is required in respect to the following functions.

- Transferring from the CX-Programmer to real I/O (CIO Area) in the CPU Unit using the PLC Memory Window.
- Transferring from file memory to real I/O (CIO Area) in the CPU Unit using the Memory Card Window.

⚠ Caution Variables must be specified either with AT settings (or external variables), or the variables must be the same size as the data size to be processed by the instruction when specifying the first or last address of multiple words in the instruction operand.

1. If a non-array variable with a different data size and without an AT setting is specified, the CX-Programmer will output an error when compiling.
2. Array Variable Specifications

- When the size to be processed by the instruction operand is fixed:
  The number of array elements must be the same as the number of elements to be processed by the instruction. Otherwise, the CX-Programmer will output an error when compiling.
- When the size to be processed by the instruction operand is not fixed:
  The number of array elements must be greater than or the same as the size specified in the other operands.
  - If the other operand specifying a size is a constant, the CX-Programmer Ver. 5.0 will output an error when compiling.
  - If the other operand specifying a size is a variable, the CX-Programmer Ver. 5.0 will not output an error when compiling, even if the size of the array variable is not the same as that specified by the other operand (variable). A warning message, however, will be displayed. In particular, if the number of array elements is less than the size specified by the other operand (e.g., the size of the instruction operand is 16, and the number of elements registered in the actual variable table is 10), the instruction will execute read/write processing for the area that exceeds the number of elements. For example, read/write processing will be executed for the 6 words following those for the number of elements registered in the actual variable table. If these words are used for other instructions (including internal variable allocations), unexpected operation will occur, which may result in serious accidents.
    Check that the system will not be adversely affected if the size of the variable specified in the operand is less than the size in the operand definition before starting PLC operations.

⚠ **Caution** Confirm safety at the destination node before transferring a program to another node or changing contents of the I/O memory area. Doing either of these without confirming safety may result in injury.

⚠ **Caution** Execute online editing only after confirming that no adverse effects will be caused by extending the cycle time. Otherwise, the input signals may not be readable.

⚠ **Caution** Confirm safety sufficiently before monitoring power flow and present value status in the Ladder Section Window or when monitoring present values in the Watch Window. If force-set/reset or set/reset operations are inadvertently performed by pressing short-cut keys, the devices connected to Output Units may malfunction, regardless of the operating mode of the CPU Unit.

4 Application Precautions

Observe the following precautions when using the CX-Programmer.

- User programs cannot be uploaded to the CX-Programmer.
- Observe the following precautions before starting the CX-Programmer.
  - Exit all applications not directly related to the CX-Programmer. Particularly exit any software such as screen savers, virus checkers, E-mail or other communications software, and schedulers or other applications that start up periodically or automatically.
  - Disable sharing hard disks, printers, or other devices with other computers on any network.

  - With some notebook computers, the RS-232C port is allocated to a modem or an infrared line by default. Following the instructions in documentation for your computer and enable using the RS-232C port as a normal serial port.
  - With some notebook computers, the default settings for saving energy do not supply the rated power to the RS-232C port. There may be both Windows settings for saving energy, as well as setting for specific computer utilities and the BIOS. Following the instructions in documentation for your computer, disable all energy saving settings.
- Do not turn OFF the power supply to the PLC or disconnect the connecting cable while the CX-Programmer is online with the PLC. The computer may malfunction.
- Confirm that no adverse effects will occur in the system before attempting any of the following. Not doing so may result in an unexpected operation.
  - Changing the operating mode of the PLC.
  - Force-setting/force-resetting any bit in memory.
  - Changing the present value of any word or any set value in memory.
- Check the user program for proper execution before actually running it on the Unit. Not checking the program may result in an unexpected operation.
- When online editing is performed, the user program and parameter area data in CS1-H, CJ1-H, and CJ1M CPU Units is backed up in the built-in flash memory. The BKUP indicator will light on the front of the CPU Unit when the backup operation is in progress. Do not turn OFF the power supply to the CPU Unit when the BKUP indicator is lit. The data will not be backed up if power is turned OFF. To display the status of writing to flash memory on the CX-Programmer, select *Display dialog to show PLC Memory Backup Status* in the PLC properties and then select ***Windows - PLC Memory Backup Status*** from the View Menu.
- Programs including function blocks (ladder programming language or structured text (ST) language) can be downloaded or uploaded in the same way as standard programs that do not contain function blocks. Tasks including function blocks, however, cannot be downloaded in task units (uploading is possible).
- If a user program containing function blocks created on the CX-Programmer Ver. 5.0 or later is downloaded to a CPU Unit that does not support function blocks (CS/CJ-series CPU Units with unit version 2.0 or earlier), all instances will be treated as illegal commands and it will not be possible to edit or execute the user program.
- If the input variable data is not in boolean format, and numerical values only (e.g., 20) are input in the parameters, the actual value in the CIO Area address (e.g., 0020) will be passed. Therefore, be sure to include an &, #, or +, - prefix before inputting the numerical value.
- Addresses can be set in input parameters, but the address itself cannot be passed as an input variable. (Even if an address is set as an input parameter, the value passed to the function block will be that for the size of data in the input variable.) Therefore, an input variable cannot be used as the operand of the instruction in the function block when the operand specifies the first or last of multiple words. Use an internal variable with an AT setting. Alternatively, specify the first or last element in an internal array variable.

- Values are passed in a batch from the input parameters to the input variables before algorithm execution (not at the same time as the instructions in the algorithm are executed). Therefore, to pass the value from a parameter to an input variable when an instruction in the function block algorithm is executed, use an internal variable or external variable instead of an input variable. The same applies to the timing for writing values to the parameters from output variables.
- Always use internal variables with AT settings in the following cases.
  - The addresses allocated to Basic I/O Units, Special I/O Units, and CPU Bus Units cannot be registered to global symbols, and these variables cannot be specified as external variables (e.g., the data set for global variables may not be stable).
  - Use internal variables when Auxiliary Area bits other than those preregistered to external variables are registered to global symbols and these variables are not specified as external variables.
  - Use internal variables when specifying PLC addresses for another node on the network: For example, the first destination word at the remote node for SEND(090) and the first source word at the remote node for RECV(098).
  - Use internal variables when the first or last of multiple words is specified by an instruction operand and the operand cannot be specified as an internal array variable (e.g., the number of array elements cannot be specified).

SECTION 1
Introduction

This section introduces the function block functionality of the CX-Programmer and explains the features that are not contained in the non-function block version of CX-Programmer.

1-1 Introducing the Function Blocks 2
1-1-1 Overview and Features 2
1-1-2 Function Block Specifications 3
1-1-3 Files Created with CX-Programmer Ver. 5.0 4
1-1-4 CX-Programmer Ver. 5.0 Function Block Menus 5
1-2 Function Blocks 7
1-2-1 Outline 7
1-2-2 Advantages of Function Blocks 7
1-2-3 Function Block Structure 8
1-3 Variables 13
1-3-1 Introduction. 13
1-3-2 Variable Usage and Properties 13
1-3-3 Variable Properties 14
1-3-4 Variable Properties and Variable Usage 15
1-3-5 Internal Allocation of Variable Addresses 15
1-4 Converting Function Block Definitions to Library Files 16
1-5 Usage Procedures 17
1-5-1 Creating Function Blocks and Executing Instances 17
1-5-2 Reusing Function Blocks 18

1-1 Introducing the Function Blocks

1-1-1 Overview and Features

The CX-Programmer Ver. 5.0 is a Programming Device that can use standard IEC 61131-3 function blocks. The CX-Programmer Ver. 5.0 function block function is supported for CS/CJ-series CPU Units with unit version 3.0 or later and has the following features.

- User-defined processes can be converted to block format by using function blocks.
- Function block algorithms can be written in the ladder programming language or in the structured text (ST) language. (See note.)
  - When ladder programming is used, ladder programs created with non-CX-Programmer Ver. 4.0 or earlier can be reused by copying and pasting.
  - When ST language is used, it is easy to program mathematical processes that would be difficult to enter with ladder programming.

  Note The ST language is an advanced language for industrial control (primarily Programmable Logic Controllers) that is described in IEC 61131-3. The ST language supported by CX-Programmer conforms to the IEC 61131-3 standard.
- Function blocks can be created easily because variables do not have to be declared in text. They are registered in variable tables.
  A variable can be registered automatically when it is entered in a ladder or ST program. Registered variables can also be entered in ladder programs after they have been registered in the variable table.
- A single function block can be converted to a library function as a single file, making it easy to reuse function blocks for standard processing.
- A program check can be performed on a single function block to easily confirm the function block's reliability as a library function.
- Programs containing function blocks (ladder programming language or structured text (ST) language) can be downloaded or uploaded in the same way as standard programs that do not contain function blocks. Tasks containing function blocks, however, cannot be downloaded in task units (uploading is possible).
- One-dimensional array variables are supported, so data handling is easier for many applications.

Note The IEC 61131 standard was defined by the International Electrotechnical Commission (IEC) as an international programmable logic controller (PLC) standard. The standard is divided into 7 parts. Specifications related to PLC programming are defined in *Part 3 Textual Languages (IEC 61131-3).*

1-1-2 Function Block Specifications

For specifications that are not listed in the following table, refer to the *CX-Programmer Ver. 5.0 Operation Manual* (W437).

Item		Specifications
Model number		WS02-CXPC1-E-V50
Setup disk		CD-ROM
Compatible CPU Units		CS/CJ-series CS1-H, CJ1-H, and CJ1M CPU Units with unit version 3.0 or later are compatible. Device Type — CPU Type • CS1G-H — CS1G-CPU42H/43H/44H/45H • CS1H-H — CS1H-CPU63H/64H/65H/66H/67H • CJ1G-H — CJ1G-CPU42H/43H/44H/45H • CJ1H-H — CJ1H-CPU65H/66H/67H • CJ1M — CJ1M-CPU11/12/13/21/22/23 **Note** If a user program containing function blocks created on the CX-Programmer Ver. 5.0 or later is downloaded to a CPU Unit that does not support function blocks (CS/CJ-series CPU Units with unit version 2.0 or earlier), all instances will be treated as illegal commands and it will not be possible to edit or execute the user program.
		CS/CJ Series Function Restrictions • Instructions Not Supported in Function Block Definitions Block Program Instructions (BPRG and BEND), Subroutine Instructions (SBS, GSBS, RET, MCRO, and SBN), Jump Instructions (JMP, CJP, and CJPN), Step Ladder Instructions (STEP and SNXT), Immediate Refresh Instructions (!), I/O REFRESH (IORF), ONE-MS TIMER (TMHH) For details, refer to *2-3 Restrictions on Function Blocks*.
Compatible computers	Computer	IBM PC/AT or compatible
	CPU	133 MHz Pentium or faster with Windows 98, SE, or NT 4.0 (with service pack 6 or higher)
	OS	Microsoft Windows 95, 98, SE, Me, 2000, XP, or NT 4.0 (with service pack 6 or higher)
	Memory	64 Mbytes min. with Windows 98, SE, or NT 4.0 (with service pack 6 or higher) Refer to *Computer System Requirements* below for details.
	Hard disk space	100 Mbytes min. available disk space
	Monitor	SVGA (800 × 600 pixels) min. **Note** Use “small font” for the font size.
	CD-ROM drive	One CD-ROM drive min.
	COM port	One RS-232C port min.

Item			Specifications	
Functions not supported by CX-Programmer Ver. 4.0 or earlier.	Defining and creating function blocks	Number of function block definitions	CS1-H/CJ1-H CPU Units: • Suffix -CPU44H/45H/64H/65H/66H/67H: 1,024 max. per CPU Unit • Suffix -CPU42H/43H/63H: 128 max. per CPU Unit CJ1M CPU Units: • CJ1M-CPU11/12/13/21/22/23: 128 max. per CPU Unit	
		Function block names	64 characters max.	
		Variables	Variable names	30,000 characters max.
			Variable types	Inputs, Outputs, Internals, and Externals
			Number of I/O variables in function block definitions	64 max. (not including EN and ENO)
			Allocation of addresses used by variables	Automatic allocation (The allocation range can be set by the user.)
			Actual address specification	Supported
			Array specifications	Supported (one-dimensional arrays only)
		Language	Function blocks can be created in ladder programming language or structured text (ST, see note).	
	Creating instances	Number of instances	CS1-H/CJ1-H CPU Units: • Suffix -CPU44H/45H/64H/65H/66H/67H: 2,048 max. per CPU Unit • Suffix -CPU42H/43H/63H: 256 max. per CPU Unit CJ1M CPU Units: CJ1M-CPU11/12/13/21/22/23: 256 max. per CPU Unit	
		Instance names	30,000 characters max.	
	Storing function blocks as files	Project files	The project file (.cxp/cxt) Includes function block definitions and instances.	
		Program files	The file memory program file (*.obj) includes function block definitions and instances.	
		Function block library files	Each function block definition can be stored as a single file (.cxf) for reuse in other projects.	

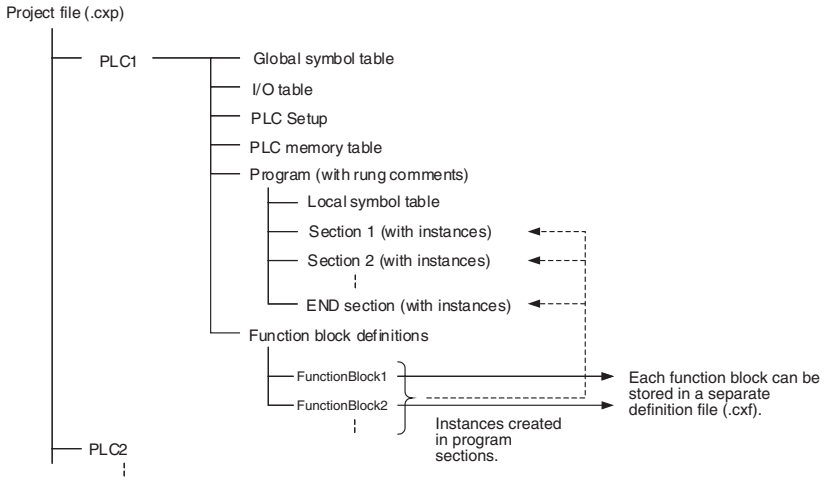
Note The structured text (ST language) conforms to the IEC 61131-3 standard, but CX-Programmer Ver. 5.0 supports only assignment statements, selection statements (CASE and IF statements), iteration statements (FOR, WHILE, REPEAT, and EXIT statements), RETURN statements, arithmetic operators, logical operators, comparison functions, numeric functions, and comments. For details, refer to *Appendix B Structured Text (ST Language) Specifications.*

1-1-3 Files Created with CX-Programmer Ver. 5.0

Project Files (*.cxp) and File Memory Program Files (*.obj)

Projects created using CX-Programmer that contain function block definitions and projects with instances are saved in the same standard project files (*.cxp) and file memory program files (*.obj).

The following diagram shows the contents of a project. The function block definitions are created at the same directory level as the program within the relevant PLC directory.



Function Block Library Files (*.cxf)

A function block definition created in a project with CX-Programmer Ver. 5.0 can be saved as a file (1 definition = 1 file), enabling definitions to be loaded into other programs and reused.

Project Text Files Containing Function Blocks (*.cxt)

Data equivalent to that in project files created with CX-Programmer Ver. 5.0 (*.cxp) can be saved as CXT text files (*.cxt).

1-1-4 CX-Programmer Ver. 5.0 Function Block Menus

The following tables list CX-Programmer Ver. 5.0 menus related to function blocks. For details on all menus, refer to the *CX-Programmer Ver. 5.0 Operation Manual* (W437).

Main Menu

Main menu	Submenu		Shortcut	Function
File	Function Block	Load Function Block from File	---	Reads the saved function block library files (*.cxf).
		Save Function Block to File	---	Saves the created function block definitions to a file (*[function block library file]**.cxf).
Edit	Update Function Block		---	When a function block definition's I/O variables have been changed after the instance was created, an error will be indicated by displaying the instance's left bus bar in red. This command updates the instance with the new information and clears the error.
Insert	Function Block Invocation		F	Creates an instance in the program (section) at the present cursor location.
	Function Block Parameter		P	When the cursor is located to the left of an input variable or the right of an output variable, sets the variable's input or output parameter.
PLC	Function Block Memory	Function Block Memory Allocation	---	Sets the range of addresses (function block instance areas) internally allocated to the selected instance's variables.
		Function Block Memory Statistics	---	Checks the status of the addresses internally allocated to the selected instance's variables.
		Function Block Instance Address	---	Checks the addresses internally allocated to each variable in the selected instance.
		Optimize Function Memory	---	Optimizes the allocation of addresses internally allocated to variables.

Main Popup Menus

Popup Menu for Function Block Definitions

Popup menu		Function
Insert Function Block	Ladder	Creates a function block definition with a ladder programming language algorithm.
	Structured Text	Creates a function block definition with an ST language algorithm.
	From file	Reads a function block definition from a function block library file (*.cxf).

Popup Menu for Inserted Function Blocks

Popup menu	Function
Open	Displays the contents of the selected function block definition on the right side of the window.
Save Function Block File	Saves the selected function block definition in a file.
Compile	Compiles the selected function block definition.

Popup Menu for Function Block Variable Tables

Popup menu		Function
Edit		Edits the variable.
Insert Variable		Adds a variable to the last line.
Insert Variable	Above	Inserts the variable above the current cursor position.
	Below	Inserts the variable below the current cursor position.
Cut		Cuts the variable.
Copy		Copies the variable.
Paste		Pastes the variable.
Find		Searches for the variable. Variable names, variable comments, or all (text strings) can be searched.
Replace		Replaces the variable.
Delete		Deletes the variable.
Rename		Changes only the name of the variable.

Popup Menu for Instances

Popup menu		Function
Edit		Changes the instance name.
Update Invocation		When a function block definition's I/O variables have been changed after the instance was created, an error will be indicated by displaying the instance's left bus bar in red. This command updates the instance with the new information and clears the error.
Go To	Function Block Definition	Displays the selected instance's function block definition on the right side of the window.

Shortcut Keys

F Key: Pasting Function Block Definitions in Program

Move the cursor to the position at which to create the copied function block instance in the Ladder Section Window, and click the **F** Key. This operation is the same as selecting *Insert - Function Block Invocation.*

P Key: Inputting Parameters

Position the cursor at the left of the input variable, or at the right of the output variable and click the **P** Key. This operation is the same as selecting *Insert - Function Block Parameter.*

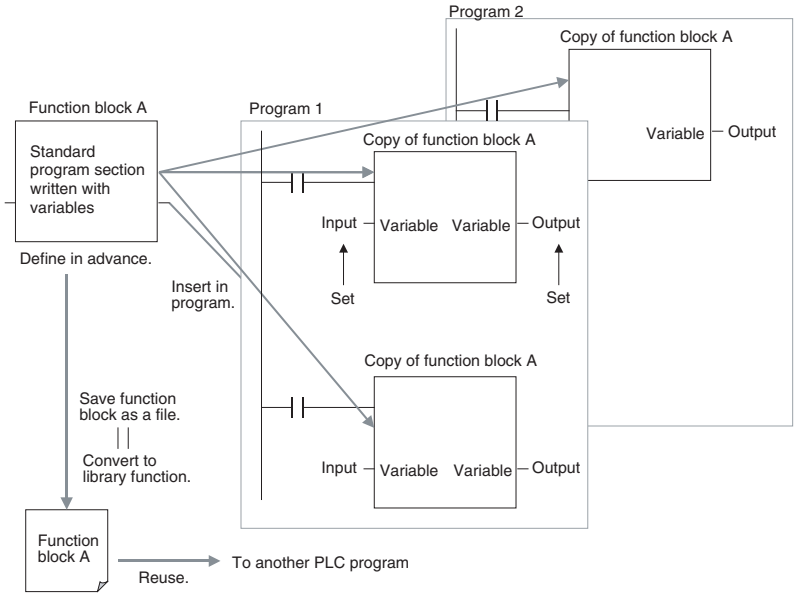
1-2 Function Blocks

1-2-1 Outline

A function block is a basic program element containing a standard processing function that has been defined in advance. Once the function block has been defined, the user just has to insert the function block in the program and set the I/O in order to use the function.

As a standard processing function, a function block does not contain actual addresses, but variables. The user sets addresses or constants in those variables. These address or constants are called parameters. The addresses used by the variables themselves are allocated automatically by the CX-Programmer for each program.

With the CX-Programmer, a single function block can be saved as a single file and reused in other PLC programs, so standard processing functions can be made into libraries.



1-2-2 Advantages of Function Blocks

Function blocks allow complex programming units to be reused easily. Once standard programming is created in a function block and saved in a file, it can be reused just by placing the function block in a program and setting the parameters for the function block's I/O. The ability to reuse existing function blocks will save significant time when creating/debugging programs, reduce coding errors, and make the program easier to understand.

Structured Programming

Structured programs created with function blocks have better design quality and require less development time.

Easy-to-read "Black Box" Design

The I/O operands are displayed as variable names in the program, so the program is like a "black box" when entering or reading the program and no extra time is wasted trying to understand the internal algorithm.

Use One Function Block for Multiple Processes

Many different processes can be created easily from a single function block by using the parameters in the standard process as input variables (such as timer SVs, control constants, speed settings, and travel distances).

Reduce Coding Errors

Coding mistakes can be reduced because blocks that have already been debugged can be reused.

Data Protection

The variables in the function block cannot be accessed directly from the outside, so the data can be protected. (Data cannot be changed unintentionally.)

Improved Reusability with Variable Programming

The function block's I/O is entered as variables, so it isn't necessary to change data addresses in a block when reusing it.

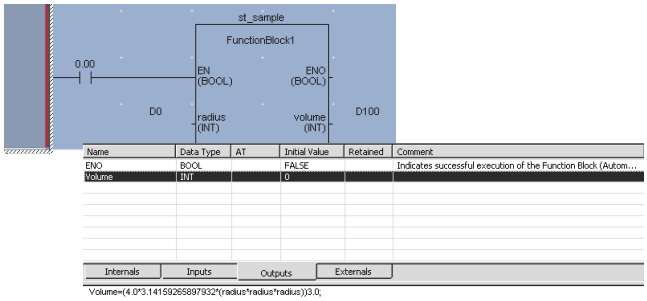
Creating Libraries

Processes that are independent and reusable (such as processes for individual steps, machinery, equipment, or control systems) can be saved as function block definitions and converted to library functions.

The function blocks are created with variable names that are not tied to actual addresses, so new programs can be developed easily just by reading the definitions from the file and placing them in a new program.

Compatible with Multiple Languages

Mathematical expressions can be entered in structured text (ST) language.

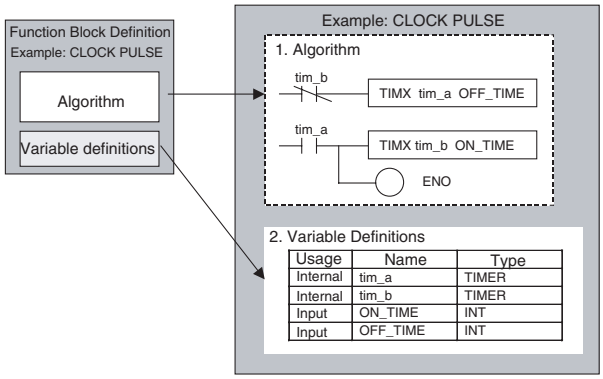


1-2-3 Function Block Structure

Function blocks consist of function block definitions that are created in advance and function block instances that are inserted in the program.

Function Block Definitions

Function block definitions are the programs contained in function blocks. Each function block definition contains the algorithm and variable definitions, as shown in the following diagram.



1. Algorithm

Standardized programming is written with variable names rather than real I/O memory addresses. In the CX-Programmer, algorithms can be written in either ladder programming or structured text.

2. Variable Definitions

The variable table lists each variable's usage (input, output, or internal) and properties (data type, etc.). For details, refer to *1-3 Variables*.

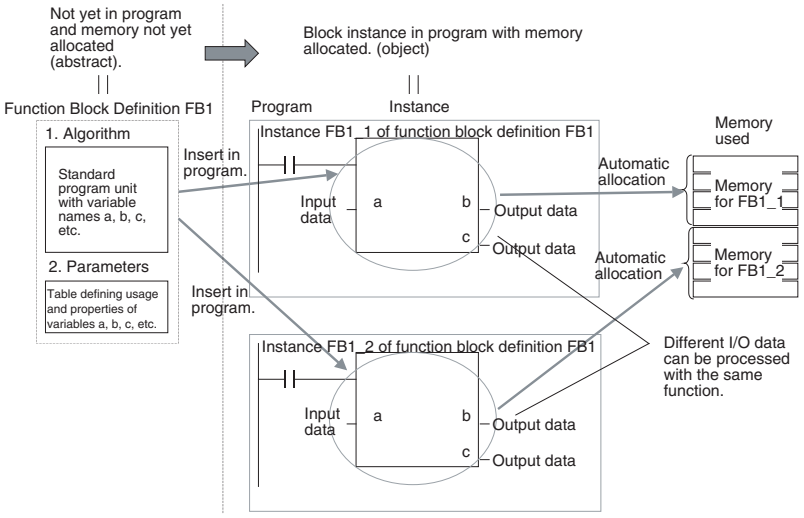
Number of Function Block Definitions

The maximum number of function block definitions that can be created for one CPU Unit is either 128 or 1,024 depending on the CPU Unit model.

Instances

To use an actual function block definition in a program, create a copy of the function block diagram and insert it in the program. Each function block definition that is inserted in the program is called an "instance" or "function block instance." Each instance is assigned an identifier called an "instance name."

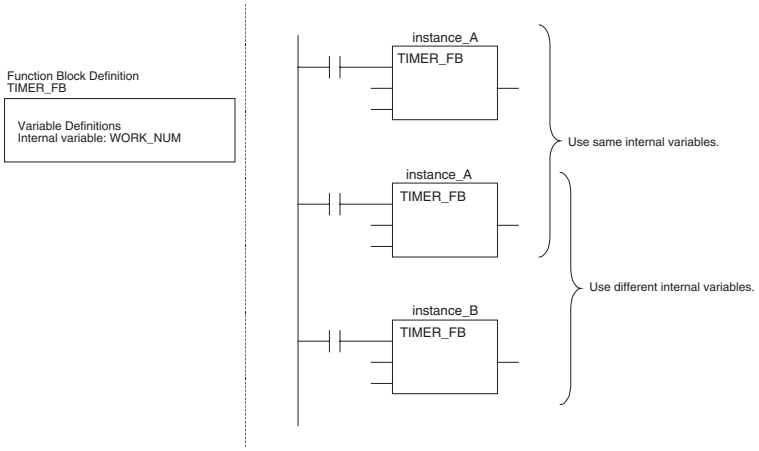
By generating instances, a single function block definition can be used to process different I/O data with the same function.



Note Instances are managed by names. More than one instance with the same name can also be inserted in the program. If two or more instances have the same name, they will use the same internal variables. Instances with different names will have different internal variables.

For example, consider multiple function blocks that use a timer as an internal variable. In this case all instances will have to be given different names. If more than one instance uses the same name, the same timer would be used in multiple locations, resulting in duplicated use of the timer.

If, however, internal variables are not used or they are used only temporarily and initialized the next time an instance is executed, the same instance name can be used to save memory.

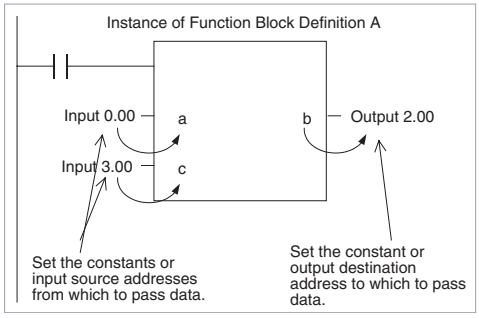


Number of Instances

Multiple instances can be created from a single function block definition. Up to either 256 or 2,048 instances can be created for a single CPU Unit depending on the CPU Unit model. The allowed number of instances is not related to the number of function block definitions and the number of tasks in which the instances are inserted.

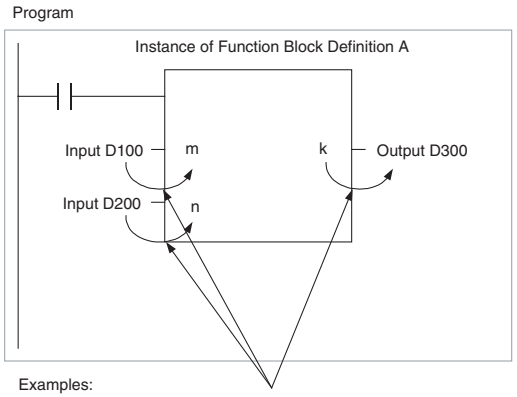
Parameters

Each time an instance is created, set the real I/O memory addresses or constants for I/O variables used to pass input data values to instances and obtain output data values from instances. These addresses and constants are called parameters.



Here, it is not the input source address itself, but the contents at the input address in the form and size specified by the variable data type that is passed to the function block. In a similar fashion, it is not the output destination address itself, but the contents for the output address in the form and size specified by the variable data type that is passed from the function block.

Even if an input source address (i.e., an input parameter) or an output destination address (i.e., an output parameter) is a word address, the data that is passed will be the data in the form and size specified by the variable data type starting from the specified word address.



Examples:

If m is type WORD, one word of data from D100 will be passed to the variable.

If n is type DWORD, two words of data from D200 and D201 will be passed to the variable.

If k is type LWORD, four words of data from the variable will be passed to the D300 to D303.

Note (1) Only addresses in the following areas can be used as parameters: CIO Area, Auxiliary Area, DM Area, EM Area (banks 0 to C), Holding Area, and Work Area.
The following cannot be used: Index and Data Registers (both direct and indirect specifications) and indirect addresses to the DM Area and EM Area (both in binary and BCD mode).

(2) Local and global symbols in the user program can also be specified as parameters. To do so, however, the data size of the local or global symbol must be the same as the data size of the function block variable.

(3) When an instance is executed, input values are passed from parameters to input variables before the algorithm is processed. Output values are passed from output variables to parameters just after processing the algorithm. If it is necessary to read or write a value within the execution cycle of the algorithm, do not pass the value to or from a parameter. Assign the value to an internal variable and use an AT setting (specified addresses).

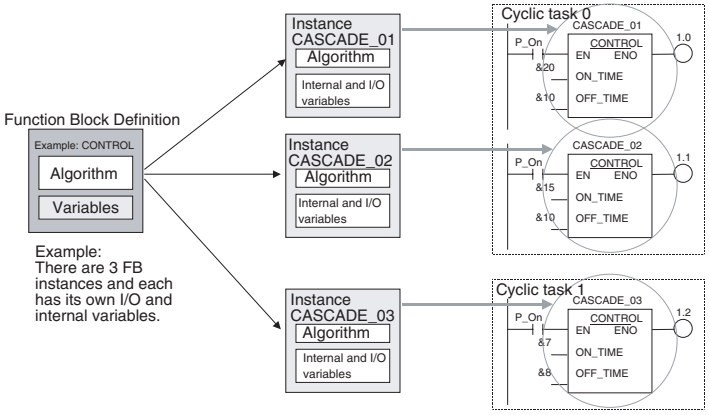
⚠ **Caution** If an address is specified in an input parameter, the values in the address are passed to the input variable. The actual address data itself cannot be passed.

⚠ **Caution** Parameters cannot be used to read or write values within the execution cycle of the algorithm. Use an internal variable with an AT setting (specified addresses). Alternatively, reference a global symbol as an external variable.

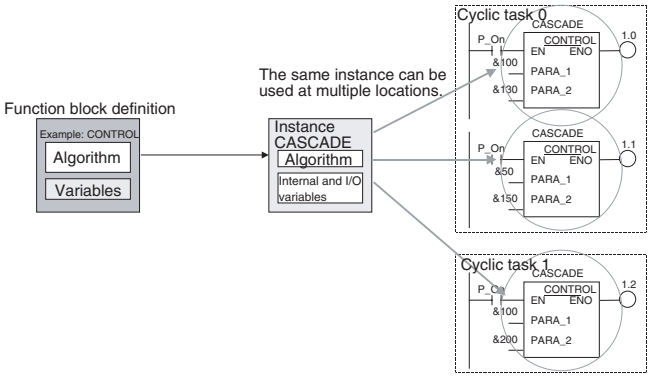
■ **Reference Information**

A variety of processes can be created easily from a single function block by using parameter-like elements (such as fixed values) as input variables and changing the values passed to the input variables for each instance.

Example: Creating 3 Instances from 1 Function Block Definition



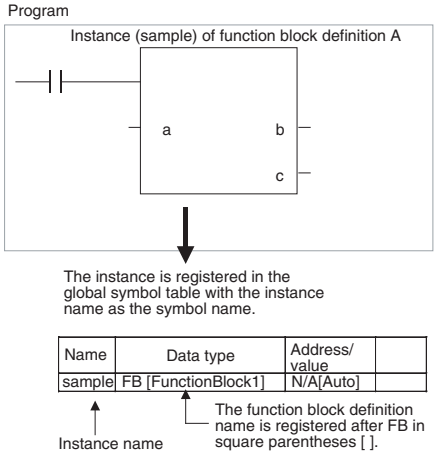
If internal variables are not used, if processing will not be affected, or if the internal variables are used in other locations, the same instance name can be used at multiple locations in the program.



Some precautions are required when using the same memory area. For example, if the same instance containing a timer instruction is used in more than one program location, the same timer number will be used causing coil duplication, and the timer will not function properly if both instructions are executed.

Registration of Instances

Each instance name is registered in the global symbol table as a file name.

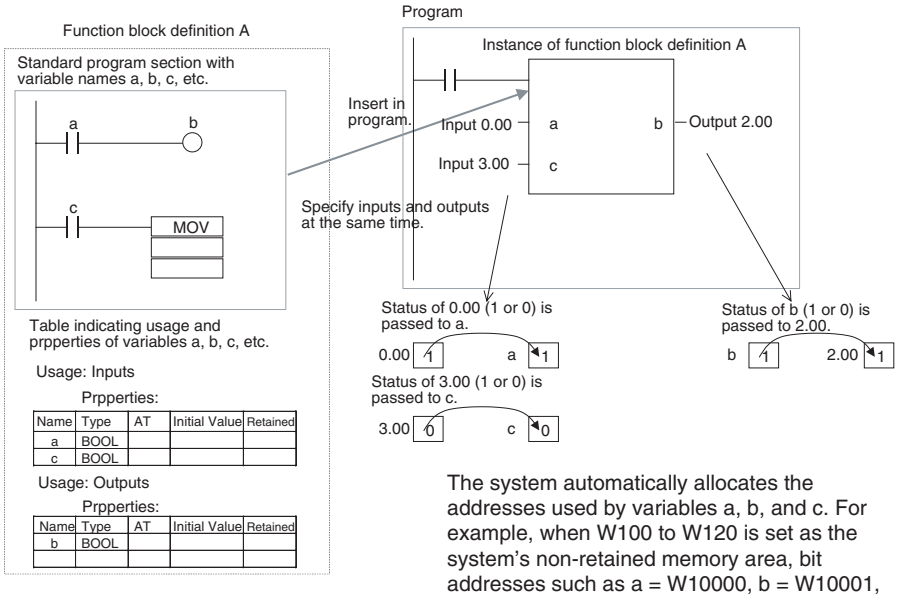


1-3 Variables

1-3-1 Introduction

In a function block, the addresses (see note) are not entered as real I/O memory addresses, they are all entered as variable names. Each time an instance is created, the actual addresses used by the variable are allocated automatically in the specified I/O memory areas by the CX-Programmer. Consequently, it isn't necessary for the user to know the real I/O memory addresses used in the function block, just as it isn't necessary to know the actual memory allocations in a computer. A function block differs from a subroutine in this respect, i.e., the function block uses variables and the addresses are like "black boxes."

Example:



The system automatically allocates the addresses used by variables a, b, and c. For example, when W100 to W120 is set as the system's non-retained memory area, bit addresses such as a = W10000, b = W10001, and c = W10002 will be allocated.

Note Constants are not registered as variables. Enter constants directly in instruction operands.

- Ladder programming language: Enter hexadecimal numerical values after the # and decimal values after the &.
- Structured text (ST language): Enter hexadecimal numerical values after 16# and enter decimal numerical values as is.

Exception: Enter directly or indirectly specified addresses for Index Registers IR0 to IR15 and Data Registers DR0 to DR15 directly into the instruction operand.

1-3-2 Variable Usage and Properties

Variable Usage

The following variable types (usages) are supported.

Internals: Internal variables are used only within an instance. They cannot be used pass data directly to or from I/O parameters.

Inputs: Input variables can input data from input parameters outside of the instance. The default input variable is an EN (Enable) variable, which passes input condition data.

Outputs: Output variables can output data to output parameters outside of the instance. The default output variable is an ENO (Enable Out) variable, which passes the instance's execution status.

Externals: External variables are either system-defined variables registered in advance with the CX-Programmer, such as the Condition Flags and some Auxiliary Area bits, or user-defined global symbols for use within instances.

For details on variable usage, refer to the section on *Variable Type (Usage)* under *Variable Definitions* in *2-1-2 Function Block Elements*.

The following table shows the number of variables that can be used and the kind of variable that is created by default for each of the variable usages.

1-3-3 Variable Properties

Variables have the following properties.

Variable Name

The variable name is used to identify the variable in the function block. It doesn't matter if the same name is used in other function blocks.

Note The variable name can be up to 30,000 characters long, but must not begin with a number. Also, the name cannot contain two underscore characters in a row. The character string cannot be the same as that of a an index register such as in IR0 to IR15. For details on other restrictions, refer to *Variable Definitions* in *2-1-2 Function Block Elements*.

Data Type

Select one of the following data types for the variable:

BOOL, INT, UINT, DINT, UDINT, LINT, ULINT, WORD, DWORD, LWORD, REAL, LREAL, TIMER, COUNTER

For details on variable data types, refer to *Variable Definitions* in *2-1-2 Function Block Elements*.

AT Settings (Allocation to an Actual Addresses)

It is possible to set a variable to a particular I/O memory address rather than having it allocated automatically by the system. To specify a particular address, the user can input the desired I/O memory address in this property. This property can be set for internal variables only. Even if a specific address is set, the variable name must still be used in the algorithm.

Refer to *Variable Definitions* in *2-1-2 Function Block Elements* for details on AT settings and *2-4-3 AT Settings for Internal Variables* for details on using AT settings.

Array Settings

A variable can be treated as a single array of data with the same properties. To convert a variable to an array, specify that it is an array and specify the maximum number of elements.

This property can be set for internal variables only. Only one-dimensional arrays are supported by the CX-Programmer Ver. 5.0.

- Setting Procedure
  Click the **Advanced** Button, select the *Array Variable* option, and input the maximum number of elements.
- When entering an array variable name in the algorithm in a function block definition, enter the array index number in square brackets after the variable number.

For details on array settings, refer to *Variable Definitions* in *2-1-2 Function Block Elements*.

Initial Value

This is the initial value set in a variable before the instance is executed for the first time. Afterwards, the value may be changed as the instance is executed.

For example, set a boolean (BOOL) variable (bit) to either 1 (TRUE) or 0 (FALSE). Set a WORD variable to a value between 0 and 65,535 (between 0000 and FFFF hex).

If an initial value is not set, the variable will be set to 0. For example, a boolean variable would be 0 (FALSE) and a WORD variable would be 0000 hex.

Retain

Select the *Retain Option* if you want a variable's data to be retained when the PLC is turned ON again and when the PLC starts operating.

- Setting Procedure
  Select the *Retain Option*.

1-3-4 Variable Properties and Variable Usage

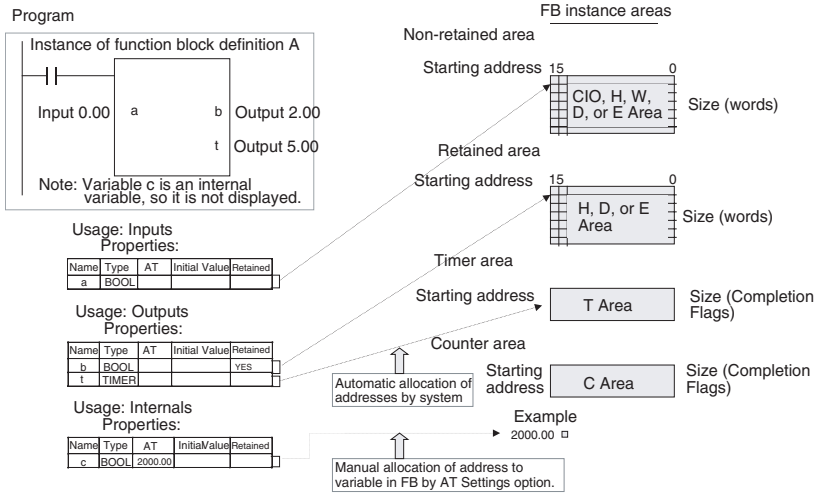
The following table shows which properties must be set, can be set, and cannot be set, based on the variable usage.

Property	Variable usage		
	Internals	Inputs	Outputs
Name	Must be set.	Must be set.	Must be set.
Data Type	Must be set.	Must be set.	Must be set.
AT (specified address)	Can be set.	Cannot be set.	Cannot be set.
Initial Value	Can be set.	Can be set. (See note.)	Can be set.
Retained	Can be set.	Can be set. (See note.)	Can be set.

Note Inputs can be set as initial values, but the value of the actual input parameter will be given priority.

1-3-5 Internal Allocation of Variable Addresses

When an instance is created from a function block definition, the CX-Programmer internally allocates addresses to the variables. Addresses are allocated to all of the variables registered in the function block definition except for variables that have been assigned actual addresses with the *AT Settings* property.



Setting Internal Allocation Areas for Variables

The user sets the function block instance areas in which addresses are allocated internally by the system. The variables are allocated automatically by the system to the appropriate instance area set by the user.

Setting Procedure

Select ***Function Block Memory - Function Block Memory Allocation*** from the *PLC* Menu. Set the areas in the Function Block Memory Allocation Dialog Box.

Function Block Instance Areas

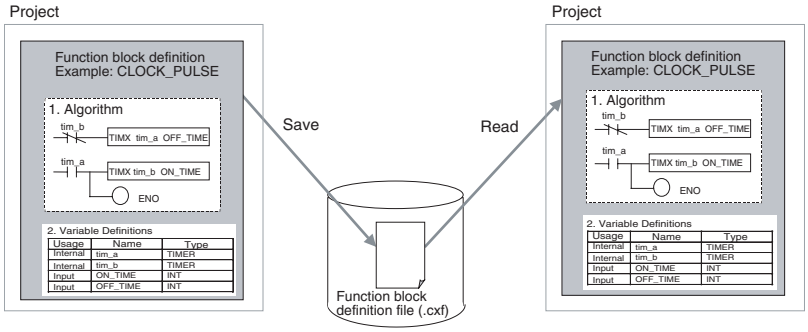
FB Instance Area	Default value			Applicable memory areas
	Start Address	End Address	Size	
Non Retain	H512	H1407	896	CIO, WR, HR, DM, EM
Retain	H1408	H1535	128	HR, DM, EM
Timers	T3072	T4095	1024	TIM
Counters	C3072	C4095	1024	CNT

Function Block Holding Area Words (H512 to H1535)

The Function Block Holding Area words are allocated from H512 to H1535. These words are different to the standard Holding Area used for programs (H000 to H511) and are used only for the function block instance area (internally allocated variable area). These words cannot be specified as instruction operands. They are displayed in red if input when a function block is not being created. Although the words can be input when creating a function block, an error will occur when the program is checked. If this area is specified not to be retained in the Function Block Memory Allocation Dialog Box, turn the power ON/OFF or clear the area without retaining the values when starting operation.

1-4 Converting Function Block Definitions to Library Files

A function block definition created using the CX-Programmer can be stored as a single file known as a function block definition file with filename extension *.cxf. These files can be reused in other projects (PLCs).



1-5 Usage Procedures

Once a function block definition has been created and an instance of the algorithm has been created, the instance is used by calling it when it is time to execute it. Also, the function block definition that was created can be saved in a file so that it can be reused in other projects (PLCs).

1-5-1 Creating Function Blocks and Executing Instances

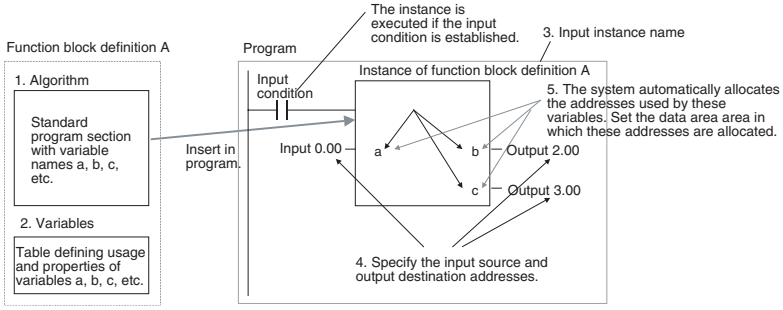
The following procedure outlines the steps required to create and execute a function block.

1,2,3... 1. First, create the function block definition including the algorithm and variable definitions in ladder program or ST language. Alternatively, insert a function block library file that has been prepared in advance.

Note (a) Create the algorithm entirely with variable names.

(b) When entering the algorithm in ladder programming language, project files created with versions of CX-Programmer earlier than Ver. 5.0 can be reused by reading the project file into the CX-Programmer Ver. 5.0 and copying and pasting useful parts.

2. When creating the program, insert copies of the completed function block definition. This step creates instances of the function block.
3. Enter an instance name for each instance.
4. Set the variables' input source addresses and/or constants and output destination addresses and/or constants as the parameters to pass data for each instance.
5. Select the created instance, select ***Function Block Memory - Function Block Memory Allocation*** from the *PLC* Menu, and set the internal data area for each type of variable.
6. Transfer the program to the CPU Unit.
7. Start program execution in the CPU Unit and the instance will be called and executed if their input conditions are ON.

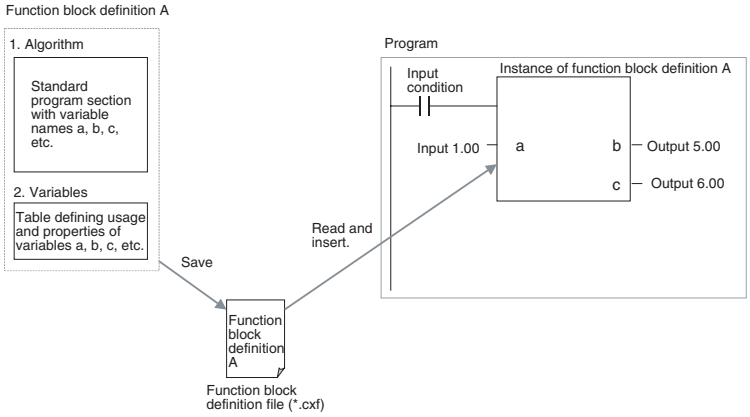


1-5-2 Reusing Function Blocks

Use the following procedure to save a function block definition as a file and use it in a program for another PLCs.

1,2,3...

1. Select the function block that you want to save and save it as a function block definition file (*.cxf).
2. Open the other PLC's project and open/read the function block definition file (*.cxf) that was saved.
3. Insert the function block definition in the program when creating the new program.



Note In the CX-Programmer Ver. 5.0, each function block definition can be compiled and checked as a program. We recommend compiling to perform a program check on each function block definition file before saving or reusing the file.

SECTION 2
Specifications

This section provides specifications for reference when using function blocks, including specifications on function blocks, instances, and compatible PLCs, as well as usage precautions and guidelines.

2-1 Function Block Specifications 21
2-1-1 Function Block Specifications 21
2-1-2 Function Block Elements 21
2-2 Instance Specifications 30
2-2-1 Composition of an Instance 30
2-2-2 Parameter Specifications. 34
2-2-3 Operating Specifications. 35
2-3 Restrictions on Function Blocks 37
2-4 Function Block Applications Guidelines 42
2-4-1 Deciding on Variable Data Types 42
2-4-2 Determining Variable Types (Inputs, Outputs, Externals, and Internals) 42
2-4-3 AT Settings for Internal Variables. 43
2-4-4 Array Settings for Internal Variables 44
2-4-5 Specifying Addresses Allocated to Special I/O Units 45
2-4-6 Using Index Registers. 46
2-5 Precautions for Instructions with Operands Specifying the First or Last of Multiple Words 49
2-6 Instruction Support and Operand Restrictions 52
2-6-1 Sequence Input Instructions 53
2-6-2 Sequence Output Instructions. 55
2-6-3 Sequence Control Instructions 56
2-6-4 Timer and Counter Instructions 57
2-6-5 Comparison Instructions. 60
2-6-6 Data Movement Instructions. 62
2-6-7 Data Shift Instructions 64
2-6-8 Increment/Decrement Instructions 67
2-6-9 Symbol Math Instructions. 67
2-6-10 Conversion Instructions 72
2-6-11 Logic Instructions. 74
2-6-12 Special Math Instructions. 76
2-6-13 Floating-point Math Instructions 76
2-6-14 Double-precision Floating-point Instructions. 80
2-6-15 Table Data Processing Instructions. 82
2-6-16 Data Control Instructions 85
2-6-17 Subroutine Instructions. 86
2-6-18 Interrupt Control Instructions. 87
2-6-19 High-speed Counter and Pulse Output Instructions (CJ1M-CPU21/22/23 Only) 88

2-6-20 Step Instructions 89
2-6-21 Basic I/O Unit Instructions 89
2-6-22 Serial Communications Instructions 91
2-6-23 Network Instructions 93
2-6-24 File Memory Instructions 95
2-6-25 Display Instructions. 95
2-6-26 Clock Instructions 96
2-6-27 Debugging Instructions 97
2-6-28 Failure Diagnosis Instructions. 98
2-6-29 Other Instructions 98
2-6-30 Block Programming Instructions 99
2-6-31 Text String Processing Instructions. 101
2-6-32 Task Control Instructions 103
2-6-33 Model Conversion Instructions 103
2-6-34 Special Instructions for Function Blocks 104
2-7 CPU Unit Function Block Specifications 104
2-7-1 Specifications 104
2-7-2 Operation of Timer Instructions 107
2-8 Number of Function Block Program Steps and Instance Execution Time . . . 108
2-8-1 Number of Function Block Program Steps (CPU Units with Unit Version 3.0 or Later) 108
2-8-2 Function Block Instance Execution Time (CPU Units with Unit Version 3.0 or Later) 109

2-1 Function Block Specifications

2-1-1 Function Block Specifications

Item	Description
Number of function block definitions	CS1-H/CJ1-H CPU Units: • Suffix -CPU44H/45H/64H/65H/66H/67H: 1,024 max. per CPU Unit • Suffix -CPU42H/43H/63H: 128 max. per CPU Unit CJ1M CPU Units: • CJ1M-CPU11/12/13/21/22/23: 128 max. per CPU Unit
Number of instances	CS1-H/CJ1-H CPU Units: • Suffix -CPU44H/45H/64H/65H/66H/67H: 2,048 max. per CPU Unit • Suffix -CPU42H/43H/63H: 256 max. per CPU Unit CJ1M CPU Units: CJ1M-CPU11/12/13/21/22/23: 256 max. per CPU Unit
Number of instance nesting levels	Nesting is not supported.
Number of I/O variables	64 variables max. per function block definition

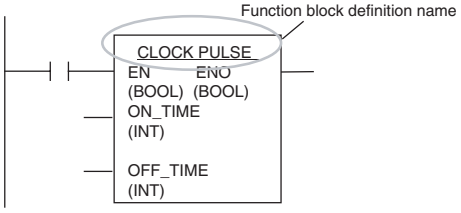
2-1-2 Function Block Elements

The following table shows the items that must be entered by the user when defining function blocks.

Item	Description
Function block definition name	The name of the function block definition
Language	The programming language used in the function block definition. Select ladder programming or structured text
Variable definitions	Variable settings, such as operands and return values, required when the function block is executed • Type (usage) of the variable • Name of the variable • Data type of the variable • Initial value of the variable
Algorithm	Enter the programming logic in ladder or structured text. • Enter the programming logic using variables. • Input constants directly without registering in variables.
Comment	Function blocks can have comments.

Function Block Definition Name

Each function block definition has a name. The names can be up to 64 characters long and there are no prohibited characters. The default function block name is FunctionBlock□, where □ is a serial number.



Language

Select either ladder programming language or structured text (ST language).

For details refer to *Appendix B Structured Text (ST Language) Specifications*.

Variable Definitions

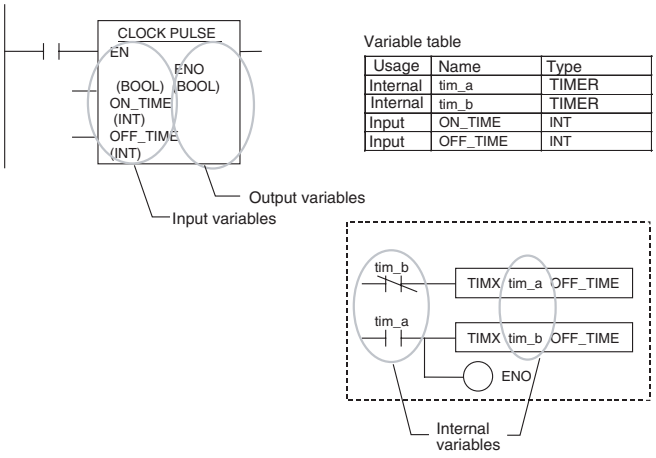
Define the operands and variables used in the function block definition.

Variable Names

- Variable names can be up to 30,000 characters long.
- Variables name cannot contain spaces or any of the following characters:
  ! “ # $ % & ‘ ( ) = - ~ ^ \ | ‘ @ { [+ ; * : }] < , > . ? /
- Variable names cannot start with a number (0 to 9).
- Variable names cannot contain two underscore characters in a row.
- The following characters cannot be used to indicate addresses in I/O memory.

  A, W, H (or HR), D (or DM), E (or EM), T (or TIM), C (or CNT) followed by the numeric value (word address)

Variable Notation



Variable table

Usage	Name	Type
Internal	tim_a	TIMER
Internal	tim_b	TIMER
Input	ON_TIME	INT
Input	OFF_TIME	INT

Variable Type (Usage)

Item	Variable type			
	Inputs	**Outputs**	**Internals**	**Externals**
Definition	Operands to the instance	Return values from the instance	Variables used only within instance	Global symbols registered as variables beforehand with the CX-Programmer or user-defined global symbols.
Status of value at next execution	The value is not passed on to the next execution.	The value is passed on to the next execution.	The value is passed on to the next execution.	The value is not passed on to the next execution.
Display	Displayed on the left side of the instance.	Displayed on the right side of the instance.	Not displayed.	Not displayed.
Number allowed	64 max. per function block (excluding EN)	64 max. per function block (excluding ENO)	Unlimited	Unlimited
AT setting	No	No	Supported	No
Array setting	No	No	Supported	No

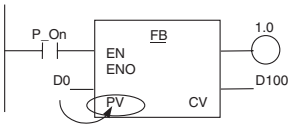
Item	Variable type			
	Inputs	**Outputs**	**Internals**	**Externals**
Retain setting	No	Supported	Supported	No
Variables created by default	EN (Enable): Receives an input condition.	ENO (Enable Output): Outputs the function block's execution status.	None	Pre-defined symbols registered in advance as variables in the CX-Programmer, such as Condition Flags and some Auxiliary Area bits.

Note For details on Externals, refer to *Appendix C External Variables*.

■ Input Variables

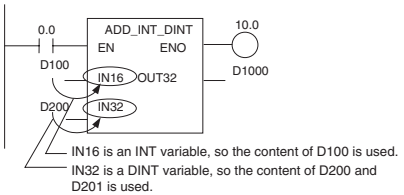
Input variables pass external operands to the instance. The input variables are displayed on the left side of the instance.

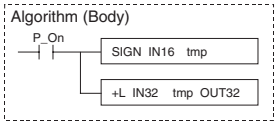
The value of the input source (data contained in the specified parameter just before the instance was called) will be passed to the input variable.



The value of the parameter specified as the input (value of D0) is passed to the instance's input variable (PV).

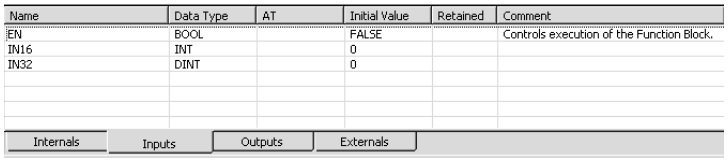
Example





Variable table

Usage	Name	Type
Internal	tmp	DINT
Input	EN	BOOL
Input	IN16	INT
Input	IN32	DINT
Output	ENO	BOOL
Output	OUT32	DINT



Name	Data Type	AT	Initial Value	Retained	Comment
EN	BOOL		FALSE		Controls execution of the Function Block.
IN16	INT		0		
IN32	DINT		0		

Internals | Inputs | Outputs | Externals

Note

1. The same name cannot be assigned to an input variable and output variable. If it is necessary to have the same variable as an input variable and output variable, register the variables with different names and transfer the value of the input variable to the output variable in the function block with an instruction such as MOV.
2. When the instance is executed, input values are passed from parameters to input variables before the algorithm is processed. Consequently, values cannot be read from parameters to input variables within the algorithm. If it is necessary to read a value within the execution cycle of the algorithm, do not pass the value from a parameter. Assign the value to an internal variable and use an AT setting (specified addresses). Alternatively, reference the global symbol as external variables.

Initial Value

Initial values can be set for input variables, but the value of the input parameter will be enabled (the input parameter value will be set when the parameter for input variable EN goes ON and the instance is executed).

Note The input parameter setting cannot be omitted when using the CX-Programmer.

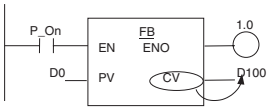
EN (Enable) Variable

When an input variable is created, the default input variable is the EN variable. The instance will be executed when the parameter for input variable EN is ON.

■ Output Variables

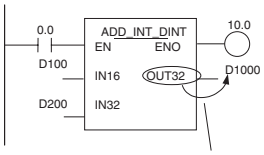
Output variables pass return values from the instance to external applications. The output variables are displayed on the right side of the instance.

After the instance is executed, the value of the output variable is passed to the specified parameter.

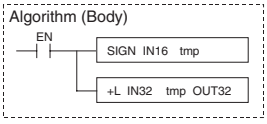


The value of the output variable (CV) is passed to the parameter specified as the output destination, which is D100 in this case.

Example

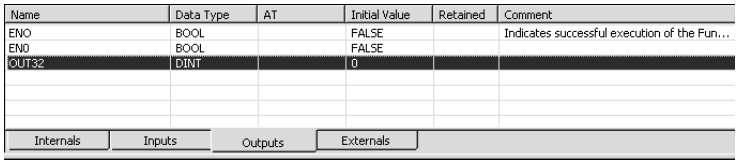


OUT32 is a DINT variable, so the variable's value is passed to D1000 and D1001.



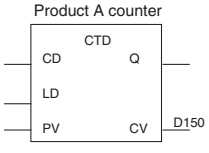
Variable table

Usage	Name	Data type
Internal	tmp	DINT
Input	EN	BOOL
Input	IN16	INT
Input	IN32	DINT
Output	ENO	BOOL
Output	OUT32	DINT



Like internal variables, the values of output variables are retained until the next time the instance is executed (i.e., when EN turns OFF, the value of the output variable is retained).

Example:
In the following example, the value of output variable CV will be retained until the next time the instance is executed.



Note

1. The same name cannot be assigned to an input variable and output variable. If it is necessary to have the same variable as an input variable and output variable, register the variables with different names and transfer the value of the input variable to the output variable in the function block with an instruction such as MOV.

2. When the instance is executed, output variables are passed to the corresponding parameters after the algorithm is processed. Consequently, values cannot be written from output variables to parameters within the algorithm. If it is necessary to write a value within the execution cycle of the algorithm, do not write the value to a parameter. Assign the value to an internal variable and use an AT setting (specified addresses).

Initial Value

An initial value can be set for an output variable that is not being retained, i.e., when the Retain Option is not selected. An initial value cannot be set for an output variable if the Retain Option is selected.
The initial value will not be written to the output variable if the IOM Hold Bit (A50012) is ON.

Auxiliary Area control bit		Initial value
IOM Hold Bit (A50012)	ON	The initial value will not be set.

ENO (Enable Output) Variable

The ENO variable is created as the default output variable. The ENO output variable will be turned ON when the instance is called. The user can change this value. The ENO output variable can be used as a flag to check whether or not instance execution has been completed normally.

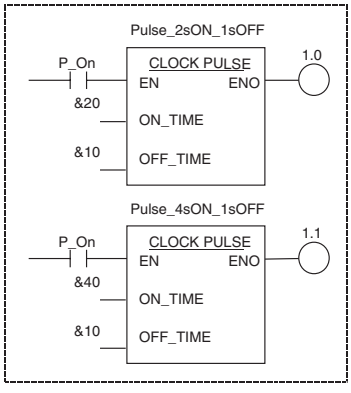
■ Internal Variables

Internal variables are used within an instance. These variables are internal to each instance. They cannot be referenced from outside of the instance and are not displayed in the instance.

The values of internal variables are retained until the next time the instance is executed (i.e., when EN turns OFF, the value of the internal variable is retained). Consequently, even if instances of the same function block definition are executed with the same I/O parameters, the result will not necessarily be the same.

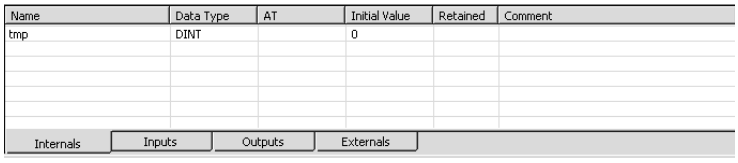
Example:

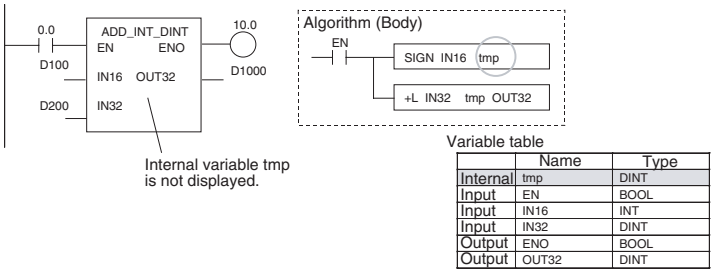
The internal variable tim_a in instance Pulse_2sON_1sOFF is different from internal variable tim_a in instance Pulse_4sON_1sOFF, so the instances cannot reference and will not affect each other's tim_a value.



Variable table

Usage	Name	Data type
Internal	tim_a	TIMER
Internal	tim_b	TIMER
Input	ON_TIME	INT
Input	OFF_TIME	INT





Variable table

	Name	Type
Internal	tmp	DINT
Input	EN	BOOL
Input	IN16	INT
Input	IN32	DINT
Output	ENO	BOOL
Output	OUT32	DINT

Retain Data through Power Interruptions and Start of Operation

Internal variables retain the value from the last time that the instance was called. In addition, the Retain Option can be selected so that an internal variable will also retains its value when the power is interrupted or operation starts (the mode is switched from PROGRAM to RUN or MONITOR mode).

When the Retain Option is selected, the value of the variable is retained when the power is interrupted or operation starts unless the CPU Unit does not have a backup battery. If the CPU Unit does not have a good battery, the value will be unstable.

Variables	Condition	Status
Variables set to *Retain*	Start of operation	Retained
	Power ON	Retained

When the Retain Option is not selected, the value of the variable will not be held when the power is interrupted or operation starts. Even variables not set to be retained, however, can be held at the start of operation by turning ON the IOM Hold Bit (A50012) and can be held during power interruptions by setting the PLC Setup, as shown in the following table.

Variables	Condition	IOM Hold Bit (A50012) setting		
		OFF	ON	
			IOM Hold Bit Status at Startup (PLC Setup) selected	IOM Hold Bit Status at Startup (PLC Setup) not selected
Variables not set to *Retain*	Start of operation	Not retained	Retained	Retained
	Power ON	Not retained	Retained	Not retained

Note The IOM Hold Bit (A50012) is supported for compatibility with previous models. To hold the values of variables in function blocks, however, use the *Retain Option* and not the IOM Hold Bit.

Initial Value

An initial value can be set for an internal variable that is not being retained (i.e., when the Retain Option not selected). An initial value cannot be set for an internal variable if the Retain Option is selected.
Internal variables that are not being retained will be initialized to 0.
The initial value will not be written to the internal variable if the IOM Hold Bit (A50012) is ON.

Auxiliary Area control bit		Initial value
IOM Hold Bit (A50012)	ON	The initial value will not be set.
	OFF	The initial value will be set.

■ **External Variables**

External variables are either system-defined variables that have been registered in CX-Programmer before hand, or variables that externally reference user-defined variables in the global symbol table.

- For details on system-defined variables, refer to *Appendix C External Variables.*
- To reference user-defined variables in the global symbol table, the variables must be registered in the global symbol table using the same variable name and data type as the external variable.

Variable Properties

Variable Name

The variable name is used to identify the variable in the function block. The name can be up to 30,000 characters long. The same name can be used in other function blocks.

Note A variable name must be input for variables, even ones with AT settings (specified address).

Data Type

Any of the following types may be used.

Data type	Content	Size	Inputs	Outputs	Internals
BOOL	Bit data	1 bit	OK	OK	OK
INT	Integer	16 bits	OK	OK	OK
UNIT	Unsigned integer	16 bits	OK	OK	OK
DINT	Double integer	32 bits	OK	OK	OK
UDINT	Unsigned double integer	32 bits	OK	OK	OK
LINT	Long (4-word) integer	64 bits	OK	OK	OK
ULINT	Unsigned long (4-word) integer	64 bits	OK	OK	OK
WORD	16-bit data	16 bits	OK	OK	OK
DWORD	32-bit data	32 bits	OK	OK	OK
LWORD	64-bit data	64 bits	OK	OK	OK
REAL	Real number	32 bits	OK	OK	OK
LREAL	Long real number	64 bits	OK	OK	OK
TIMER	Timer (See note 1.)	Flag: 1 bit PV: 16 bits	Not supported	Not supported	OK
COUNTER	Counter (See note 2.)	Flag: 1 bit PV: 16 bits	Not supported	Not supported	OK

Note

(1) The TIMER data type is used to enter variables for timer numbers (0 to 4095) in the operands for TIMER instructions (TIM, TIMH, etc.). When this variable is used in another instruction, the Timer Completion Flag (1 bit) or the timer present value (16 bits) is specified (depending on the instruction operand). The TIMER data type cannot be used in structured text function blocks.

(2) The COUNTER data type is used to enter variables for counter numbers (0 to 4095) in the operands for COUNTER instructions (CNT, CNTR, etc.). When this variable is used in another instruction, the Counter Completion Flag (1 bit) or the counter present value (16 bits) is specified (depending on the instruction operand). The COUNTER data type cannot be used in structured text function blocks.

AT Settings (Allocation to Actual Addresses)

With internal variables, it is possible to set the variable to a particular I/O memory address rather than having it allocated automatically by the system. To specify a particular address, the user can input the desired I/O memory address in this property. It is still necessary to use variable name in programming even if a particular address is specified.

Note (1) The AT property can be set for internal variables only.

(2) AT settings can be used only with the CIO (Core I/O Area), A (Auxiliary Area), D (Data Memory Area), E (Extended Memory Area, H (Holding Relay Area), W (Internal Relay Area).
The AT property cannot be set in the following memory areas:

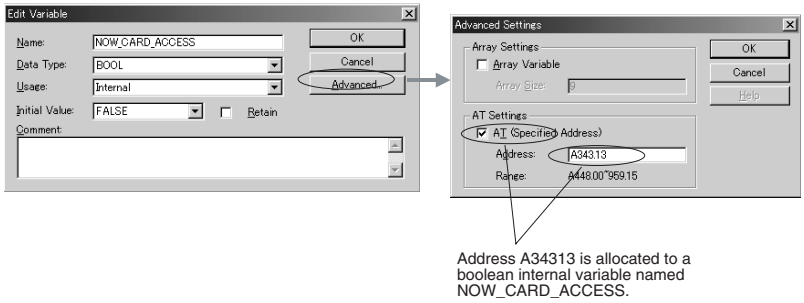
- Index Register and Data Register Areas (directly/indirectly specified)
- Indirectly specified DM/EM (: binary mode, *:BCD mode)

(3) AT settings can be used for the following allocations.

- Addresses for Basic I/O Units, CPU Bus Units, or Special I/O Units
- Auxiliary Area bits not registered as external variables in advance
- PLC addresses for other nodes in the network
- Instruction operands specifying the beginning word (or end word) of multiple words.

Example:
If the READ DATA FILE instruction (FREAD) is being used in the function block definition and it is necessary to check the File Memory Operation Flag (A34313), use an internal variable and specify the flag's address in the AT setting.

Register an internal variable, select the AT setting option, and specify A34313 as the address. The status of the File Memory Operation Flag can be checked through this internal variable.



When the AT setting is used, the function block loses its flexibility. This function should thus be used only when necessary.

Array Setting

With internal variables, a variable can be defined as an array.

Note Only one-dimensional arrays are supported by the CX-Programmer.

With the array setting, a large number of variables with the same properties can be used by registering just one variable.

- An array can have from 1 to 32,000 array elements.
- The array setting can be set for internal variables only.
- Any data type can be specified for an array variable, as long as it is an internal variable.
- When entering an array variable name in the algorithm of a function block definition, enter the array index number in square brackets after the variable name. The following three methods can be used to specify the index. (In this case the array variable is a[].)
  - Directly with numbers (for ladder or ST language programming)
    Example: a[2]
  - With a variable (for ladder or ST language programming)
    Example: a[n], where n is a variable

Note INT, DINT, LINT, UINT, UDINT, or ULINT can be used as the variable data type.

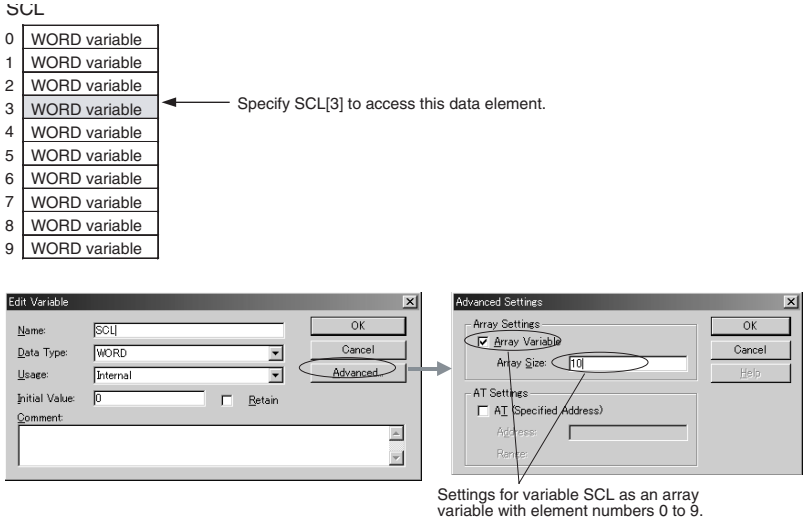
- With an equation (for ST language programming only)
  Example: a[b+c], where b and c are variables

Note Equations can contain only arithmetic operators (+, –, *, and /).

An array is a collection of data elements that are the same type of data. Each array element is specified with the same variable name and a unique index. (The index indicates the location of the element in the array.)

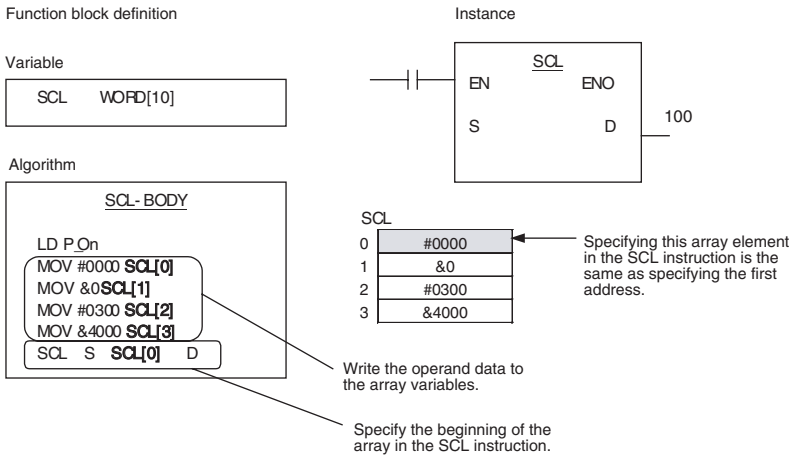
A one-dimensional array is an array with just one index number.

Example: When an internal variable named SCL is set as an array variable with 10 elements, the following 10 variables can be used:
SCL[0], SCL[1], SCL[2], SCL[3], SCL[4], SCL[5], SCL[6], SCL[7], SCL[8], and SCL[9]



Note Use an internal array variable when specifying the first or last of multiple words in an instruction operand to enable reusing the function block if an internal variable with a AT property cannot be set for the operand and an external variable cannot be set. Prepare an internal array variable with the number of elements for the required size, and after setting the data in each array element, specify the first or last element in the array variable for the operand.

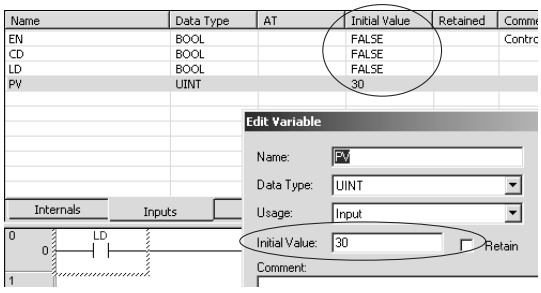
Example:



Note For details, refer to *2-5 Precautions for Instructions with Operands Specifying the First or Last of Multiple Words*.

Initial Values

When an instance is executed the first time, initial values can be set for input variables, internal variables, and output variables. For details, refer to *Initial Value* under the preceding descriptions of input variables, internal variables, and output variables.



Retaining Data through Power Interruptions and Start of Operation

The values of internal variables can be retained through power interruptions and the start of operation. When the Retain Option is selected, the variable will be allocated to a region of memory that is retained when the power is interrupted and PLC operation starts.

Algorithm

Enter the logic programming using the registered variables.

Operand Input Restrictions

Addresses cannot be directly input into instruction operands within function blocks. Addresses that are directly input will be treated as variable names.

Note Exception: Input directly or indirectly specified addresses for Index Registers IR0 to IR15 and Data Registers DR0 to DR15 directly into the instruction operand. Do not input variables.

Input constants directly into instruction operands.

- Ladder programming language: Enter hexadecimal numerical values after the # and decimal values after the &.
- Structured text (ST language): Enter hexadecimal numerical values after 16# and enter decimal numerical values as is.

Comment

A comment of up to 30,000 characters long can be entered.

2-2 Instance Specifications

2-2-1 Composition of an Instance

The following table lists the items that the user must set when registering an instance.

Item	Description
Instance name	Name of the instance
Language Variable definitions	The programming and variables are the same as in the function block definition.
Function block instance areas	The ranges of addresses used by the variables
Comments	A comment can be entered for each instance.

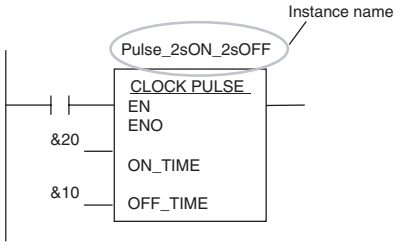
Instance Name

This is the name of the instance.

- Instance names can be up to 30,000 characters long.
- Instance names cannot contain spaces or any of the following characters:
  ! “ # $ % & ‘ ( ) = - ~ ^ \ | ‘ @ { [+ ; * : }] < , > . ? /
- Instance names cannot start with a number (0 to 9).

There are no other restrictions.

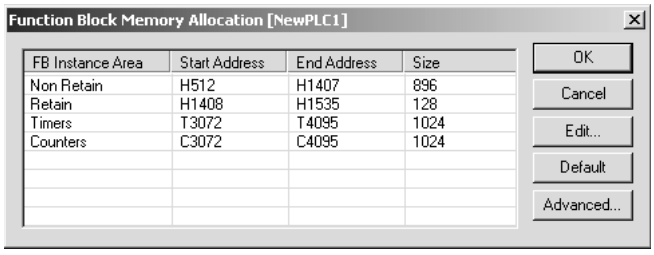
The instance name is displayed above the instance in the diagram.



Function Block Instance Areas

To use a function block, the system requires memory to store the instance’s internal variables and I/O variables. These areas are known as the function block instance areas and the user must specify the first addresses and sizes of these areas. The first addresses and area sizes can be specified in 1-word units.

When the CX-Programmer compiles the function, it will output an error if there are any instructions in the user program that access words in these areas.



FB Instance Area	Default value			Applicable memory areas
	Start Address	End Address	Size	
Non Retain	H512	H1407	896	CIO, WR, HR, DM, EM
Retain	H1408	H1535	128	HR, DM, EM
Timers	T3072	T4095	1024	TIM
Counters	C3072	C4095	1024	CNT

Function Block Instance Area Types

The following settings are made in the function block instance area:

Non-retained Areas

Item	Contents
Allocated variables	Variables for which the retain property for power OFF and operation start is set as non-retained (See note 1.)
Applicable areas	H (Function block Special Holding Area), I/O (CIO Area), H (Holding Area), W (Internal Relay Area), D (Data Memory Area) (see note 2), E (Extended Data Memory Area) (See notes 2 and 3.)
Setting unit	Set in words
Allocated words (default)	H512 to H140

Note (1) Except when the data type is set to TIMER or COUNTER.

(2) Bit data can be accessed even if the DM or EM Area is specified for the non-retained area or retained area.

(3) The same bank number cannot be specified as the current bank in the user program if the EM Area is specified for the non-retained area or retained area.

Retained Area

Item	Contents
Allocated variables	Variables for which the retain property for power OFF and operation start is set as retained (See note 1.)
Applicable areas	H (Function block Special Holding Area), H (Holding Area), D (Data Memory Area) (see note 1), E (Extended Data Memory Area) (See notes 2 and 3.)
Setting unit	Set in words
Allocated words (default)	H1408 to H1535

Note

(1) Except when the data type is set to TIMER or COUNTER.

(2) Bit data can be accessed even if the DM or EM Area is specified for the non-retained area or retained area.

(3) The same bank number cannot be specified as the current bank in the user program if the EM Area is specified for the non-retained area or retained area.

Timer Area

Item	Contents
Allocated variables	Variables with TIMER set as the data type.
Applicable areas	T (Timer Area) Timer Flag (1 bit) or timer PVs (16 bits)
Allocated words (default)	T3072 to T4095 Timer Flag (1 bit) or timer PVs (16 bits)

Counter Area

Item	Contents
Allocated variables	Variables with COUNTER set as the data type.
Applicable areas	C (Counter Area) Counter Flag (1 bit) or counter PVs (16 bits)
Allocated words (default)	C3072 to C4095 Counter Flag (1 bit) or counter PVs (16 bits)

Function Block Holding Area (H512 to H1535)

The default allocation of Function Block Holding Area words set as retained and non-retained words is H512 to H1535. These words are different to the standard Holding Area used for programs (H000 to H511), and are used only for the function block instance area (internally allocated variable area).

- These words cannot be specified in AT settings for internal variables.
- These words cannot be specified as instruction operands.
  - These words are displayed in red if they are input when a function block is not being created.
  - Although the words can be input when creating a function block, an error will occur when the program is checked.
- If this area is specified as non-retained, turn the power ON/OFF or clear the area without retaining the values when starting operation.

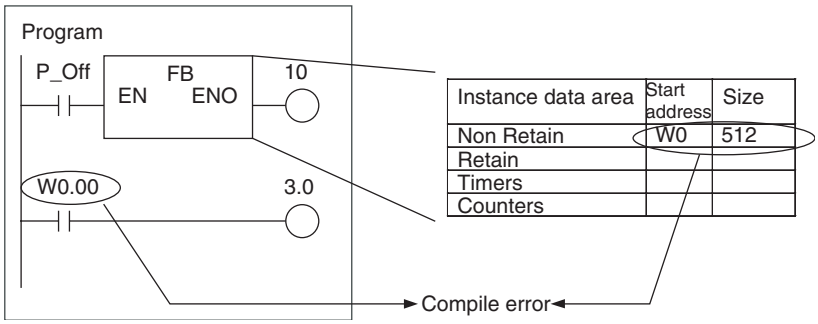
Note To prevent overlapping of instance area addresses with addresses used in the program, set H512 to H1535 (Function Block Holding Area words) for the non-retained area and retained area.

Accessing Function Block Instance Area from the User Program

If the user program contains an instruction to access the function block instance area, an error will be displayed in the Compile Tab of the Output Window of CX-Programmer if the following operations are attempted.

- Attempting to write during online editing (writing not possible)
- Executing program check (Selecting *Compile* from the Program Menu or *Compile All PLC Programs* from the PLC Menu)

Example: If W0 to W511 is specified as the non-retained area of the function block instance area and W0.00 is used in the ladder program, an error will occur when compiling and be displayed as “ERROR: [omitted]...- Address - W0.00 is reserved for Function Block use].



Note The allocations in the function block instance area for variables are automatically reallocated when a variable is added or deleted. A single instance requires addresses in sequence, however, so if addresses in sequence cannot be obtained, all variables will be allocated different addresses. As a result, unused areas will be created. If this occurs, execute the optimization operation to effectively use the allocated areas and remove the unused areas.

Comments

A comment of up to 30,000 characters long can be entered.

Creating Multiple Instances

Calling the Same Instance

A single instance can be called from multiple locations. In this case, the internal variables will be shared.

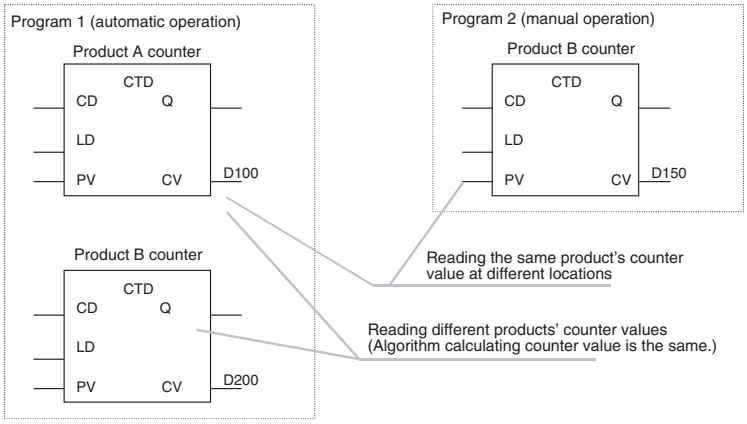
Making Multiple Instances

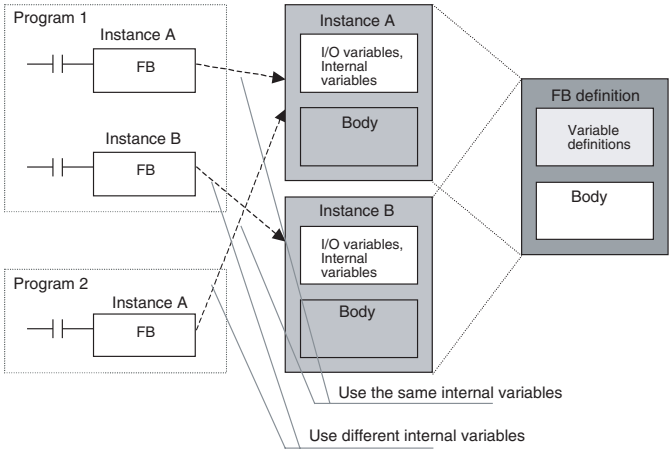
Multiple instances can be created from a single function block definition. In this case, the values of internal variables will be different in each instance.

Example: Counting Product A and Product B

Prepare a function block definition called Down Counter (CTD) and set up counters for product A and product B. There are two types of programs, one for automatic operation and another for manual operation. The user can switch to the appropriate mode of operation.

In this case, multiple instances will be created from a single function block. The same instance must be called from multiple locations.





2-2-2 Parameter Specifications

The data that can be set by the user in the input parameters and output parameters is as follows:

Item	Applicable data
Input parameters	Values (See note 1.), addresses, and program symbols (global symbols and local symbols) (See note 2.) **Note** The data that is passed to the input variable from the parameter is the actual value of the size of the input variable data. (An address itself will not be passed even if an address is set in the parameter.) **Note** Input parameters must be set. If even one input parameter has not been set, a fatal error will occur and the input parameters will not be transferred to the actual PLC.
Output parameters	Addresses, program symbols (global symbols, local symbols) (See note 2.)

Note (1) The following table shows the methods for inputting values in parameters.

Input variable data type	Contents	Size	Parameter value input method	Setting range
BOOL	Bit data	1 bit	P_Off, P_On	0 (FALSE), 1 (TRUE)
INT	Integer	16 bits	Positive value: & or + followed by integer Negative value: − followed by integer	−32,768 to 32,767
DINT	Double integer	32 bits		−2,147,483,648 to 2,147,483,647
LINT	Long (8-byte) integer	64 bits		−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Input variable data type	Contents	Size	Parameter value input method	Setting range
UINT	Unsigned integer	16 bits	Positive value: & or + followed by integer	&0 to 65,535
UDINT	Unsigned double integer	32 bits		&0 to 4,294,967,295
ULINT	Unsigned long (8-byte) integer	64 bits		&0 to 18,446,744,073,709,551,615
REAL	Real number	32 bits	Positive value: & or + followed by real number (with decimal point) Negative value: − followed by real number (with decimal point)	-3.402823×10^{38} to $-1.175494 \times 10^{-38}$, 0, 1.175494×10^{-38} to 3.402823×10^{38}
LREAL	Long real number	64 bits		$-1.79769313486232 \times 10^{308}$ to $-2.22507385850720 \times 10^{-308}$, 0, $2.22507385850720 \times 10^{-308}$, $1.79769313486232 \times 10^{308}$
WORD	16-bit data	16 bits	# followed by hexadecimal number (4 digits max.) & or + followed by decimal number	#0000 to FFFF or &0 to 65,535
DWORD	32-bit data	32 bits	# followed by hexadecimal number (8 digits max.) & or + followed by decimal number	#00000000 to FFFFFFFF or &0 to 4,294,967,295
LWORD	64-bit data	64 bits	# followed by hexadecimal number (16 digits max.) & or + followed by decimal number	#0000000000000000 to FFFFFFFFFFFFFFFF or &0 to 18,446,744,073,709,551,615

(2) The size of function block input variables and output variables must match the size of program symbols (global and local), as shown in the following table.

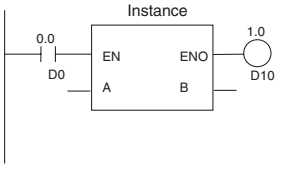
Size	Function block variable data type	Program symbol (global, local) data type
1 bit	BOOL	BOOL
16 bits	INT, UINT, WORD	INT, UINT, UINT BCD, WORD
32 bits	DINT, UDINT, REAL, DWORD	DINT, UDINT, UDINT BCD, REAL, DWORD
64 bits	LINT, ULINT, LREAL, LWORD	LINT, ULINT, ULINT BCD, LREAL, LWORD
More than 1 bit	Non-boolean	CHANNEL, NUMBER (see note)

Note The program symbol NUMBER can be set only in the input parameters. The value that is input must be within the size range for the function block variable data type.

2-2-3 Operating Specifications

Calling Instances

The user can call an instance from any location. The instance will be executed when the input to EN is ON.

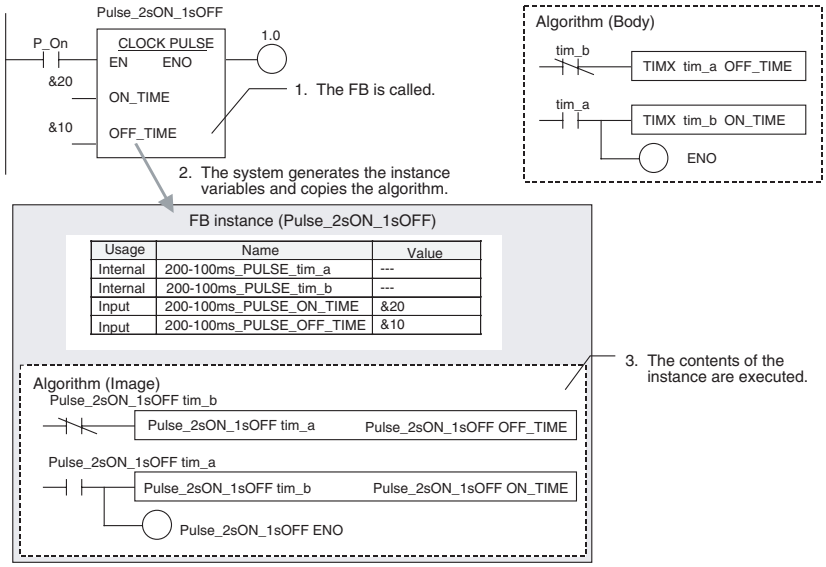


In this case, the input to EN is bit 0.0 at the left of the diagram.

- When the input to EN is ON, the instance is executed and the execution results are reflected in bit 1.0 and word D10.
- When the input to EN is OFF, the instance is not executed, bit 1.0 is turned OFF, and the content of D10 is not changed.

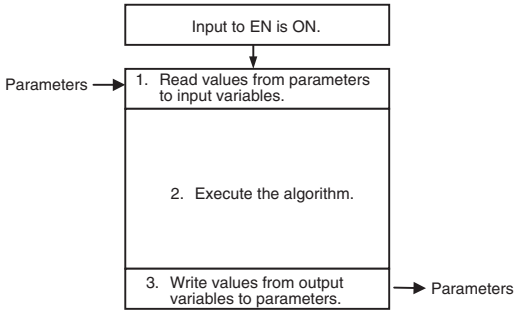
Operation when the Instance Is Executed

The system calls a function block when the input to the function block's EN input variable is ON. When the function block is called, the system generates the instance's variables and copies the algorithm registered in the function block. The instance is then executed.



The order of execution is as follows:

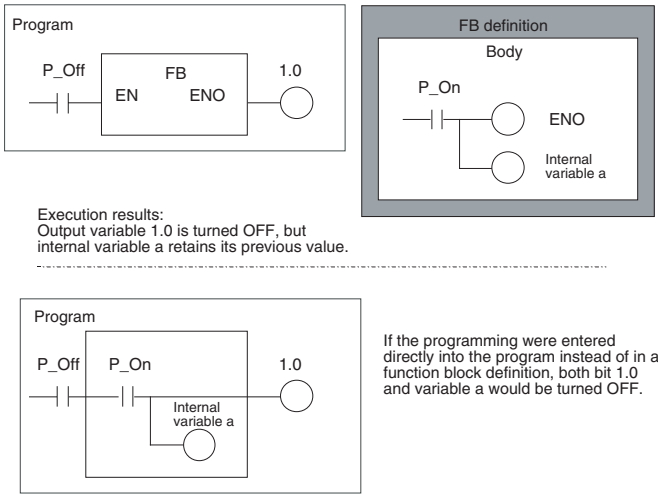
1. Read data from parameters to input variables.
2. Execute the algorithm.
3. Write data from output variables to parameters.



Data cannot be exchanged with parameters in the algorithm itself.
In addition, if an output variable is not changed by the execution of the algorithm, the output parameter will retain its previous value.

Operation when the Instance Is Not Executed

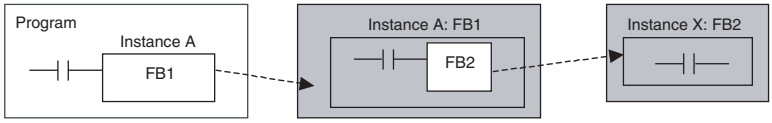
When the input to the function block's EN input variable is OFF, the function block is not called, so the internal variables of the instance do not change (values are retained). In the same way the output variables do not change when EN is OFF (values are retained).



⚠ Caution An instance will not be executed while its EN input variable is OFF, so Differentiation and Timer instructions will not be initialized while EN is OFF. If Differentiation or Timer instructions are being used, use the Always ON Flag (P_On) for the EN input condition and include the instruction's input condition within the function block definition.

Nesting

A function block cannot be called from another function block, i.e., nesting is not supported.



2-3 Restrictions on Function Blocks

Ladder Programming Restrictions

There are some restrictions on instructions used in ladder programs.

Instructions Prohibited in Function Block Definitions

The following instructions cannot be used in function block definitions. A compile error will occur if any of these instructions is used.

- Block Programming Instructions (All instructions, including BPRG and BEND)
- Subroutine Instructions (SBS, GSBS, RET, MCRO, SBN, GSBN, and GRET)
- Jump Instructions (JMP, CJP, CJPN, and JME)
- Step Instructions (STEP and SNXT)
- Immediate Refresh Instructions (!)
- I/O REFRESH Instruction (IORF)
- TMHH and TMHHX Instructions
- CV Address Conversion Instructions (FRMCV and TOCV)

- Instructions manipulating record positions (PUSH, FIFO, LIFO, SETR, and GETR)
- FAILURE POINT DETECTION Instruction (FPD)
- Move Timer/Counter PV to Register Instruction (MOVRW)

AT Setting Restrictions (Unsupported Data Areas)

Addresses in the following areas cannot be used for AT settings.

- Index Registers (neither indirect nor direct addressing is supported) and Data Registers

  Note Input the address directly, not the AT setting.
- Indirect addressing of DM or EM Area addresses (Neither binary-mode nor BCD-mode indirect addressing is supported.)

Direct Addressing of I/O Memory in Instruction Operands

- Addresses, not variables, can be directly input in Index Registers (both indirect and direct addressing) and Data Registers.
  The following values can be input in instruction operands:
  Direct addressing: IR0 to IR15; Indirect addressing: ,IR0 to ,IR15; Constant offset (example): +5,IR0; DR offset: DR0,IR0; Auto-increment: ,IR0++; Auto-decrement: --,IR0
- Direct addressing in instruction operands is not supported for any other areas in I/O memory.

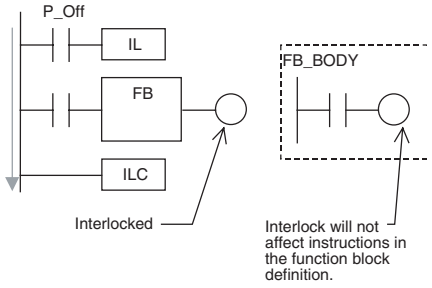
I/O Variable Restrictions (Unsupported Data Areas)

Addresses in the following data areas cannot be used as parameters for input and output variables.

- Index Registers (neither indirect nor direct addressing is supported) and Data Registers
- Indirect addressing of DM or EM Area addresses (Neither binary-mode nor BCD-mode indirect addressing is supported.)

Interlock Restrictions

When a function block is called from an interlocked program section, the contents of the function block definition will not be executed. The interlocked function block will behave just like an interlocked subroutine.

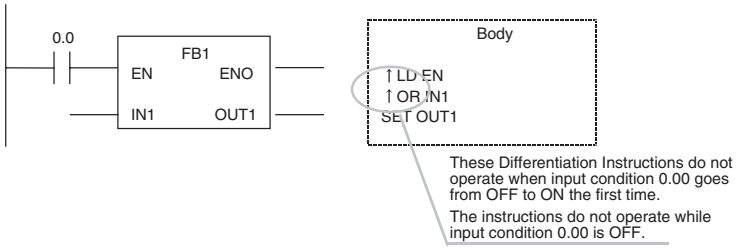


Differentiation Instructions in Function Block Definitions

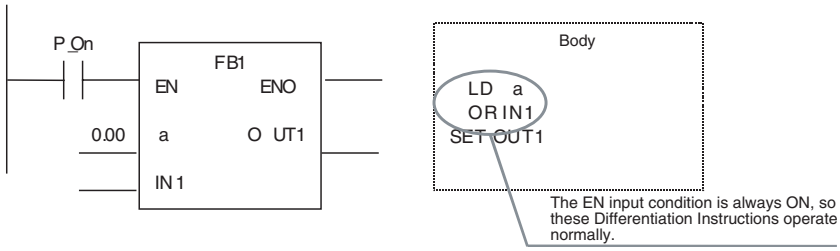
An instance will not be executed while its EN input variable is OFF, so the following precautions are essential when using a Differentiation Instruction in a function block definition. (Differentiation Instructions include DIFU, DIFD, and any instruction with an @ or % prefix.)

- As long as the instance's EN input variable is OFF, the execution condition will retain its previous status (the last status when the EN input variable was ON) and the Differentiation Instruction will not operate.
- When the instance's EN input variable goes ON, the present execution condition status will not be compared to the last cycle's status. The present execution condition will be compared to the last condition when the EN input variable was ON, so the Differentiation Instruction will not operate properly. (If the EN input variable remains ON, the Differentiation Instruction will operate properly when the next rising edge or falling edge occurs.)

Example:



If Differentiation Instructions are being used, always use the Always ON Flag (P_On) for the EN input condition and include the instruction's input condition within the function block definition.



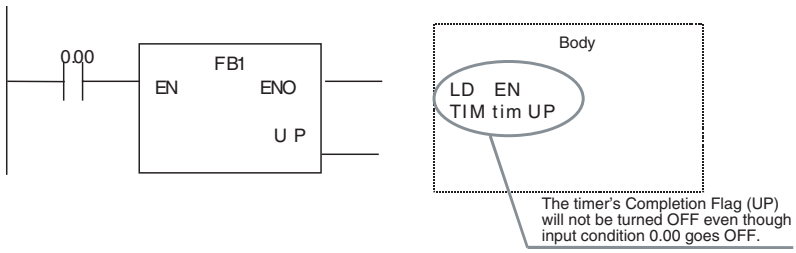
- Input a decimal numerical value after "#" when specifying the first operand of the following instructions.
  MILH(517), MILR(518), MILC(519), DIM(631), MSKS(690), MSKR(692), CLI(691), FAL(006), FALS(007), TKON(820), TKOF(821)

  Note "&" is not supported.
- CNR(545), CNRX(547) (RESET TIMER/COUNTER) instructions cannot be used to reset multiple timers and counters within a function block at the same time.
  Always specify the same variable for the first operand (timer/counter number 1) and second operand (timer/counter number 2). Different variables cannot be specified for the first and second operand.

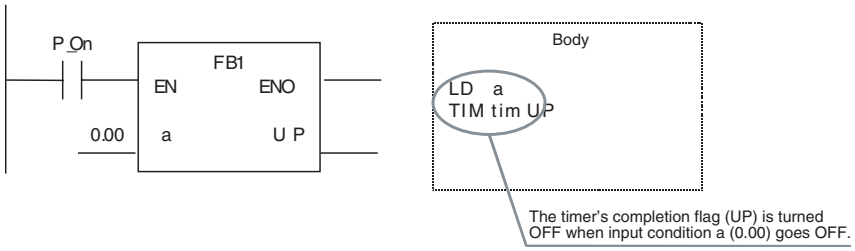
Timer Instructions in Function Block Definitions

An instance will not be executed while its EN input variable is OFF, so the following precautions are essential when using a Timer Instruction in a function block definition.

The Timer Instruction will not be initialized even though the instance's EN input variable goes OFF. Consequently, the timer's Completion Flag will not be turned OFF if the EN input variable goes OFF after the timer started operating.



If Timer Instructions are being used, always use the Always ON Flag (P_On) for the EN input condition and include the instruction's input condition within the function block definition.



- If the same instance containing a timer is used in multiple locations at the same time, the timer will be duplicated.

ST Programming Restrictions

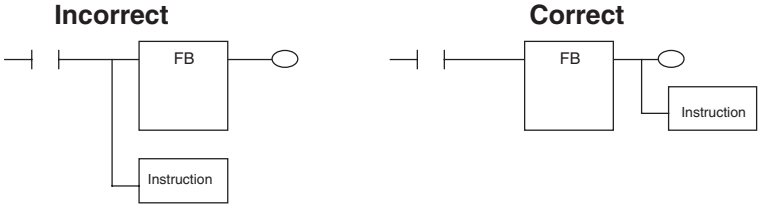
- Only the following statements and operators are supported.
  - Assignment statements
  - Selection statements (CASE and IF statements)
  - Iteration statements (FOR, WHILE, REPEAT, and EXIT statements)
  - RETURN statements
  - Arithmetic operators
  - Logical operators
  - Comparison operators
  - Numerical Functions
  - Arithmetic Functions
  - Comments
- The TIMER and COUNTER data types cannot be used.

For further details, refer to *Appendix B Structured Text (ST Language) Specifications.*

Program Structure Precautions

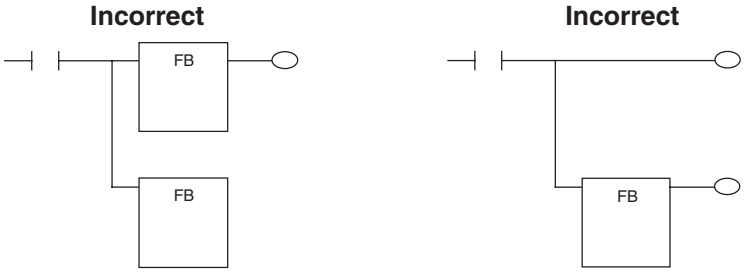
No Branches to the Left of the Instance

Branches are not allowed on the left side of the instance. Branches are allowed on the right side.



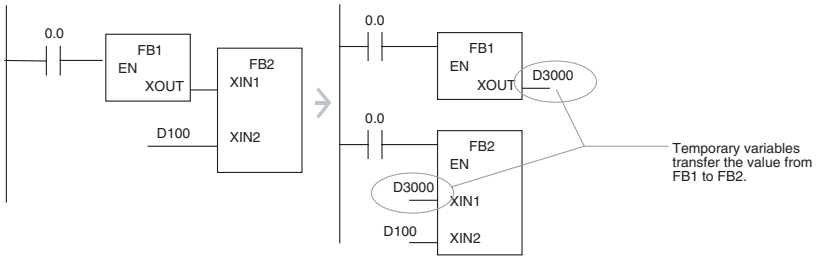
Only One Instance per Rung

A program rung cannot have more than one instance.



No Function Block Connections

A function block's input cannot be connected to another function block's output. In this case, a variable must be registered to transfer the execution status from the first function block's output to the second function blocks input.



Downloading in Task Units

Tasks including function blocks cannot be downloaded in task units, but uploading is possible.

Programming Console Displays

When a user program created with the CX-Programmer is downloaded to the CPU Unit and read by a Programming Console, the instances will all be displayed as question marks. (The instance names will not be displayed.)

Online Editing Restrictions

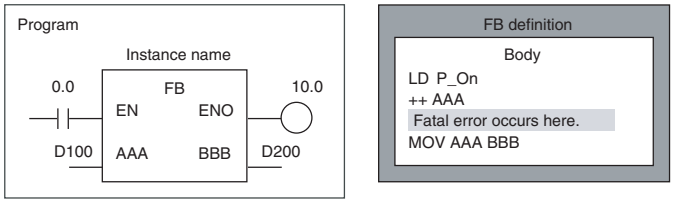
The following online editing operations cannot be performed on the user program in the CPU Unit.

- Changing or deleting function block definitions (variable table or algorithm)
- Inserting instances or changing instance names

Note The instance's I/O parameters can be changed, instances can be deleted, and instructions outside of an instance can be changed.

Error-related Restrictions

If a fatal error occurs in the CPU Unit while a function block definition is being executed, ladder program execution will stop at the point where the error occurred.



In this case, the MOV AAA BBB instruction will not be executed and output variable D200 will retain the same value that it had before the function block was executed.

Prohibiting Access to FB Instance Areas

To use a function block, the system requires memory areas to store the instance's internal variables and I/O variables.

Function block instance area	Initial value of start address	Initial value of size	Allowed data areas
Non-retained	H512	896	CIO, WR, HR, DM, EM
Retained	H1408	128	HR, DM, EM
Timer	T3072	1,024	TIM
Counter	C3072	1,024	CNT

If there is an instruction in the user program that accesses an address in an FB instance area, the CX-Programmer will output an error in the following cases.

- When a program check is performed by the user by selecting ***Program - Compile*** from the Program Menu or ***Compile All Programs*** from the *PLC* Menu.
- When attempting to write the program through online editing (writing is not possible).

2-4 Function Block Applications Guidelines

This section provides guidelines for using function blocks with the CX-Programmer.

2-4-1 Deciding on Variable Data Types

Integer Data Types (1, 2, or 4-word Data)

Use the following data types when handling single numbers in 1, 2, or 4-word units.

- INT and UINT
- DINT and DINT
- LINT and ULINT

Note Use signed integers if the numbers being used will fit in the range.

Word Data Types (1, 2, or 4-word Data)

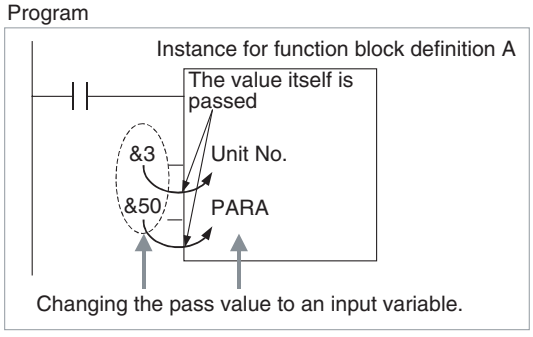
Use the following data types when handling groups of data (non-numeric data) in 1, 2, or 4-word units.

- WORD
- DWORD
- LWORD

2-4-2 Determining Variable Types (Inputs, Outputs, Externals, and Internals)

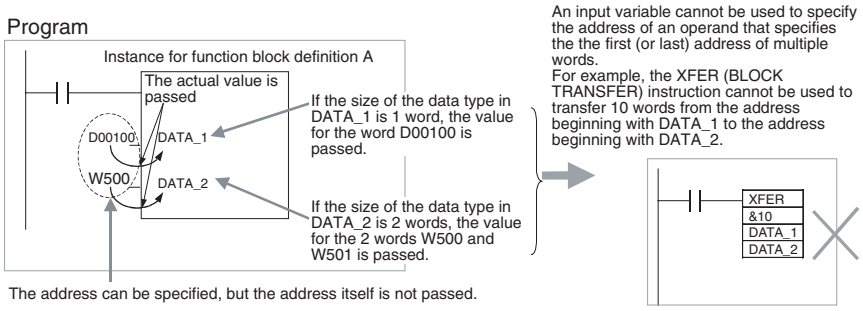
Using Input Variable to Change Passed Values

To paste a function block into the program and then change the value (not the address itself) to be passed to the function block for each instance, use an input variable.



The following two restrictions apply.

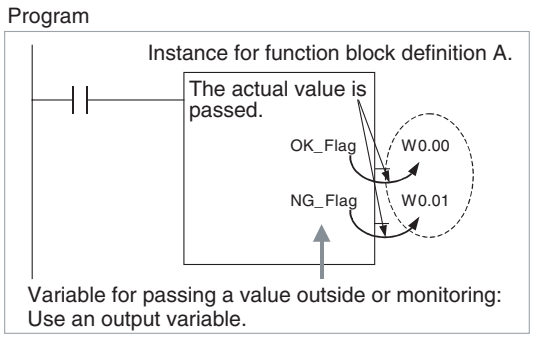
- An address can be set in an input parameter, but an address itself cannot be passed to an input variable (even if an address is set in the input parameter, the value for the size of the input variable data type is passed to the function block). Therefore, when the first or last of multiple words is specified in the instruction operand within the function block, an input variable cannot be used for the operand. Specify either to use internal variables with AT settings, specify the first or last element in an internal array variable, or use an external variable (as described in *2-4-4 Array Settings for Internal Variables*).



- Values are passed in a batch from the input parameters to the input variables before algorithm execution (not at the same time as the instruction in the algorithm is executed). Therefore, to pass the value from a parameter to an input variable when the instruction in the function block algorithm is executed, use an internal variable or external variable instead of an input variable.

Passing Values from or Monitoring Output Variables

To paste into the program and then pass values outside (the program) from the function block for each instance, or monitor values, use output variables.



The following restrictions apply.

- Values are passed from output variables to output parameters all at once after algorithm execution.

External Variables: Condition Flags, Clock Pulses, Auxiliary Area Bits, Global Symbols in Program

Condition Flags (e.g., Always ON Flag, Equals Flag), Clock Pulses (e.g., 1.0 second clock pulse bit), pre-registered Auxiliary Area Bits (e.g., First Cycle Flag), and global symbols used in the program are all external variables defined by the system.

Internal Variables: Internally Allocated Variables and Variables Requiring AT Settings

Variables that are not specified as Inputs, Outputs, or Externals are Internals. Internal variables include variables with internally allocated addresses, variables requiring addresses with AT settings (e.g., I/O allocation addresses, addresses specially allocated for Special I/O Units), or variables requiring array settings. For details on conditions requiring AT settings or array settings, refer to *2-4-3 AT Settings for Internal Variables*, and *2-4-4 Array Settings for Internal Variables*.

2-4-3 AT Settings for Internal Variables

Always specify AT settings for internal variables under the following conditions.

- When addresses allocated to Basic I/O Units, Special I/O Units, or CPU Bus Units are used and these addresses are registered to global symbols that cannot be specified as external variables (e.g., data set for global symbols is unstable).

Note The method for specifying Index Registers for Special I/O Unit allocation addresses requires AT settings to be specified for the first address of the allocation area. (For details, refer to *2-4-5 Specifying Addresses Allocated to Special I/O Units*.)

- When Auxiliary Area bits that are not pre-registered to external variables are used, and these bits are registered to global symbols that are not specified as external variables.
- When setting the first destination word at the remote node for SEND(090) and the first source word at the local node for RECV(098).
- When the instruction operand specifies the first or last of multiple words, and internal array variable cannot be specified for the operand (e.g., the number of array elements cannot be specified).

2-4-4 Array Settings for Internal Variables

Using Array Variables to Specify First or Last Word in Multiword Operands

When specifying the first or last of a range of words in an instruction operand (see note), the instruction operates according to the address after AT specification or internal allocation. (Therefore, the variable data type and number of elements for the variable are unrelated to the operation of the instruction.) Always specify a variable with an AT setting or an array variable with a number of elements that matches the data size to be processed by the instruction.

Note Some examples are the first source word or first destination word of the XFER(070) (BLOCK TRANSFER) instruction, the first source word for SEND(090), or control data for applicable instructions.

For details, refer to *2-5 Precautions for Instructions with Operands Specifying the First or Last of Multiple Words*. Use the following method to specify an array variable.

1,2,3... 1. Prepare an internal array variable with the required number of elements.

Note Make sure that the data size to be processed by the instruction is the same as the number of elements. For details on the data sizes processed by each instruction, refer to *2-6 Instruction Support and Operand Restrictions.*

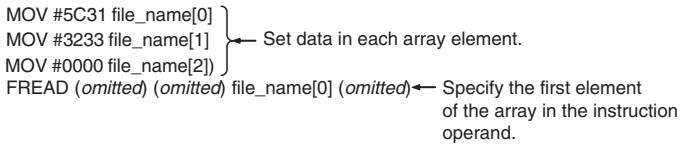
2. Set the data in each of the array elements using the MOV instruction in the function block definition.
3. Specify the first (or last) element of the array variable for the operand. This enables specification of the first (or last) address in a range of words.

Examples are provided below.

Handling a Single String of Data in Multiple Words

In this example, an array contains the directory and filename (operand S2) for an FREAD instruction.

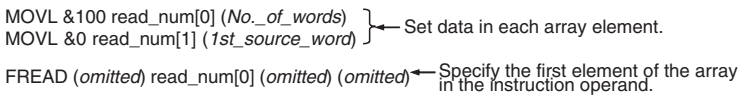
- Variable table
  Internal variable, data type = WORD, array setting with 10 elements, variable names = filename[0] to filename[9]
- Ladder programming



Handling Control Data in Multiple Words

In this example, an array contains the number of words and first source word (operand S1) for an FREAD instruction.

- Variable table
  Internal variable, data type = DINT, array setting with 3 elements, variable names = read_num[0] to read_num[9]
- Ladder programming



Handling a Block of Read Data in Multiple Words

The allowed amount of read data must be determined in advance and an array must be prepared that can handle the maximum amount of data. In this example, an array receives the FREAD instruction's read data (operand D).

- Variable table
  Internal variable, data type = WORD, array setting with 100 elements, variable names = read_data[0] to read_data[99]
- Ladder programming

Division Using Integer Array Variables (Ladder Programming Only)

A two-element array can be used to store the result from a ladder program's SIGNED BINARY DIVIDE (/) instruction. The result from the instruction is D (quotient) and D+1 (remainder). This method can be used to obtain the remainder from a division operation in ladder programming.

Note When ST language is used, it isn't necessary to use an array to receive the result of a division operation. Also, the remainder can't be calculated directly in ST language. The remainder must be calculated as follows:
Remainder = Dividend − (Divisor × Quotient)

2-4-5 Specifying Addresses Allocated to Special I/O Units

Use Index Registers IR0 to IR15 (indirectly specified constant offset) to specify addresses allocated to Special I/O Units based on the value passed for the unit number as an input parameter within the function block definition as shown in the following examples.

Note For details on using Index Registers in function blocks, refer to *2-4-6 Using Index Registers.*

Examples

Example 1: Specifying the CIO Area within a Function Block (Same for DM Area)

Special I/O Units

Variables: Use the unit number as an input variable, and specifying the first allocation address as an internal variable with the AT set to CIO 2000.

Programs: Use the following procedure.

1,2,3...

1. Multiply the unit number (input variable) by &10, and create the unit number offset (internal variable, DINT data type).
2. Use the MOVR(560) (MOVE TO REGISTER) instruction to store the real I/O memory address for the first allocation address (internal variable, AT = CIO 2000) in the Index Register (e.g., IR0).

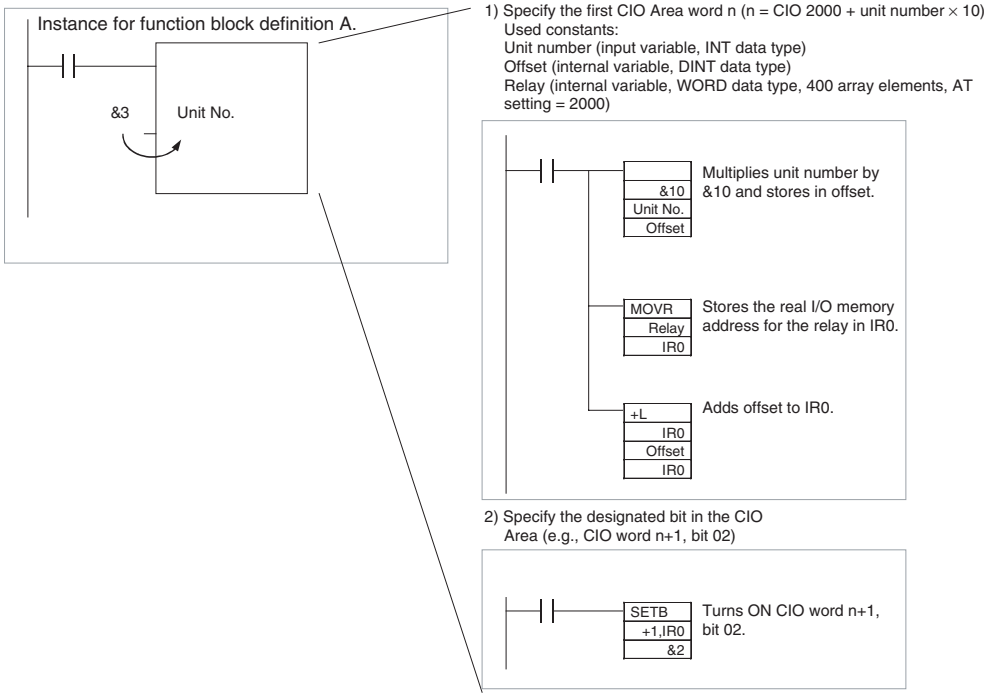
3. Add the unit number offset to the real I/O memory address within the Index Register (e.g., IR0).

Example 2: Specifying the Designated Bit in the CIO Area (e.g., CIO Word n+a, Bit b)

Programs: Use either of the following methods.

- Word addresses: Specify the constant offset of the Index Register using an indirect specification (e.g., +a,IR0).
- Bit addresses: Specify an instruction that can specify a bit address within a word (e.g., &b in second operand of SETB instruction when writing and TST instruction when reading).

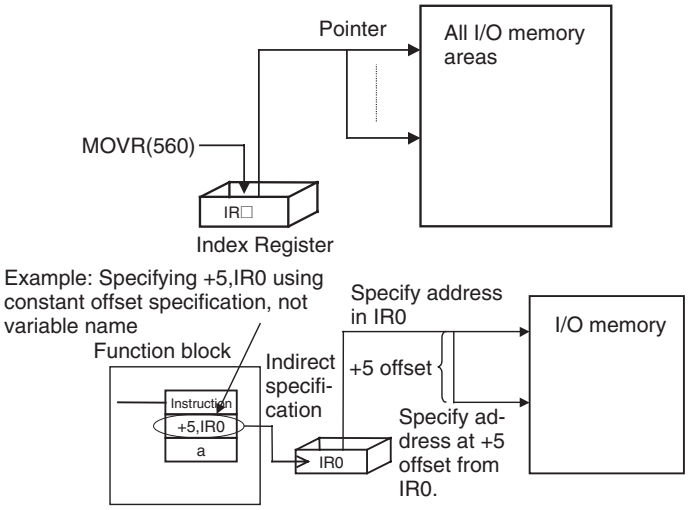
Example: Special I/O Units



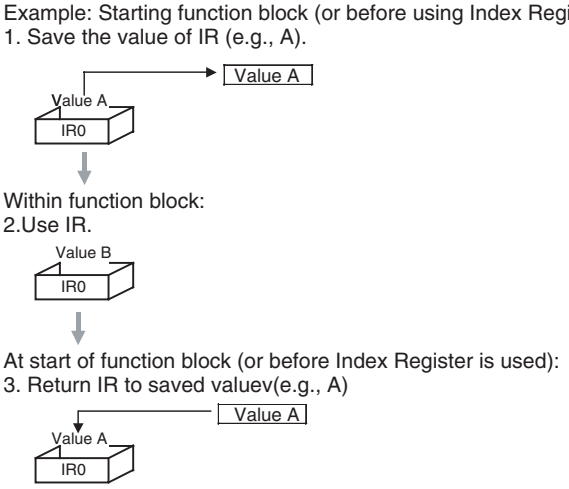
2-4-6 Using Index Registers

Index Registers IR0 to IR15 function as pointers for specifying I/O memory addresses. These Index Registers can be used within function blocks to directly specify addresses using IR0 to IR15 and not the variable names (Index Register direct specification: IR0 to IR15; Index Register indirect specification: ,IR0 to ,IR15)

Note After storing the real I/O memory addresses in the Index Registers using the MOVR(560) instruction, Index Registers can be indirectly specified using general instructions. This enables all I/O memory areas to be specified dynamically.



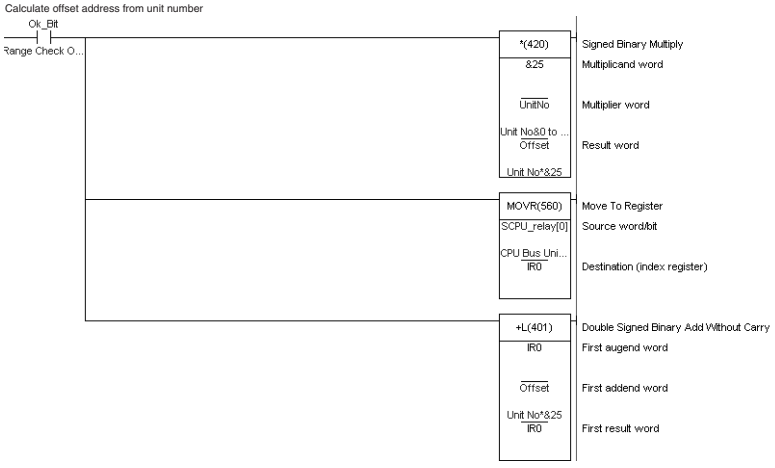
Note (1) When Index Registers IR0 to IR15 are used within function blocks, using the same Index Register within other function blocks or in the program outside of function blocks will create competition between the two instances and the program will not execute properly. Therefore, when using Index Registers (IR0 to IR15), always save the value of the Index Register at the point when the function block starts (or before the Index Register is used), and when the function block is completed (or after the Index Register has been used), incorporate processing in the program to return the Index Register to the saved value.



(2) Always set the value before using Index Registers. Operation will not be stable if Index Registers are used without the values being set.

Application Examples

The following examples are for using Index Registers IR0 to IR15 within function blocks.

Example	Details
Saving the Index Register Value before Using Index Register 	When Index Registers are used within this function block, processing to save the Index Register value is performed when the function starts (or before the Index Register is used) to enable the value to be returned to the original Index Register value after the function block is completed (or after the Index Register is used). Example: Save the contents of Index Register IR0 by storing it in *SaveIR[0]* (internal variable, data type DINT, 1 array element).
Using Index Registers 1) Setting the value in the Index Register. (Stores the real I/O memory address for first CIO Area word n) 	Example: The real I/O memory address for the first word of CIO 1500 + unit number × 25 allocated in the CPU Bus Unit allocation area based on the CPU Bus Unit's unit number (&0 to &15) passed from the function block is stored in IR0. Procedure: Assumes that unit numbers &0 to &15 have already been input (from outside the function block) in *UnitNo* (input variables, INT data type). 1. Multiple *UnitNo* by *&25*, and store in *Offset* (internal variable, DINT data type) 2. Store the real I/O memory address for *SCPU_Relay* (internal variable, WORD data type, (if required, specify the array as 400 elements (see note), AT setting = 1500)) in Index Register IR0. **Note** Specifying an array for SCPU_relay, such as *SCPU_relay [2]*, for example, enables the address CIO 1500 + (*UnitNo* × &25) + 2 to be specified. This also applies in example 2 below. 3. Increment the real I/O memory address in Index Register IR0 by the value for the variable *Offset* (variable *UnitNo* × &25).

Example	Details
2) Specifying constant offset of Index Register (Specifying a bit between CIO n+0 to n+24)	The real I/O memory address for CIO 1500 + (*UnitNo* × &25) is stored in Index Register IR0 by the processing in step 1 above. Therefore the word address is specified using the constant offset from IR0. For example, specifying +2,IR0 will specify CIO 1500 + (*UnitNo* × &25) + 2. **Note** CIO 1500 + (*UnitNo* × &25) + 2 can also by specified by specifying *SCPU_relay [2]* using the array setting with *SCPU_relay*. Specify bit addresses using instructions that can specify bit addresses within words (e.g., second operand of TST(350/351)/ SETB(532) instructions).
	Example: Variable *NodeSelf_OK* turns ON when *NetCheck_OK* (internal variable, BOOL data type) is ON and bit 15 of the word at the +6 offset from IR0 (CIO 1500 + *UnitNo* × &25 +6) is ON.
Returning the Index Register to the Prior Value 	The Index Register returns to the original value after this function block is completed (or after the Index Register has been used). Example: The value for variable *SaveIR[0]* that was saved is stored in Index Register IR0, and the value is returned to the contents from when this function started (or prior to using the Index Register).

2-5 Precautions for Instructions with Operands Specifying the First or Last of Multiple Words

When using ladder programming to create function blocks with instruction operands specifying the first or last of a range of words, the following precautions apply when specifying variables for the operand.

When the operand specifies the first or last word of multiple words, the instruction operates according to the internally allocated address for AT setting (or external variable setting). Therefore, the variable data type and number of array elements are unrelated to the operation of the instruction. Either specify a variable with an AT setting, or an array variable with a size that matches the data size to be processed by the instruction.

For details on whether an AT setting (or external variable setting) or an array setting for a number of elements is required to specify the first address of a range of words in the instruction operand, refer to *2-6 Instruction Support and Operand Restrictions*.

Note To specify the first or last of multiple words in an instruction operand, always specify a variable with AT setting (or an external variable), or a variable with the same size as the data size to be processed in the instruction. The following precautions apply.

1,2,3... 1. If a non-array variable is specified without AT setting and without a matching data size, the CX-Programmer will output an error when compiling.

2. The following precautions apply to when an array variable is specified.

Size to Be Processed in the Instruction Operand Is Fixed

Make sure that the number of elements in the array is the same as size to be processed by the instruction. Otherwise, the CX-Programmer will output an error when compiling.

Size to Be Processed in the Instruction Operand Is Not Fixed

Make sure that the number of elements in the array is the same or greater than the size specified by another operand.

Other Operand Specifying Size: Constant

The CX-Programmer outputs an error when compiling.

Other Operand Specifying Size: Variable

The CX-Programmer will not output an error when compiling (a warning message will be displayed) even if the number of elements in the array does not match the size specified in another operand (variable).

In particular, when the number of elements in the array is less than the size specified by another operand, (for example, when instruction processing size is 16 and the number of elements actually registered in the variable table is 10), the instruction will execute read/write processing in the areas exceeding the number of elements. (In this example, read/write processing will be executed for the next 6 words after the number of elements registered in the actual variable table.) If the same area is being used by another instruction (including internal variable allocations), unexpected operation may occur, which may result in a serious accident.

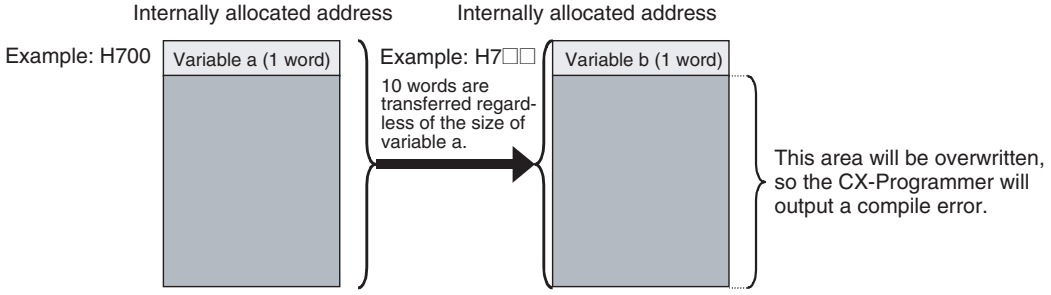
Do not use variables with a size that does not match the data size to be processed by the instruction in the operand specifying the first address (or last address) for a range of words. Always use either non-array variables data type with a size that is the same as the data size required by the instruction or array variable with the number of elements that is the same as the data size required by the instruction. Otherwise, the following errors will occur.

Non-array Variables without Matching Data Size and without AT Setting

If the operand specifying the first address (or last address) of multiple words uses a non-array variable data type with a size that does not match the data size required by the instruction and an AT setting is also not used, the CX-Programmer will output a compile error.

Example: BLOCK TRANSFER(070) instruction: XFER W S D
(W: Number of words, S: First source word; D: First destination word)
When *&10* is specified in W, variable *a* with data type WORD is specified in S, and variable *b* with data type WORD is specified in D: XFER &10 a b
The XFER(070) instruction will transfer the data in the 10 words beginning from the automatically allocated address in variable *a* to the 10 words beginning with the automatically allocated address in variable *b*. Therefore, the CX-Programmer will output a compile error.

Example: XFER &10 a b
(variables a and b are WORD data types)



Array Variables

The result depends on the following conditions.

Size to Be Processed by Instruction Is Fixed

If the size to be processed by the instruction is a fixed operand, and this size does not match the number of array elements, the CX-Programmer will output a compile error.

Example: LINE TO COLUMN(064) instruction; COLM S D N

(S: Bit number, D: First destination word, N: Source word)

E.g., COLM a b[0] c

If an array for a WORD data type with 10 array elements is specified in D when it should be for 16 array elements, the CX-Programmer will output an error when compiling.

Size to Be Processed by Instruction Is Not Fixed

When the operand size to be processed by the instruction is not fixed (when the size is specified by another operand in the instruction), make sure that the number of array elements is the same or greater than the size specified in the other operand (i.e., size to be processed by the instruction).

Other Operand Specifying Size: Constant

The CX-Programmer will output an error when compiling.

Example: BLOCK TRANSFER: XFER W S D

(W: Number of words, S: First source word; D: First destination word)

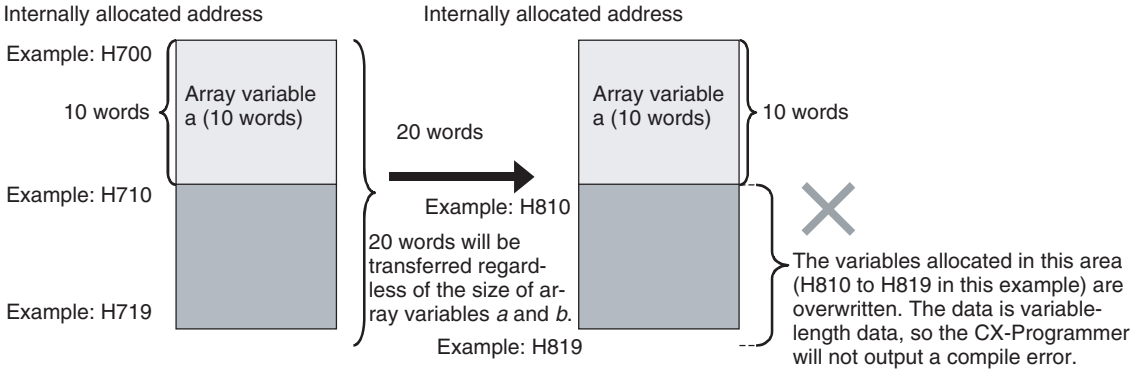
When *&20* is specified in W, array variable *a* with data type WORD and 10 elements is specified in S, and array variable *b* with data type WORD and 10 elements is specified in D:

XFER &20 a[0] b[0]

Even though the array variables a[0] and b[0] are both 10 words, the XFER(070) instruction will execute transfer processing for the 20 words specified in W. As a result, the XFER(070) instruction will perform read/write processing for the I/O memory area following the number of array elements that was allocated, as shown in the following diagram.

Therefore, if a[10 elements] is internally allocated words (e.g., H700 to H709), and b[10 elements] is internally allocated words (e.g., H800 to H809), XFER(070) will transfer data in words H700 to H719 to words H800 to H819. In this operation, if another internally allocated variable (e.g., c), is allocated words in H810 to H819, the words will be overwritten, causing unexpected operation to occur. To transfer 20 words, make sure that the number of elements is specified as 20 elements for both array variable *a* and *b*.

XFER &20 a[0] b[0]
Using a WORD data type with 10 elements for both variables *a* and *b*:
To transfer 20 words, be sure to specify 20 elements for both array variables *a* and *b*.



Other Operand Specifying Size: Variable

Even if the number of array elements does not match the size (i.e., size to be processed by the instruction) specified in another operand (variable), the CX-Programmer will not output an error when compiling. The instruction will be executed according to the size specified by the operand, regardless of the number of elements in the array variable.

Particularly if the number of elements in the array is less than the size (i.e., size to be processed by the instruction) specified by another operand (variable), other variables will be affected and unexpected operation may occur.

2-6 Instruction Support and Operand Restrictions

The tables in this appendix indicate which instructions can be used in function blocks created with ladder programming language, the restrictions that apply to them and to the operands, including the variables (whether array variables and AT settings or external variable specifications are required, and which data types can be used).

Instruction Support

Instructions that are not supported for use in function block definitions by the CX-Programmer and CS/CJ-series CPU Units with unit version 3.0 are given as *Not supported in function blocks* in the *Symbol* column.

Restrictions on Operands

- Operands that specify the first or last of multiple words, thereby requiring AT setting or specification of array variables, are indicated as follows in the *AT setting or array required* column.
  Yes: An AT setting (or external specification) or array variable must be specified for the operand to specify the first or last of multiple words.
  - The value within parentheses is the fixed size used by the instruction for reading, writing, or other processing. This size indicates either the data type size or the size required for the array variable specified in word units. For array variables, this size must be the same as the number of elements. Otherwise, the CX-Programmer will output an error when compiling.

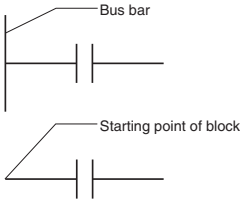
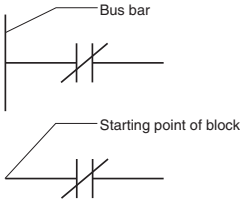
- If “not fixed” is indicated in parentheses, the size used by the instruction for reading, writing, or other processing can be changed. Make sure that the maximum size required for the number of array elements is provided.
  Even if the number of array elements in an operand with unfixed size does not match the size specified in another operand, the CX-Programmer will not output an error when compiling. The instruction will operate according to the size specified in the other operand, regardless of the number of array variable elements.

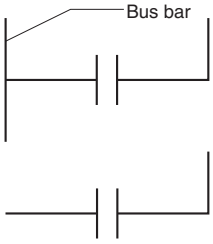
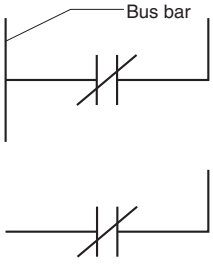
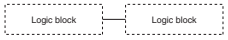
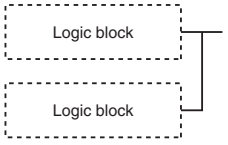
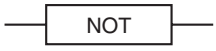
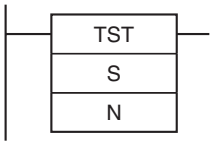
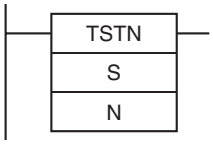
---: Operands that do not require an AT setting or specification of array variables.

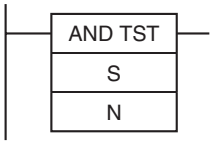
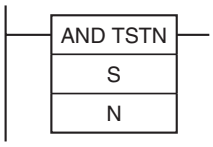
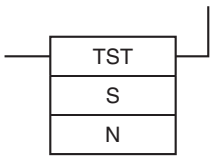
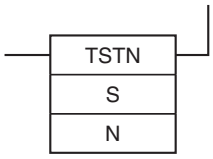
Note When specifying the first or last word of multiple words in an instruction operand, input parameters cannot be used to pass data to or from variables. Either an AT setting or an array variable with the required number of elements must be prepared, and after the array data is set in the function block definition, the first or last element in the array variable must be specified for the operand.

- Any operands for which an AT setting must be specified for an I/O memory address on a remote node in the network are indicated as *Specify address at remote node with AT setting* in the *AT setting or array required* column.

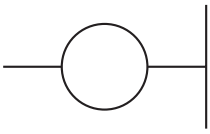
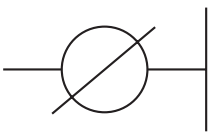
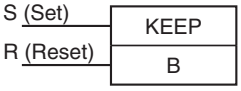
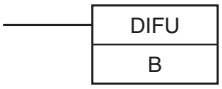
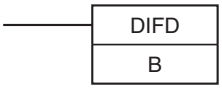
2-6-1 Sequence Input Instructions

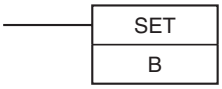
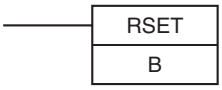
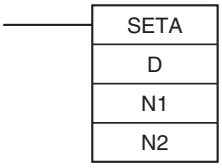
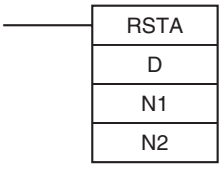
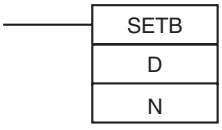
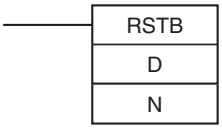
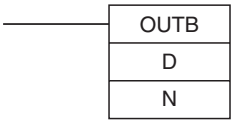
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
LOAD	LD @LD %LD !LD !@LD !%LD			B: Bit	BOOL	---
LOAD NOT	LD NOT !LD NOT @LD NOT %LD NOT !@LD NOT !%LD NOT			B: Bit	BOOL	---
AND	AND @AND %AND !AND !@AND !%AND			B: Bit	BOOL	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
AND NOT	AND NOT !AND NOT @AND NOT %AND NOT !@AND NOT !%AND NOT			---	BOOL	---
OR	OR @OR %OR !OR !@OR !%OR			---	BOOL	---
OR NOT	OR NOT !OR NOT @OR NOT %OR NOT !@OR NOT !%OR NOT			---	BOOL	---
AND LOAD	AND LD			---	---	---
OR LOAD	OR LD			---	---	---
NOT	NOT	520		---	---	---
CONDITION ON	UP	521		---	---	---
CONDITION OFF	DOWN	522		---	---	---
BIT TEST	LD TST	350		S: Source word N: Bit number	WORD UINT	--- ---
BIT TEST	LD TSTN	351		S: Source word N: Bit number	WORD UINT	--- ---

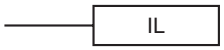
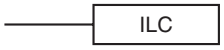
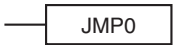
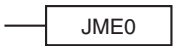
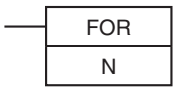
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BIT TEST	AND TST	350		S: Source word	WORD	---
				N: Bit number	UINT	---
BIT TEST	AND TSTN	351		S: Source word	WORD	---
				N: Bit number	UINT	---
BIT TEST	OR TST	350		S: Source word	WORD	---
				N: Bit number	UINT	---
BIT TEST	OR TSTN	351		S: Source word	WORD	---
				N: Bit number	UINT	---

2-6-2 Sequence Output Instructions

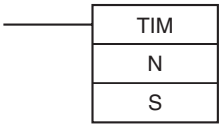
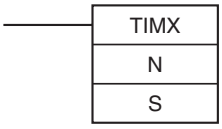
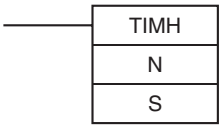
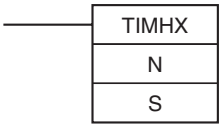
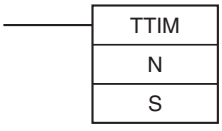
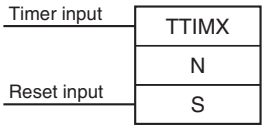
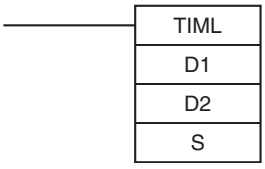
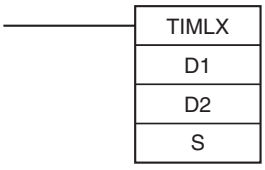
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
OUTPUT	OUT !OUT			---	BOOL	---
OUTPUT NOT	OUT NOT !OUT NOT			---	BOOL	---
KEEP	KEEP !KEEP	011		---	BOOL	---
DIFFERENTIATE UP	DIFU !DIFU	013		---	BOOL	---
DIFFERENTIATE DOWN	DIFD !DIFD	014		---	BOOL	---

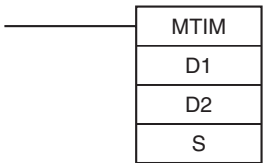
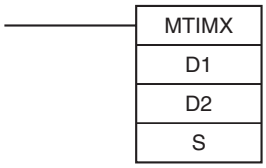
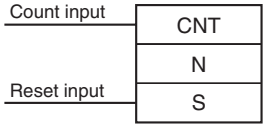
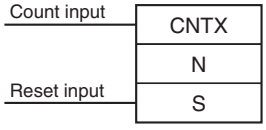
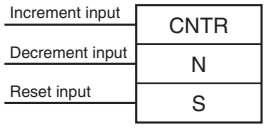
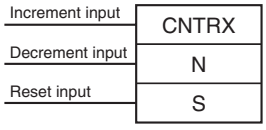
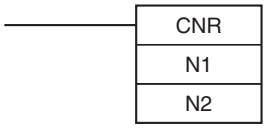
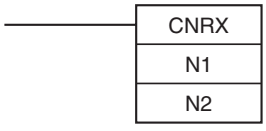
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SET	SET @SET %SET !SET !@SET !%SET			B: Bit	BOOL	---
RESET	RSET @RSET %RSET !RSET !@RSET !%RSET			B: Bit	BOOL	---
MULTIPLE BIT SET	SETA @SETA	530		D: Beginning word N1: Beginning bit N2: Number of bits	UINT UINT UINT	Yes (not fixed) --- ---
MULTIPLE BIT RESET	RSTA @RSTA	531		D: Beginning word N1: Beginning bit N2: Number of bits	UINT UINT UINT	Yes (not fixed) --- ---
SINGLE BIT SET	SETB @SETB !SETB	532		D: Word address N: Bit number	UINT UINT	--- ---
SINGLE BIT RESET	RSTB @RSTB !RSTB	533		D: Word address N: Bit number	UINT UINT	--- ---
SINGLE BIT OUTPUT	OUTB @OUTB !OUTB	534		D: Word address N: Bit number	UINT UINT	--- ---

2-6-3 Sequence Control Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
END	END	001		---	---	---
NO OPERATION	NOP	000	---	---	---	---
INTERLOCK	IL	002		---	---	---
INTERLOCK CLEAR	ILC	003		---	---	---
MULTI-INTERLOCK DIFFERENTIATION HOLD	MILH	517		N: Interlock number	# + decimal only	---
				D: Interlock Status Bit	BOOL	---
MULTI-INTERLOCK DIFFERENTIATION RELEASE	MILR	518		N: Interlock number	# + decimal only	---
				D: Interlock Status Bit	BOOL	---
MULTI-INTERLOCK CLEAR	MILC	519		N: Interlock number	# + decimal only	---
JUMP	JMP	004	Not supported in function blocks	N: Jump number	---	---
JUMP END	JME	005	Not supported in function blocks	N: Jump number	---	---
CONDITIONAL JUMP	CJP	510	Not supported in function blocks	N: Jump number	---	---
CONDITIONAL JUMP	CJPN	511	Not supported in function blocks	N: Jump number	---	---
MULTIPLE JUMP	JMP0	515		---	---	---
MULTIPLE JUMP END	JME0	516		---	---	---
FOR-NEXT LOOPS	FOR	512		N: Number of loops	UINT	---
BREAK LOOP	BREAK	514		---	---	---
FOR-NEXT LOOPS	NEXT	513		---	---	---

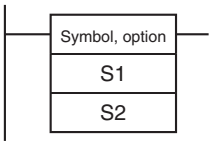
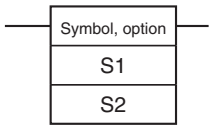
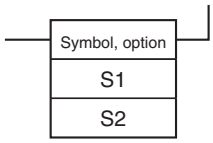
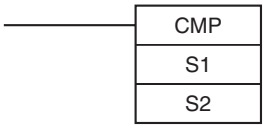
2-6-4 Timer and Counter Instructions

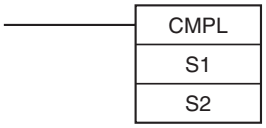
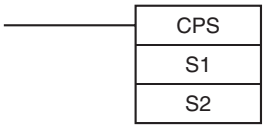
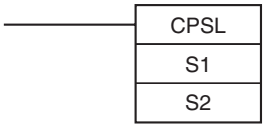
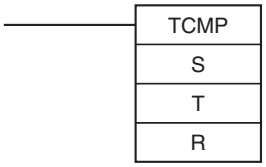
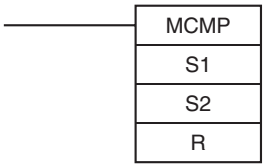
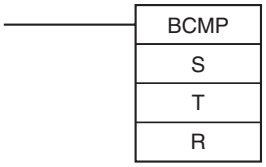
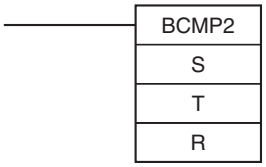
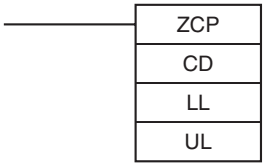
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
TIMER	TIM (BCD)			N: Timer number	TIMER	---
				S: Set value	WORD	---
	TIMX (BIN)	550		N: Timer number	TIMER	---
				S: Set value	UINT	---
HIGH-SPEED TIMER	TIMH (BCD)	015		N: Timer number	TIMER	---
				S: Set value	WORD	---
	TIMHX (BIN)	551		N: Timer number	TIMER	---
				S: Set value	UINT	---
ONE-MS TIMER	TMHH (BCD)	540	Not supported in function blocks.	N: Timer number	TIMER	---
				S: Set value	WORD	---
	TMHHX (BIN)	552	Not supported in function blocks.	N: Timer number	TIMER	---
				S: Set value	UINT	---
ACCUMULA-TIVE TIMER	TTIM (BCD)	087		N: Timer number	TIMER	---
				S: Set value	WORD	---
	TTIMX (BIN)	555		N: Timer number	TIMER	---
				S: Set value	UINT	---
LONG TIMER	TIML (BCD)	542		D1: Completion Flag	WORD	---
				D2: PV word	DWORD	---
				S: SV word	DWORD	---
	TIMLX (BIN)	553		D1: Completion Flags	UINT	---
				D2: PV word	UDINT	---
				S: SV word	UDINT	---

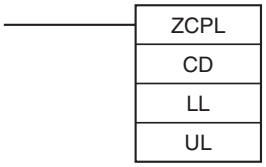
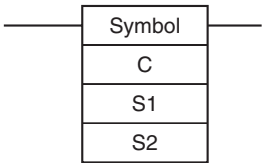
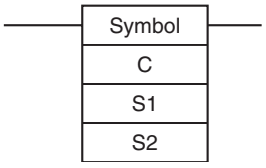
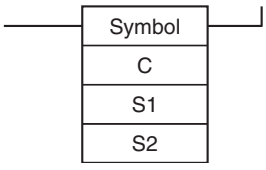
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
MULTI-OUTPUT TIMER	MTIM (BCD)	543		D1: Completion Flags	UINT	---
				D2: PV word	WORD	---
				S: 1st SV word	WORD	Yes (8)
	MTIMX (BIN)	554		D1: Completion Flags	UINT	---
				D2: PV word	UINT	---
				S: 1st SV word	WORD	Yes (8)
COUNTER	CNT (BCD)			N: Counter number	COUNTER	---
				S: Set value	WORD	---
	CNTX (BIN)	546		N: Counter number	COUNTER	---
				S: Set value	UINT	---
REVERSIBLE COUNTER	CNTR (BCD)	012		N: Counter number	COUNTER	---
				S: Set value	WORD	---
	CNTRX (BIN)	548		N: Counter number	COUNTER	---
				S: Set value	UINT	---
RESET TIMER/ COUNTER	CNR @CNR (BCD)	545		N1: 1st number in range	TIMER or COUNTER (See note.)	---
				N2: Last number in range	TIMER or COUNTER (See note.)	---
	CNRX @CNRX (BIN)	547		N1: 1st number in range	TIMER or COUNTER (See note.)	---
				N2: Last number in range	TIMER or COUNTER (See note.)	---

Note Enabled when the same variable is specified for N1 and N2.

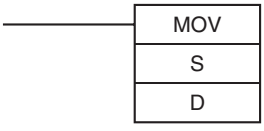
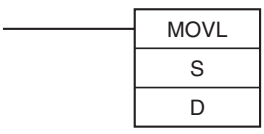
2-6-5 Comparison Instructions

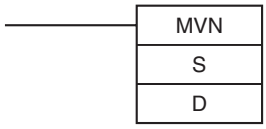
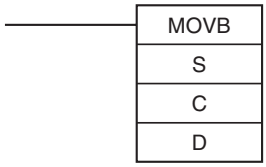
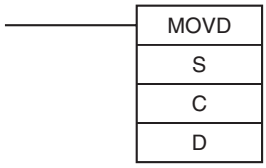
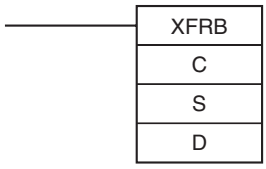
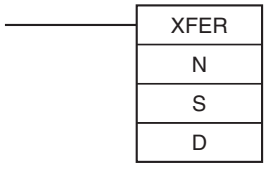
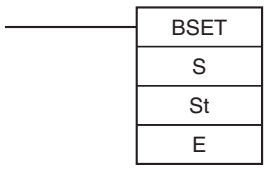
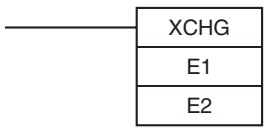
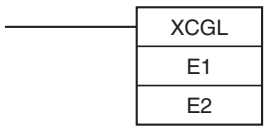
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
Symbol Comparison (Unsigned)	LD,AND, OR + =, <>, <, <=, >, >=	300 (=) 305 (<>) 310 (<) 315 (<=) 320 (>) 325 (>=)	Using LD:  Using AND:  Using OR: 	S1: Comparison data 1	UINT	---
				S2: Comparison data 2	UINT	---
Symbol Comparison (Double-word, unsigned)	LD,AND, OR + =, <>, <, <=, >, >= + L	301 (=) 306 (<>) 311 (<) 316 (<=) 321 (>) 326 (>=)	---	S1: Comparison data 1	UDINT	---
				S2: Comparison data 2	UDINT	---
Symbol Comparison (Signed)	LD,AND, OR + =, <>, <, <=, >, >= + S	302 (=) 307 (<>) 312 (<) 317 (<=) 322 (>) 327 (>=)	---	S1: Comparison data 1	INT	---
				S2: Comparison data 2	INT	---
Symbol Comparison (Double-word, signed)	LD,AND, OR + =, <>, <, <=, >, >= + SL	303 (=) 308 (<>) 313 (<) 318 (<=) 323 (>) 328 (>=)	---	S1: Comparison data 1	DINT	---
				S2: Comparison data 2	DINT	---
UNSIGNED COMPARE	CMP !CMP	020		S1: Comparison data 1	UINT	---
				S2: Comparison data 2	UINT	---

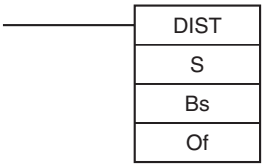
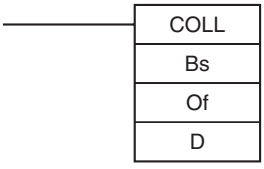
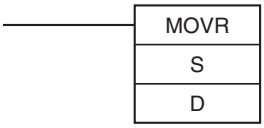
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE UNSIGNED COMPARE	CMPL	060		S1: Comparison data 1	UDINT	---
				S2: Comparison data 2	UDINT	---
SIGNED BINARY COMPARE	CPS !CPS	114		S1: Comparison data 1	INT	---
				S2: Comparison data 2	INT	---
DOUBLE SIGNED BINARY COMPARE	CPSL	115		S1: Comparison data 1	DINT	---
				S2: Comparison data 2	DINT	---
TABLE COMPARE	TCMP @TCMP	085		S: Source data	WORD	---
				T: 1st word of table	WORD	Yes (16)
				R: Result word	UINT	---
MULTIPLE COMPARE	MCMP @MCMP	019		S1: 1st word of set 1	WORD	Yes (16)
				S2: 1st word of set 2	WORD	Yes (16)
				R: Result word	UINT	
UNSIGNED BLOCK COMPARE	BCMP @BCMP	068		S: Source data	WORD	---
				T: 1st word of table	WORD	Yes (32)
				R: Result word	UINT	---
EXPANDED BLOCK COMPARE	BCMP2 @BCMP2	502		S: Source data	WORD	---
				T: 1st word of block	WORD	Yes (not fixed)
				R: Result word	WORD	---
AREA RANGE COMPARE	ZCP	088		CD: Compare data (1 word)	UINT	---
				LL: Lower limit of range	UINT	---
				UL: Upper limit of range	UINT	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE AREA RANGE COMPARE	ZCPL	116		CD: Compare data (2 words)	UDINT	---
				LL: Lower limit of range	UDINT	---
				UL: Upper limit of range	UDINT	---
Time Comparison	LD, AND, OR + =DT, <> DT, <DT, <=DT, >DT, >=DT	341 (=DT) 342 (<>DT) 343 (<DT) 344 (<=DT) 345 (>DT) 346 (>=DT)	LD (LOAD):  AND:  OR: 	C: Control word	WORD	---
				S1: 1st word of present time	WORD	Yes (3)
				S2: 1st word of comparison time	WORD	Yes (3)

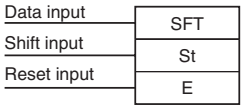
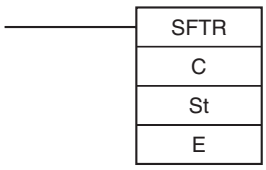
2-6-6 Data Movement Instructions

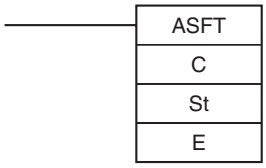
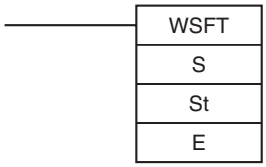
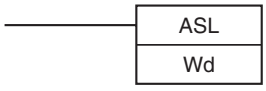
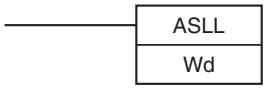
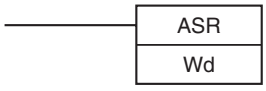
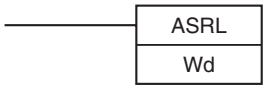
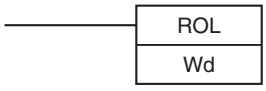
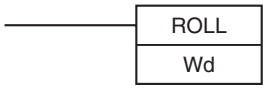
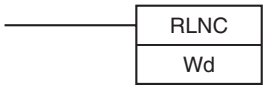
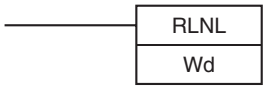
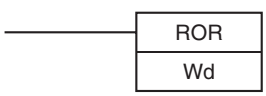
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
MOVE	MOV @MOV !MOV !@MOV	021		S: Source	WORD	---
				D: Destination	WORD	---
DOUBLE MOVE	MOVL @MOVL	498		S: 1st source word	DWORD	---
				D: 1st destination word	DWORD	---

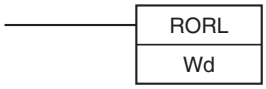
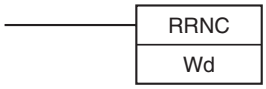
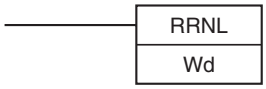
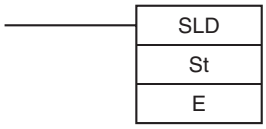
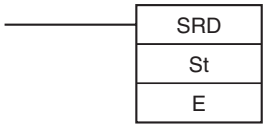
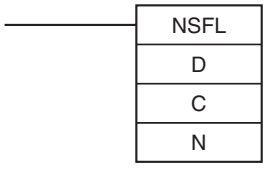
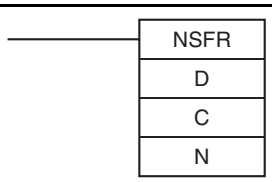
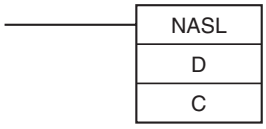
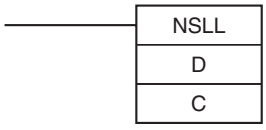
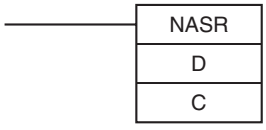
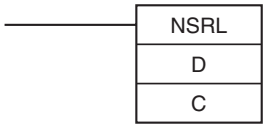
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
MOVE NOT	MVN @MVN	022		S: Source	WORD	---
				D: Destination	WORD	---
DOUBLE MOVE NOT	MVNL @MVNL	499		S: 1st source word	DWORD	---
				D: 1st destination word	DWORD	---
MOVE BIT	MOVB @MOVB	082		S: Source word or data	WORD	---
				C: Control word	UINT	---
				D: Destination word	WORD	---
MOVE DIGIT	MOVD @MOVD	083		S: Source word or data	WORD	---
				C: Control word	UINT	---
				D: Destination word	UINT	---
MULTIPLE BIT TRANSFERÅ@	XFRB @XFRB	062		C: Control word	UINT	---
				S: 1st source word	WORD	Yes (not fixed)
				D: 1st destination word	WORD	Yes (not fixed)
BLOCK TRANSFER	XFER @XFER	070		N: Number of words	UINT	---
				S: 1st source word	WORD	Yes (not fixed)
				D: 1st destination word	WORD	Yes (not fixed)
BLOCK SET	BSET @BSET	071		S: Source word	WORD	---
				St: Starting word	WORD	Yes (not fixed)
				E: End word	WORD	Yes (not fixed)
DATA EXCHANGE	XCHG @XCHG	073		E1: 1st exchange word	WORD	---
				E2: Second exchange word	WORD	---
DOUBLE DATA EXCHANGE	XCGL @XCGL	562		E1: 1st exchange word	DWORD	---
				E2: Second exchange word	DWORD	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SINGLE WORD DISTRIBUTE	DIST @DIST	080		S: Source word	WORD	---
				Bs: Destination base address	WORD	Yes (not fixed)
				Of: Offset	UINT	---
DATA COLLECT	COLL @COLL	081		Bs: Source base address	WORD	Yes (not fixed)
				Of: Offset	WORD	---
				D: Destination word	WORD	---
MOVE TO REGISTER	MOVR @MOVR	560		S: Source (desired word orbit)	BOOL	---
				D: Destination (Index Register)	WORD	---
MOVE TIMER/ COUNTER PV TO REGISTER	MOVRW @MOVRW	561	Not supported in function blocks	S: Source (desired TC number)	---	---
				D: Destination (Index Register)	---	---

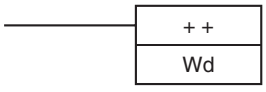
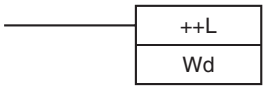
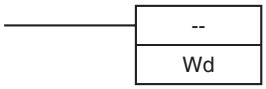
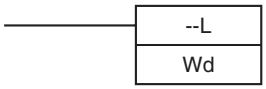
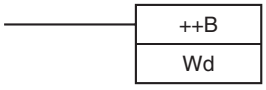
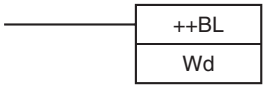
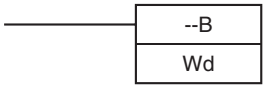
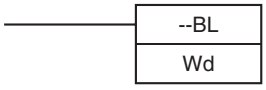
2-6-7 Data Shift Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SHIFT REGISTER	SFT	010		St: Starting word	UINT	Yes (not fixed)
				E: End word	UINT	Yes (not fixed) D1 and D2 must be the same array variable when array variables are required.
REVERSIBLE SHIFT REGISTER	SFTR @SFTR	084		C: Control word	UINT	---
				St: Starting word	UINT	Yes (not fixed)
				E: End word	UINT	Yes (not fixed) D1 and D2 must be the same array variable when array variables are required.

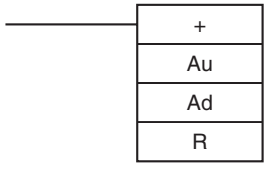
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
ASYNCHRONOUS SHIFT REGISTER	ASFT @ASFT	017		C: Control word	UINT	---
				St: Starting word	UINT	Yes (not fixed)
				E: End word	UINT	Yes (not fixed) D1 and D2 must be the same array variable when array variables are required.
WORD SHIFT	WSFT @WSFT	016		S: Source word	WORD	
				St: Starting word	UINT	Yes (not fixed)
				E: End word	UINT	Yes (not fixed) D1 and D2 must be the same array variable when array variables are required.
ARITHMETIC SHIFT LEFT	ASL @ASL	025		Wd: Word	UINT	---
DOUBLE SHIFT LEFT	ASLL @ASLL	570		Wd: Word	UDINT	---
ARITHMETIC SHIFT RIGHT	ASR @ASR	026		Wd: Word	UINT	---
DOUBLE SHIFT RIGHT	ASRL @ASRL	571		Wd: Word	UDINT	---
ROTATE LEFT	ROL @ROL	027		Wd: Word	UINT	---
DOUBLE ROTATE LEFT	ROLL @ROLL	572		Wd: Word	UDINT	---
ROTATE LEFT WITHOUT CARRY	RLNC @RLNC	574		Wd: Word	UINT	---
DOUBLE ROTATE LEFT WITHOUT CARRY	RLNL @RLNL	576		Wd: Word	UDINT	---
ROTATE RIGHT	ROR @ROR	028		Wd: Word	UINT	---

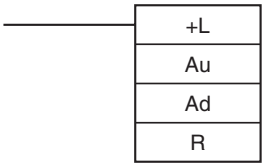
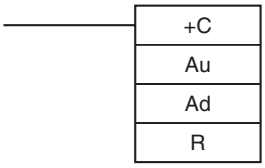
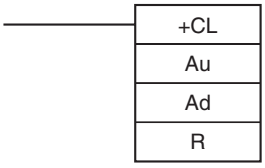
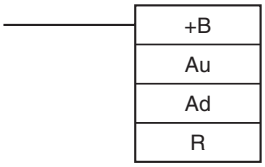
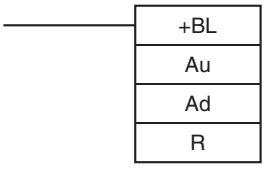
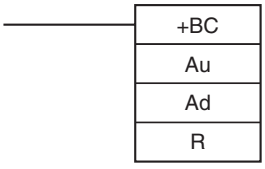
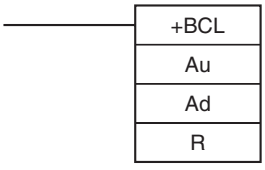
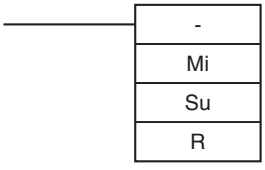
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE ROTATE RIGHT	RORL @RORL	573		Wd: Word	UDINT	---
ROTATE RIGHT WITHOUT CARRY	RRNC @RRNC	575		Wd: Word	UINT	---
DOUBLE ROTATE RIGHT WITHOUT CARRY	RRNL @RRNL	577		Wd: Word	UDINT	---
ONE DIGIT SHIFT LEFT	SLD @SLD	074		St: Starting word E: End word	UINT UINT	Yes (not fixed) Yes (not fixed)
ONE DIGIT SHIFT RIGHT	SRD @SRD	075		St: Starting word E: End word	UINT UINT	Yes (not fixed) Yes (not fixed)
SHIFT N-BIT DATA LEFT	NSFL @NSFL	578		D: Beginning word for shift C: Beginning bit N: Shift data length	UINT UINT UINT	Yes (not fixed) --- ---
SHIFT N-BIT DATA RIGHT	NSFR @NSFR	579		D: Beginning word for shift C: Beginning bit N: Shift data length	UINT UINT UINT	Yes (not fixed) --- ---
SHIFT N-BITS LEFT	NASL @NASL	580		D: Shift word C: Control word	UINT UINT	--- ---
DOUBLE SHIFT N-BITS LEFT	NSLL @NSLL	582		D: Shift word C: Control word	UDINT UINT	--- ---
SHIFT N-BITS RIGHT	NASR @NASR	581		D: Shift word C: Control word	UINT UINT	--- ---
DOUBLE SHIFT N-BITS RIGHT	NSRL @NSRL	583		D: Shift word C: Control word	UDINT UINT	--- ---

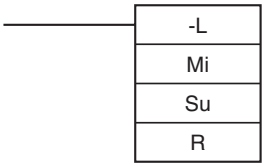
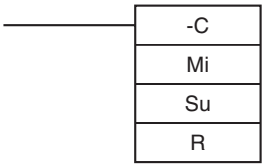
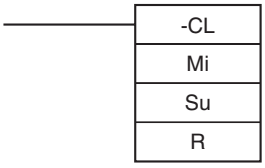
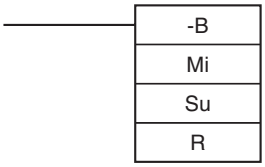
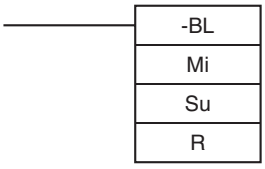
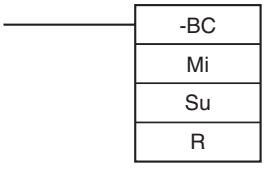
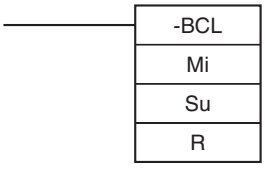
2-6-8 Increment/Decrement Instructions

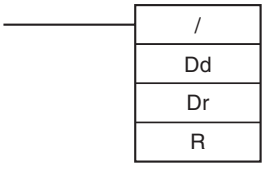
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
INCREMENT BINARY	++ @++	590		Wd: Word	UINT	---
DOUBLE INCREMENT BINARY	++L @++L	591		Wd: Word	UDINT	---
DECREMENT BINARY	-- @--	592		Wd: Word	UINT	---
DOUBLE DECREMENT BINARY	--L @--L	593		Wd: 1st word	UDINT	---
INCREMENT BCD	++B @++B	594		Wd: Word	WORD	---
DOUBLE INCREMENT BCD	++BL @++BL	595		Wd: 1st word	DWORD	---
DECREMENT BCD	--B @--B	596		Wd: Word	DWORD	---
DOUBLE DECREMENT BCD	--BL @--BL	597		Wd: 1st word	WORD	---

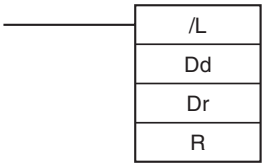
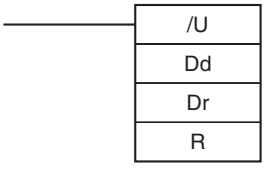
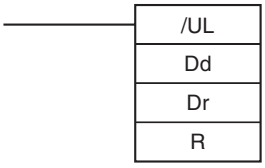
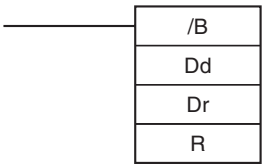
2-6-9 Symbol Math Instructions

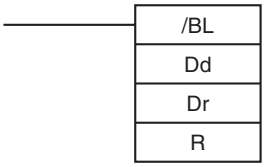
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SIGNED BINARY ADD WITHOUT CARRY	+ @+	400		Au: Augend word	INT	---
				Ad: Addend word	INT	---
				R: Result word	INT	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE SIGNED BINARY ADD WITHOUT CARRY	+L @+L	401		Au: 1st augend word	DINT	---
				Ad: 1st addend word	DINT	---
				R: 1st result word	DINT	---
SIGNED BINARY ADD WITH CARRY	+C @+C	402		Au: Augend word	INT	---
				Ad: Addend word	INT	---
				R: Result word	INT	---
DOUBLE SIGNED BINARY ADD WITH CARRY	+CL @+CL	403		Au: 1st augend word	DINT	---
				Ad: 1st addend word	DINT	---
				R: 1st result word	DINT	---
BCD ADD WITHOUT CARRY	+B @+B	404		Au: Augend word	WORD	---
				Ad: Addend word	WORD	---
				R: Result word	WORD	---
DOUBLE BCD ADD WITHOUT CARRY	+BL @+BL	405		Au: 1st augend word	DWORD	---
				Ad: 1st addend word	DWORD	---
				R: 1st result word	DWORD	---
BCD ADD WITH CARRY	+BC @+BC	406		Au: Augend word	WORD	---
				Ad: Addend word	WORD	---
				R: Result word	WORD	---
DOUBLE BCD ADD WITH CARRY	+BCL @+BCL	407		Au: 1st augend word	DWORD	---
				Ad: 1st addend word	DWORD	---
				R: 1st result word	DWORD	---
SIGNED BINARY SUBTRACT WITHOUT CARRY	- @-	410		Mi: Minuend word	INT	---
				Su: Subtrahend word	INT	---
				R: Result word	INT	---

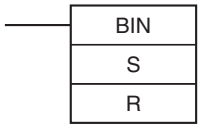
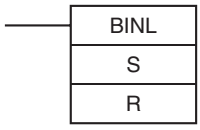
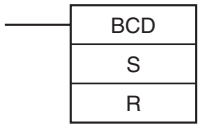
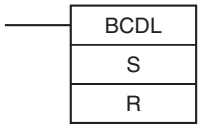
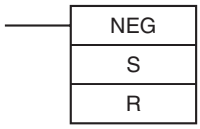
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE SIGNED BINARY SUBTRACT WITHOUT CARRY	-L @-L	411		Mi: Minuend word	DINT	---
				Su: Subtrahend word	DINT	---
				R: Result word	DINT	---
SIGNED BINARY SUBTRACT WITH CARRY	-C @-C	412		Mi: Minuend word	INT	---
				Su: Subtrahend word	INT	---
				R: Result word	INT	---
DOUBLE SIGNED BINARY WITH CARRY	-CL @-CL	413		Mi: Minuend word	DINT	---
				Su: Subtrahend word	DINT	---
				R: Result word	DINT	---
BCD SUBTRACT WITHOUT CARRY	-B @-B	414		Mi: Minuend word	WORD	---
				Su: Subtrahend word	WORD	---
				R: Result word	WORD	---
DOUBLE BCD SUBTRACT WITHOUT CARRY	-BL @-BL	415		Mi: 1st minuend word	DWORD	---
				Su: 1st subtrahend word	DWORD	---
				R: 1st result word	DWORD	---
BCD SUBTRACT WITH CARRY	-BC @-BC	416		Mi: Minuend word	WORD	---
				Su: Subtrahend word	WORD	---
				R: Result word	WORD	---
DOUBLE BCD SUBTRACT WITH CARRY	-BCL @-BCL	417		Mi: 1st minuend word	DWORD	---
				Su: 1st subtrahend word	DWORD	---
				R: 1st result word	DWORD	---
SIGNED BINARY MULTIPLY	* @*	420		Md: Multiplicand word	INT	---
				Mr: Multiplier word	INT	---
				R: Result word	DINT	---

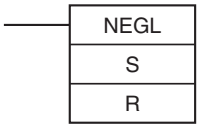
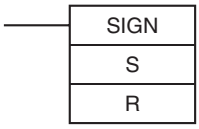
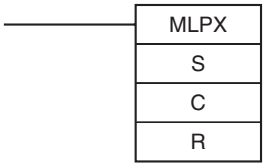
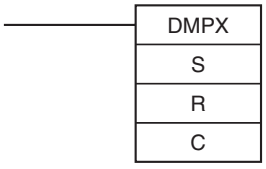
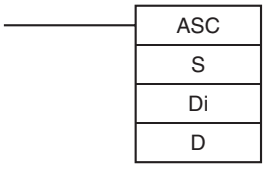
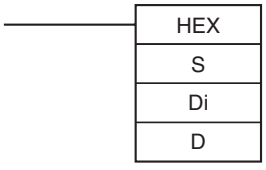
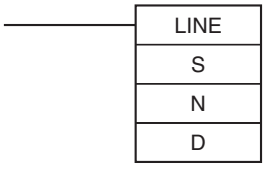
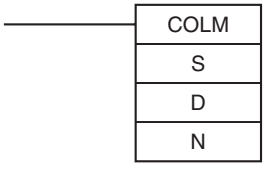
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE SIGNED BINARY MULTIPLY	*L @*L	421		Md: 1st multiplicand word	DINT	---
				Mr: 1st multiplier word	DINT	---
				R: 1st result word	LINT	---
UNSIGNED BINARY MULTIPLY	*U @*U	422		Md: Multiplicand word	UINT	---
				Mr: Multiplier word	UINT	---
				R: Result word	UINT	---
DOUBLE UNSIGNED BINARY MULTIPLY	*UL @*UL	423		Md: 1st multiplicand word	UDINT	---
				Mr: 1st multiplier word	UDINT	---
				R: 1st result word	ULINT	---
BCD MULTIPLY	*B @*B	424		Md: Multiplicand word	WORD	---
				Mr: Multiplier word	WORD	---
				R: Result word	DWORD	---
DOUBLE BCD MULTIPLY	*BL @*BL	425		Md: 1st multiplicand word	DWORD	---
				Mr: 1st multiplier word	DWORD	---
				R: 1st result word	LWORD	---
SIGNED BINARY DIVIDE	/ @/	430		Dd: Dividend word	INT	---
				Dr: Divisor word	INT	---
				R: Result word	DWORD	Yes (2) INT must be used when array variables are required.

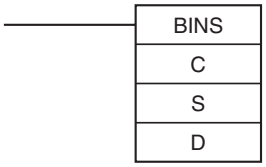
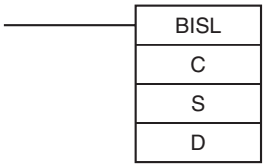
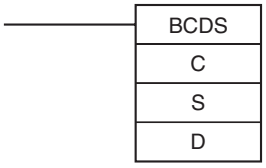
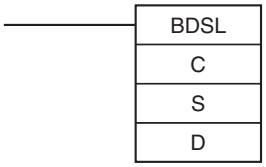
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE SIGNED BINARY DIVIDE	/L @/L	431		Dd: 1st dividend word	DINT	---
				Dr: 1st divisor word	DINT	---
				R: 1st result word	LWORD	Yes (2) DINT must be used when array variables are required.
UNSIGNED BINARY DIVIDE	/U @/U	432		Dd: Dividend word	UINT	---
				Dr: Divisor word	UINT	---
				R: Result word	DWORD	Yes (2) UINT must be used when array variables are required.
DOUBLE UNSIGNED BINARY DIVIDE	/UL @/UL	433		Dd: 1st dividend word	UDINT	---
				Dr: 1st divisor word	UDINT	---
				R: 1st result word	LWORD	Yes (2) UDINT must be used when array variables are required.
BCD DIVIDE	/B @/B	434		Dd: Dividend word	WORD	---
				Dr: Divisor word	WORD	---
				R: Result word	DWORD	Yes (2) WORD must be used when array variables are required.

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE BCD DIVIDE	/BL @/BL	435		Dd: 1st dividend word	DWORD	---
				Dr: 1st divisor word	DWORD	---
				R: 1st result word	LWORD	Yes (2) DWORD must be used when array variables are required.

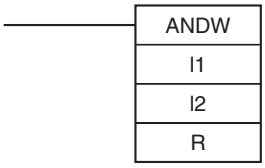
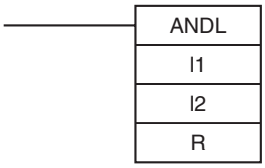
2-6-10 Conversion Instructions

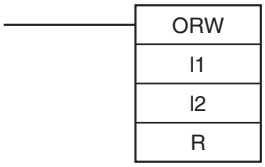
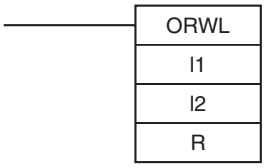
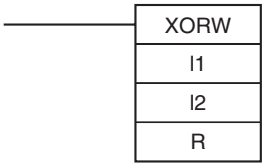
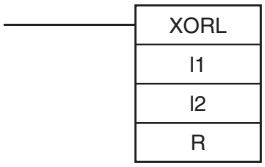
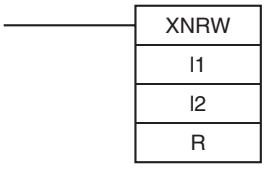
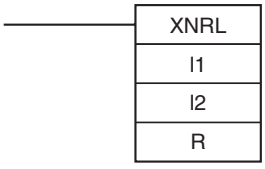
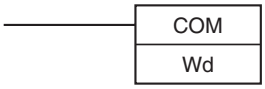
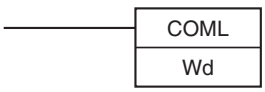
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BCD-TO-BINARY	BIN @BIN	023		S: Source word	WORD	---
				R: Result word	UINT	---
DOUBLE BCD-TO-DOUBLE BINARY	BINL @BINL	058		S: 1st source word	DWORD	---
				R: 1st result word	UDINT	---
BINARY-TO-BCD	BCD @BCD	024		S: Source word	UINT	---
				R: Result word	WORD	---
DOUBLE BINARY-TO-DOUBLE BCD	BCDL @BCDL	059		S: 1st source word	UDINT	---
				R: 1st result word	DWORD	---
2'S COMPLEMENT	NEG @NEG	160		S: Source word	WORD	---
				R: Result word	UINT	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE 2'S COMPLEMENT	NEGL @NEGL	161		S: 1st source word	DWORD	---
				R: 1st result word	UDINT	---
16-BIT TO 32-BIT SIGNED BINARY	SIGN @SIGN	600		S: Source word	WORD	---
				R: 1st result word	DINT	---
DATA DECODER	MLPX @MLPX	076		S: Source word	UINT	---
				C: Control word	UINT	---
				R: 1st result word	UINT	Yes (not fixed)
DATA ENCODER	DMPX @DMPX	077		S: 1st source word	UINT	Yes (not fixed)
				R: Result word	UINT	---
				C: Control word	UINT	---
ASCII CONVERT	ASC @ASC	086		S: Source word	UINT	---
				Di: Digit designator	UINT	---
				D: 1st destination word	UINT	Yes (3)
ASCII TO HEX	HEX @HEX	162		S: 1st source word	UINT	Yes (2)
				Di: Digit designator	UINT	---
				D: Destination word	UINT	Yes (not fixed)
COLUMN TO LINE	LINE @LINE	063		S: 1st source word	WORD	Yes (16)
				N: Bit number	UINT	---
				D: Destination word	UINT	---
LINE TO COLUMN	COLM @COLM	064		S: Source word	WORD	---
				D: 1st destination word	WORD	Yes (16)
				N: Bit number	UINT	---

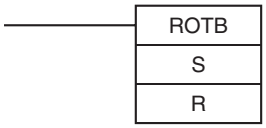
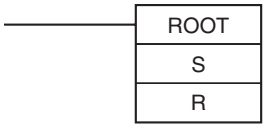
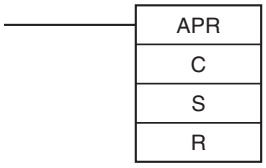
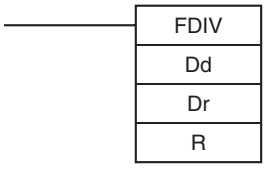
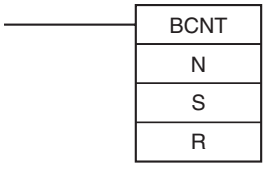
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SIGNED BCD-TO-BINARY	BINS @BINS	470		C: Control word	UINT	---
				S: Source word	WORD	---
				D: Destination word	INT	---
DOUBLE SIGNED BCD-TO-BINARY	BISL @BISL	472		C: Control word	UINT	---
				S: 1st source word	DWORD	---
				D: 1st destination word	DINT	---
SIGNED BINARY-TO-BCD	BCDS @BCDS	471		C: Control word	UINT	---
				S: Source word	INT	---
				D: Destination word	WORD	---
DOUBLE SIGNED BINARY-TO-BCD	BDSL @BDSL	473		C: Control word	UINT	---
				S: 1st source word	DINT	---
				D: 1st destination word	DWORD	---

2-6-11 Logic Instructions

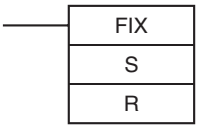
Instruction	Mnemonic	Function code	Symbol	Operand	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
LOGICAL AND	ANDW @ANDW	034		I1: Input 1	WORD	---
				I2: Input 2	WORD	---
				R: Result word	WORD	---
DOUBLE LOGICAL AND	ANDL @ANDL	610		I1: Input 1	DWORD	---
				I2: Input 2	DWORD	---
				R: Result word	DWORD	---

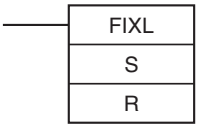
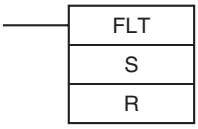
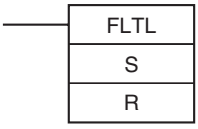
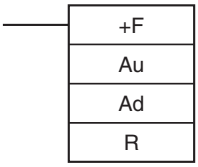
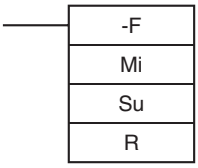
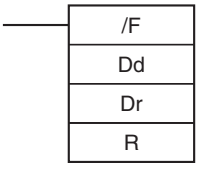
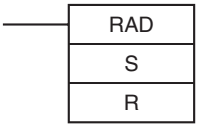
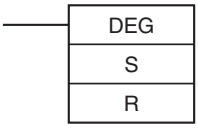
Instruction	Mnemonic	Function code	Symbol	Operand	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
LOGICAL OR	ORW @ORW	035		I1: Input 1	WORD	---
				I2: Input 2	WORD	---
				R: Result word	WORD	---
DOUBLE LOGICAL OR	ORWL @ORWL	611		I1: Input 1	DWORD	---
				I2: Input 2	DWORD	---
				R: Result word	DWORD	---
EXCLUSIVE OR	XORW @XORW	036		I1: Input 1	WORD	---
				I2: Input 2	WORD	---
				R: Result word	WORD	---
DOUBLE EXCLUSIVE OR	XORL @XORL	612		I1: Input 1	DWORD	---
				I2: Input 2	DWORD	---
				R: Result word	DWORD	---
EXCLUSIVE NOR	XNRW @XNRW	037		I1: Input 1	WORD	---
				I2: Input 2	WORD	---
				R: Result word	WORD	---
DOUBLE EXCLUSIVE NOR	XNRL @XNRL	613		I1: Input 1	DWORD	---
				I2: Input 2	DWORD	---
				R: Result word	DWORD	---
COMPLEMENT	COM @COM	029		Wd: Word	WORD	---
DOUBLE COMPLEMENT	COML @COML	614		Wd: Word	DWORD	---

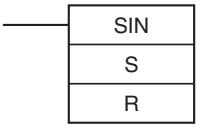
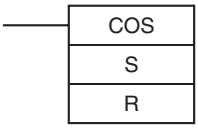
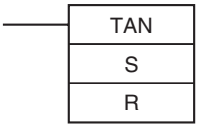
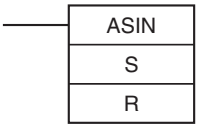
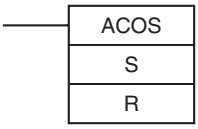
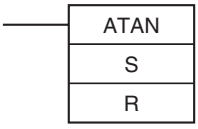
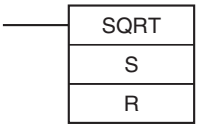
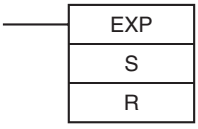
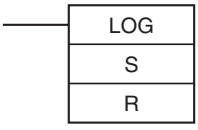
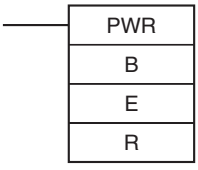
2-6-12 Special Math Instructions

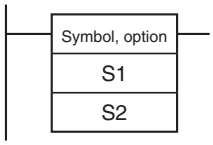
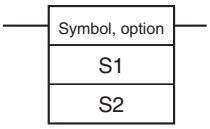
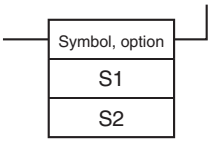
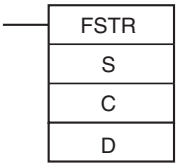
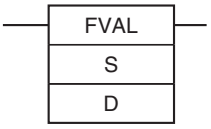
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BINARY ROOT	ROTB @ROTB	620		S: 1st source word	UDINT	---
				R: Result word	UINT	---
BCD SQUARE ROOT	ROOT @ROOT	072		S: 1st source word	DWORD	---
				R: Result word	WORD	---
ARITHMETIC PROCESS	APR @APR	069		C: Control word	UINT	Yes (not fixed)
				S: Source data	WORD	---
				R: Result word	WORD	---
FLOATING POINT DIVIDE	FDIV @FDIV	079		Dd: 1st dividend word	UDINT	---
				Dr: 1st divisor word	UDINT	---
				R: 1st result word	UDINT	---
BIT COUNTER	BCNT @BCNT	067		N: Number of words	UINT	---
				S: 1st source word	UINT	Yes (not fixed)
				R: Result word	UINT	---

2-6-13 Floating-point Math Instructions

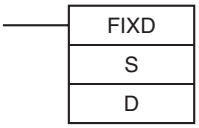
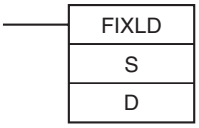
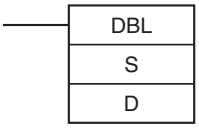
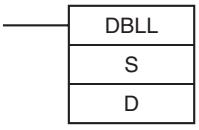
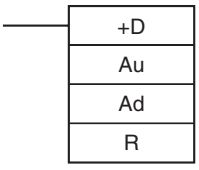
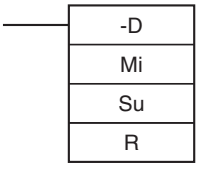
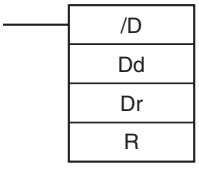
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
FLOATING TO 16-BIT	FIX @FIX	450		S: 1st source word	REAL	---
				R: Result word	INT	---

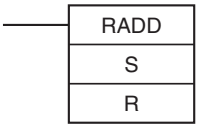
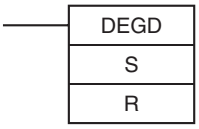
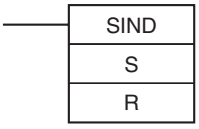
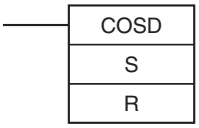
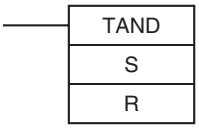
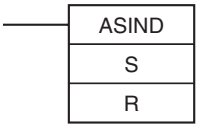
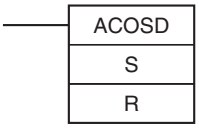
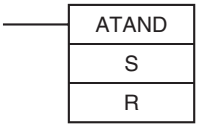
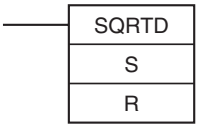
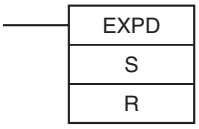
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
FLOATING TO 32-BIT	FIXL @FIXL	451		S: 1st source word	REAL	---
				R: Result word	DINT	---
16-BIT TO FLOATING	FLT @FLT	452		S: Source word	INT	---
				R: 1st result word	REAL	---
32-BIT TO FLOATING	FLTL @FLTL	453		S: 1st source word	DINT	---
				R: Result word	REAL	---
FLOATING-POINT ADD	+F @+F	454		Au: 1st augend word	REAL	---
				Ad: 1st addend word	REAL	---
				R: 1st result word	REAL	---
FLOATING-POINT SUBTRACT	-F @-F	455		Mi: 1st Minuend word	REAL	---
				Su: 1st Subtrahend word	REAL	---
				R: 1st result word	REAL	---
FLOATING- POINT MULTIPLY	*F @*F	456		Md: 1st Multiplicand word	REAL	---
				Mr: 1st Multiplier word	REAL	---
				R: 1st result word	REAL	---
FLOATING- POINT DIVIDE	/F @/F	457		Dd: 1st Dividend word	REAL	---
				Dr: 1st Divisor word	REAL	---
				R: 1st result word	REAL	---
DEGREES TO RADIANS	RAD @RAD	458		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
RADIANS TO DEGREES	DEG @DEG	459		S: 1st source word	REAL	---
				R: 1st result word	REAL	---

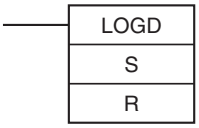
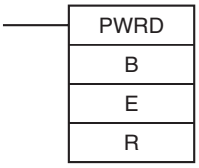
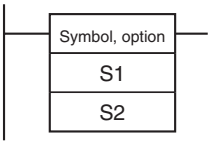
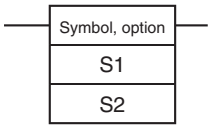
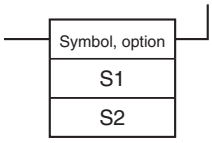
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SINE	SIN @SIN	460		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
COSINE	COS @COS	461		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
TANGENT	TAN @TAN	462		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
ARC SINE	ASIN @ASIN	463		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
ARC COSINE	ACOS @ACOS	464		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
ARC TANGENT	ATAN @ATAN	465		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
SQUARE ROOT	SQRT @SQRT	466		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
EXPONENT	EXP @EXP	467		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
LOGARITHM	LOG @LOG	468		S: 1st source word	REAL	---
				R: 1st result word	REAL	---
EXPONENTIAL POWER	PWR @PWR	840		B: 1st base word	REAL	---
				E: 1st exponent word	REAL	---
				R: 1st result word	REAL	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parenthe-ses.)
Floating Symbol Comparison	LD, AND, OR + =F, <>F, <F, <=F, >F, >=F	329 (=F) 330 (<>F) 331 (<F) 332 (<=F) 333 (>F) 334 (>=F)	Using LD:  Using AND:  Using OR: 	S1:Comparoson data 1	REAL	---
				S2:Comparison data 2	REAL	---
FLOATING- POINT TO ASCII	FSTR @FSTR	448		S: 1st source word	REAL	---
				C: Control word	UINT	Yes (3)
				D: Destination word	UINT	Yes (not fixed)
ASCII TO FLOATING-POINT	FVAL @FVAL	449		S: Source word	UINT	Yes (not fixed)
				D: 1st destination word	REAL	---

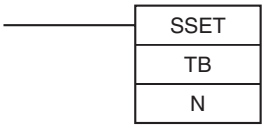
2-6-14 Double-precision Floating-point Instructions

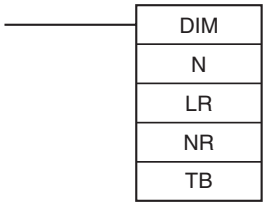
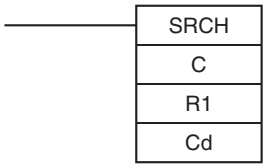
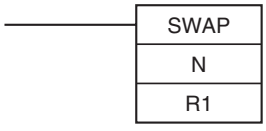
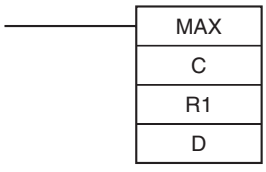
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE FLOATING TO 16-BIT BINARY	FIXD @FIXD	841		S: 1st source word	LREAL	---
				D: Destination word	INT	---
DOUBLE FLOATING TO 32-BIT BINARY	FIXLD @FIXLD	842		S: 1st source word	LREAL	---
				D: 1st destination word	DINT	---
16-BIT BINARY TO DOUBLE FLOATING	DBL @DBL	843		S: Source word	INT	---
				D: 1st destination word	LREAL	---
32-BIT BINARY TO DOUBLE FLOATING	DBLL @DBLL	844		S: 1st source word	DINT	---
				D: 1st destination word	DINT	---
DOUBLE FLOATING-POINT ADD	+D @+D	845		Au: 1st augend word	LREAL	---
				Ad: 1st addend word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE FLOATING-POINT SUBTRACT	-D @-D	846		Mi: 1st minuend word	LREAL	---
				Su: 1st subtra-hend word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE FLOATING-POINT MULTIPLY	*D @*D	847		Md: 1st multipli-cand word	LREAL	---
				Mr: 1st multiplier word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE FLOATING-POINT DIVIDE	/D @/D	848		Dd: 1st Dividend word	LREAL	---
				Dr: 1st divisor word	LREAL	---
				R: 1st result word	LREAL	---

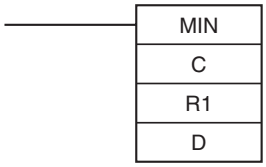
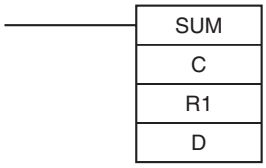
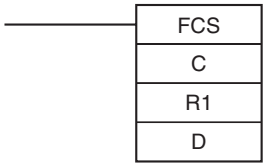
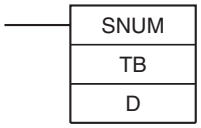
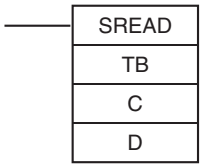
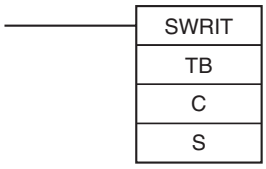
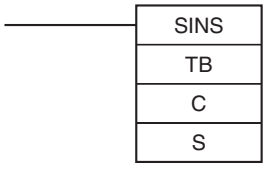
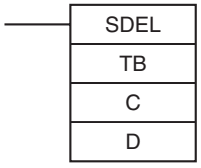
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE DEGREES TO RADIANS	RADD @RADD	849		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE RADIANS TO DEGREES	DEGD @DEGD	850		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE SINE	SIND @SIND	851		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE COSINE	COSD @COSD	852		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE TANGENT	TAND @TAND	853		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE ARC SINE	ASIND @ASIND	854		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE ARC COSINE	ACOSD @ACOSD	855		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE ARC TANGENT	ATAND @ATAND	856		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE SQUARE ROOT	SQRTD @SQRTD	857		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE EXPONENT	EXPD @EXPD	858		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DOUBLE LOGARITHM	LOGD @LOGD	859		S: 1st source word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE EXPONENTIAL POWER	PWRD @PWRD	860		B: 1st base word	LREAL	---
				E: 1st exponent word	LREAL	---
				R: 1st result word	LREAL	---
DOUBLE SYMBOL COMPARISON	LD, AND, OR + =D, <>D, <D, <=D, >D, >=D	335 (=D) 336 (<>D) 337 (<D) 338 (<=D) 339 (>D) 340 (>=D)	Using LD:  Using AND:  Using OR: 	S1:Comparoson data 1	LREAL	---
				S2:Comparison data 2	LREAL	---

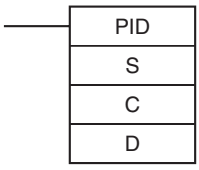
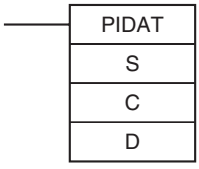
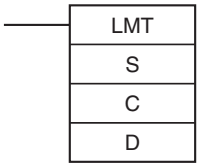
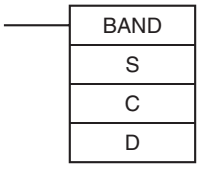
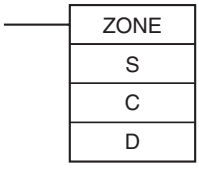
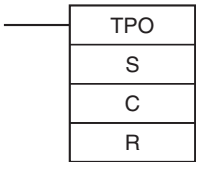
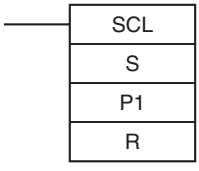
2-6-15 Table Data Processing Instructions

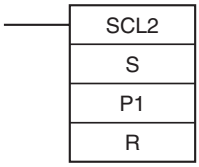
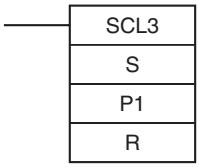
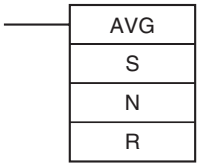
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SET STACK	SSET @SSET	630		TB: 1st stack address	UINT	Yes (not fixed)
				N: Number of words	UINT	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
PUSH ONTO STACK	PUSH @PUSH	632	Not supported in function blocks	TB: 1st stack address	---	---
				S: Source word	---	---
FIRST IN FIRST OUT	FIFO @FIFO	633	Not supported in function blocks	TB: 1st stack address	---	---
				D: Destination word	---	---
LAST IN FIRST OUT	LIFO @LIFO	634	Not supported in function blocks	TB: 1st stack address	---	---
				D: Destination word	---	---
DIMENSION RECORD TABLE	DIM @DIM	631		N: Table number	# + decimal only	---
				LR: Length of each record	UINT	---
				NR: Number of records	UINT	---
				TB: 1st table word	UINT	Yes (not fixed)
SET RECORD LOCATION	SETR @SETR	635	Not supported in function blocks	N: Table number	---	---
				R: Record number	---	---
				D: Destination Index Register	---	---
GET RECORD NUMBER	GETR @GETR	636	Not supported in function blocks	N: Table number	---	---
				IR: Index Register	---	---
				D: Destination word	---	---
DATA SEARCH	SRCH @SRCH	181		C: 1st control word	UDINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
				Cd: Comparison data	WORD	---
SWAP BYTES	SWAP @SWAP	637		N: Number of words	UINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
FIND MAXIMUM	MAX @MAX	182		C: 1st control word	UDINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
				D: Destination word	UINT	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
FIND MINIMUM	MIN @MIN	183		C: 1st control word	UDINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
				D: Destination word	UINT	---
SUM	SUM @SUM	184		C: 1st control word	UDINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
				D: 1st destination word	UDINT	---
FRAME CHECK SUM	FCS @FCS	180		C: 1st control word	UDINT	---
				R1: 1st word in range	UINT	Yes (not fixed)
				D: 1st destination word	UINT	---
STACK SIZE READ	SNUM @SNUM	638		TB: First stack address	UINT	Yes (not fixed)
				D: Destination word	UINT	---
STACK DATA READ	SREAD @SREAD	639		TB: First stack address	UINT	Yes (not fixed)
				C: Offset value	UINT	---
				D: Destination word	UINT	---
STACK DATA OVERWRITE	SWRIT @SWRIT	640		TB: First stack address	UINT	Yes (not fixed)
				C: Offset value	UINT	---
				S: Source data	UINT	---
STACK DATA INSERT	SINS @SINS	641		TB: First stack address	UINT	Yes (not fixed)
				C: Offset value	UINT	---
				S: Source data	UINT	---
STACK DATA DELETE	SDEL @SDEL	642		TB: First stack address	UINT	Yes (not fixed)
				C: Offset value	UINT	---
				D: Destination word	UINT	---

2-6-16 Data Control Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
PID CONTROL	PID	190		S: Input word	UINT	---
				C: 1st parameter word	WORD	Yes (39)
				D: Output word	UINT	---
PID CONTROL WITH AUTO TUNING	PIDAT	191		S: Input word	UINT	---
				C: 1st parameter word	WORD	Yes (40)
				D: Output word	UINT	---
LIMIT CONTROL	LMT @LMT	680		S: Input word	INT	---
				C: 1st limit word	DINT	Yes (2)
				D: Output word	INT	---
DEAD BAND CONTROL	BAND @BAND	681		S: Input word	INT	---
				C: 1st limit word	UINT	Yes (2)
				D: Output word	UINT	---
DEAD ZONE CONTROL	ZONE @ZONE	682		S: Input word	INT	---
				C: 1st limit word	UDINT	Yes (2)
				D: Output word	UINT	---
TIME-PROPORTIONAL OUTPUT	TPO	685		S: Input word	UINT	---
				C: 1st parameter word	WORD	Yes (7)
				R: Pulse Output Bit	BOOL	---
SCALING	SCL @SCL	194		S: Input word	UINT	---
				P1: 1st parameter word	LWORD	Yes (2)
				R: Result word	WORD	---

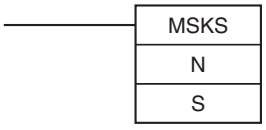
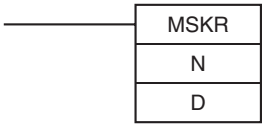
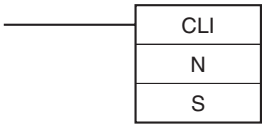
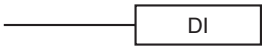
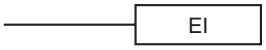
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SCALING 2	SCL2 @SCL2	486		S: Source word	INT	---
				P1: 1st parameter word	WORD	Yes (3)
				R: Result word	WORD	---
SCALING 3	SCL3 @SCL3	487		S: Source word	WORD	---
				P1: 1st parameter word	WORD	Yes (3)
				R: Result word	INT	---
AVERAGE	AVG	195		S: Source word	UINT	---
				N: Number of cycles	UINT	---
				R: Result word	UINT	Yes (not fixed)

2-6-17 Subroutine Instructions

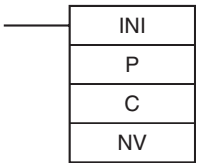
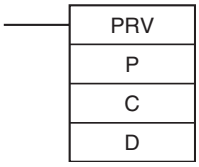
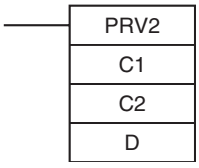
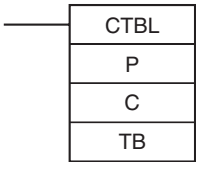
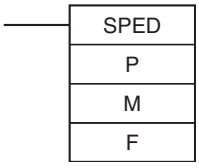
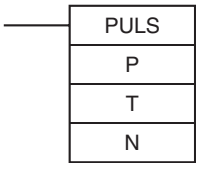
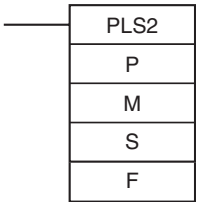
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SUBROUTINE CALL	SBS @SBS	091	Not supported in function blocks	N: Subroutine number	---	---
SUBROUTINE ENTRY	SBN	092	Not supported in function blocks	N: Subroutine number	---	---
SUBROUTINE RETURN	RET	093	Not supported in function blocks		---	---
MACRO	MCRO @MCRO	099	Not supported in function blocks	N: Subroutine number	---	---
				S: 1st input parameter word	---	---
				D: 1st output parameter word	---	---
GLOBAL SUBROUTINE CALL	GSBS @GSBS	750	Not supported in function blocks	N: Subroutine number	---	---

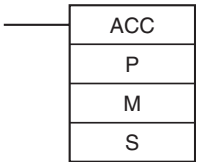
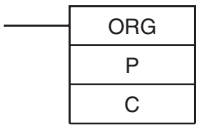
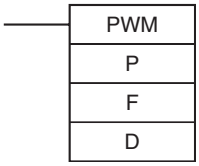
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
GLOBAL SUBROUTINE ENTRY	GSBN	751	Not supported in function blocks	N: Subroutine number	---	---
GLOBAL SUBROUTINE RETURN	GRET	752	Not supported in function blocks		---	---

2-6-18 Interrupt Control Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SET INTERRUPT MASK	MSKS @MSKS	690		N: Interrupt identifier	# + decimal only	---
				S: Interrupt data	UINT	---
READ INTERRUPT MASK	MSKR @MSKR	692		N: Interrupt identifier	# + decimal only	---
				D: Destination word	UINT	---
CLEAR INTERRUPT	CLI @CLI	691		N: Interrupt identifier	# + decimal only	---
				S: Interrupt data	UINT	---
DISABLE INTERRUPTS	DI @DI	693		---	---	---
ENABLE INTERRUPTS	EI	694		---	---	---

2-6-19 High-speed Counter and Pulse Output Instructions (CJ1M-CPU21/22/23 Only)

Instruction	Mnemonic	Function code	Symbol/Operands	Function	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
MODE CONTROL	INI @INI	880		P: Port specifier	WORD	---
				C: Control data	UINT	---
				NV: 1st word with new PCV	DWORD	---
HIGH-SPEED COUNTER PV READ	PRV @PRV	881		P: Port specifier	WORD	---
				C: Control data	UINT	---
				D: 1st destination word	WORD	Yes (1 or 2)
COUNTER FREQUENCY CONVERT (CJ1M CPU Unit Ver. 2.0 or later only)	PRV2	883		C1: Control data	WORD	---
				C2: Pulses/revolution	UINT	---
				D: 1st destination word	UDINT	---
COMPARISON TABLE LOAD	CTBL @CTBL	882		P: Port specifier	UINT	---
				C: Control data	UINT	---
				TB: 1st comparison table word	LWORD	Yes (not fixed)
SPEED OUTPUT	SPED @SPED	885		P: Port specifier	UINT	---
				M: Output mode	WORD	---
				F: 1st pulse frequency word	UDINT	---
SET PULSES	PULS @PULS	886		P: Port specifier	UINT	---
				T: Pulse type	UINT	---
				N: Number of pulses	DINT	---
PULSE OUTPUT	PLS2 @PLS2	887		P: Port specifier	UINT	---
				M: Output mode	WORD	---
				S: 1st word of settings table	WORD	Yes (6)
				F: 1st word of starting frequency	UDINT	---

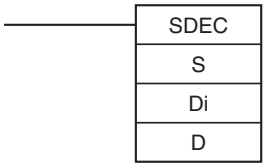
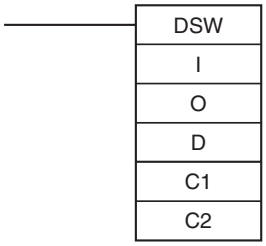
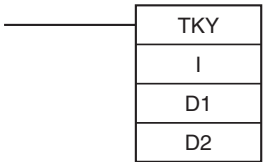
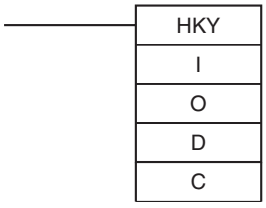
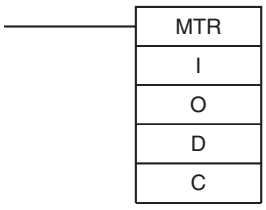
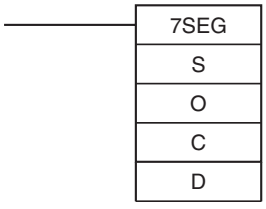
Instruction	Mnemonic	Function code	Symbol/Operands	Function	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
ACCELERATION CONTROL	ACC @ACC	888		P: Port specifier	UINT	---
				M: Output mode	UINT	---
				S: 1st word of settings table	WORD	Yes (3)
ORIGIN SEARCH	ORG @ORG	889		P: Port specifier	UINT	---
				C: Control data	WORD	---
PULSE WITH VARIABLE DUTY FACTOR	PWM @PWM	891		P: Port specifier	UINT	---
				F: Frequency	UINT	---
				D: Duty factor	UINT	---

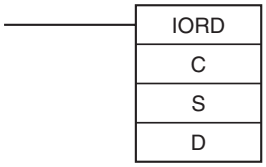
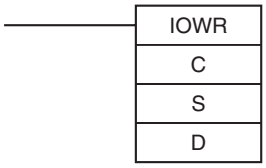
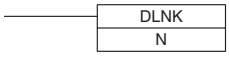
2-6-20 Step Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
STEP DEFINE	STEP	008	Not supported in function blocks	B: Bit	---	---
STEP START	SNXT	009	Not supported in function blocks	B: Bit	---	---

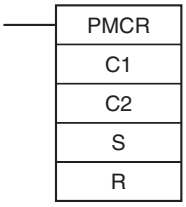
2-6-21 Basic I/O Unit Instructions

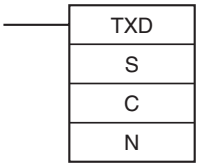
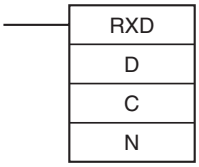
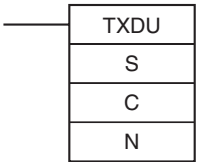
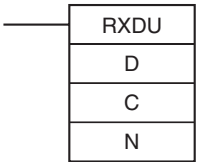
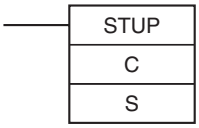
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
I/O REFRESH	IORF @IORF	097	Not supported in function blocks	St: Starting word	---	---
				E: End word	---	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
7-SEGMENT DECODER	SDEC @SDEC	078		S: Source word	UINT	---
				Di: Digit designator	UINT	---
				D: 1st destination word	UINT	Yes (not fixed)
DIGITAL SWITCH INPUT	DSW	210		I: Data input word (D0 to D3)	UINT	---
				O: Output word	UINT	---
				D: 1st result word	WORD	---
				C1: Number of digits	UINT	---
				C2: System word	WORD	---
TEN KEY INPUT	TKY	211		I: Data input word	UINT	---
				D1: 1st register word	UDINT	---
				D2: Key input word	UINT	---
HEXADECIMAL KEY INPUT	HKY	212		I: Data input word	UINT	---
				O: Output word	UINT	---
				D: 1st register word	WORD	Yes (3)
				C: System word	WORD	---
MATRIX INPUT	MTR	213		I: Data input word	UINT	---
				O: Output word	UINT	---
				D: 1st destination word	ULINT	Yes (3)
				C: System word	WORD	---
7-SEGMENT DISPLAY OUTPUT	7SEG	214		S: 1st source word	WORD	Yes (2)
				O: Output word	UINT	---
				C: Control data	UINT	---
				D: System word	WORD	---

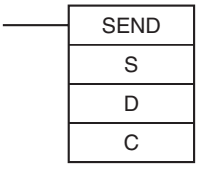
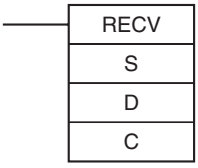
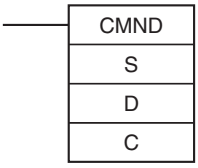
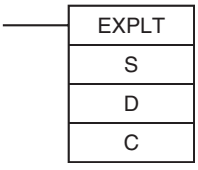
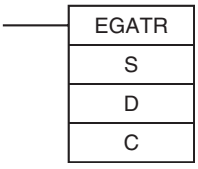
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
INTELLIGENT I/O READ	IORD @IORD	222		C: Control data	UINT	---
				S: Transfer source and number of words	UDINT	Yes (2) UINT must be used when array variables are required.
				D: Transfer destination and number of words	UINT	Yes (not fixed)
INTELLIGENT I/O WRITE	IOWR @IOWR	223		C: Control data	UINT	---
				S: Transfer source and number of words	WORD	Yes (not fixed)
				D: Transfer destination and number of words	UINT	Yes (2) UINT must be used when array variables are required.
CPU BUS UNIT I/O REFRESH	DLNK @DLNK	226		N: Unit number	UINT	---

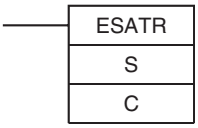
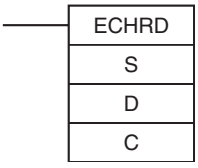
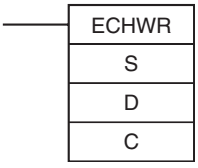
2-6-22 Serial Communications Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
PROTOCOL MACRO	PMCR @PMCR	260		C1:Control word 1	UINT	---
				C2: Control word 2	UINT	---
				S: 1st send word	UINT	Yes (not fixed)
				R: 1st receive word	UINT	Yes (not fixed)

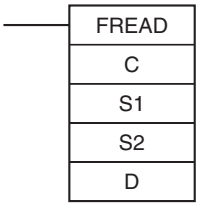
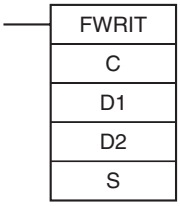
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
TRANSMIT	TXD @TXD	236		S: 1st source word	UINT	Yes (not fixed)
				C: Control word	UINT	---
				N: Number of bytes 0000 to 0100 hex(0 to 256 decimal)	UINT	---
RECEIVE	RXD @RXD	235		D: 1st destination word	UINT	Yes (not fixed)
				C: Control word	UINT	---
				N: Number of bytes to store 0000 to 0100 hex(0 to 256 decimal)	UINT	---
TRANSMIT VIA SERIAL COMMUNICATIONS UNIT	TXDU @TXDU	256		S: 1st source word	UINT	Yes (not fixed)
				C: 1st control word	UDINT	---
				N: Number of send bytes (4 digits BCD)	UINT	---
RECEIVE VIA SERIAL COMMUNICATIONS UNIT	RXDU @RXDU	255		D: 1st destination word	UINT	Yes (not fixed)
				C: 1st control word	UDINT	---
				N: Number of storage bytes	UINT	---
CHANGE SERIAL PORT SETUP	STUP @STUP	237		C: Control word (port)	UINT	---
				S: 1st source word	UINT	Yes (not fixed)

2-6-23 Network Instructions

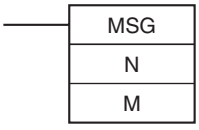
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
NETWORK SEND	SEND @SEND	090		S: 1st source word	UINT	Yes (not fixed)
				D: 1st destination word	UINT	Specify address at remote node with AT setting.
				C: 1st control word	WORD	Yes (5)
NETWORK RECEIVE	RECV @RECV	098		S: 1st source word	UINT	Specify address at remote node with AT setting.
				D: 1st destination word	UINT	Yes (not fixed)
				C: 1st control word	WORD	Yes (5)
DELIVER COMMAND	CMND @CMND	490		S: 1st command word	UINT	Yes (not fixed)
				D: 1st response word	UINT	Yes (not fixed)
				C: 1st control word	WORD	Yes (6)
EXPLICIT MESSAGE SEND	EXPLT	720		S: 1st word of send message	WORD	Yes (not fixed)
				D: 1st word of received message	WORD	Yes (not fixed)
				C: 1st control word	LWORD	Yes (4) WORD must be used when array variables are required.
EXPLICIT GET ATTRIBUTE	EGATR	721		S: 1st word of send message	ULINT	Yes (4) WORD must be used when array variables are required.
				D: 1st word of received message	WORD	Yes (not fixed)
				C: 1st control word message	LWORD	Yes (4)

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
EXPLICIT SET ATTRIBUTE	ESATR	722		S: 1st word of send message	WORD	Yes (not fixed)
				C: 1st control word	WORD	Yes (3)
EXPLICIT WORD READ	ECHRD	723		S: 1st source word in remote CPU Unit	UINT	Specify the destination address when using AT settings.
				D: 1st destination word in local CPU Unit	UINT	Yes (2)
				C: 1st control word	WORD	Yes (5)
EXPLICIT WORD WRITE	ECHWR	724		S: 1st source word in local CPU Unit	UINT	Yes (not fixed)
				D: 1st destination word in remote CPU Unit	UINT	Specify the destination address when using AT settings.
				C: 1st control word	WORD	Yes (5)

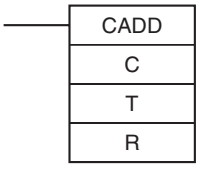
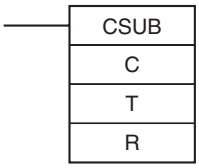
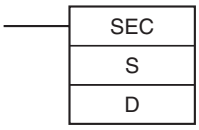
2-6-24 File Memory Instructions

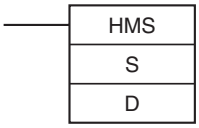
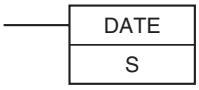
Instruction	Mnemonic	Function code	Symbol	Operand	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
READ DATA FILE	FREAD @FREAD	700		C: Control word	UINT	---
				S1: 1st source word	LWORD	Yes (2) DWORD must be used when array variables are required.
				S2: Filename	UINT	Yes (39)
				D: 1st destination word	UINT	Yes (not fixed)
WRITE DATA FILE	FWRIT @FWRIT	701		C: Control word	UINT	---
				D1: 1st destination word	LWORD	Yes (2) DWORD must be used when array variables are required.
				D2: Filename	UINT	Yes (39)
				S: 1st source word	UINT	Yes (not fixed)

2-6-25 Display Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
DISPLAY MESSAGE	MSG @MSG	046		N: Message number	UINT	---
				M: 1st message word	UINT	Yes (16)

2-6-26 Clock Instructions

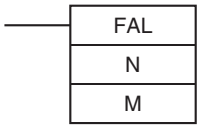
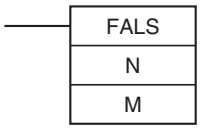
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
CALENDAR ADD	CADD @CADD	730		C: 1st calendar word	WORD	Yes (3)
				T: 1st time word	DWORD	Yes (2) WORD must be used when array variables are required.
				R: 1st result word	WORD	Yes (3)
CALENDAR SUBTRACT	CSUB @CSUB	731		C: 1st calendar word	WORD	Yes (3)
				T: 1st time word	DWORD	Yes (2) WORD must be used when array variables are required.
				R: 1st result word	WORD	Yes (3)
HOURS TO SECONDS	SEC @SEC	065		S: 1st source word	DWORD	Yes (2) WORD must be used when array variables are required.
				D: 1st destination word	DWORD	Yes (2) WORD must be used when array variables are required.

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SECONDS TO HOURS	HMS @HMS	066		S: 1st source word	DWORD	Yes (2) WORD must be used when array variables are required.
				D: 1st destination word	DWORD	Yes (2) WORD must be used when array variables are required.
CLOCK ADJUSTMENT	DATE @DATE	735		S: 1st source word	LWORD	Yes (4) WORD must be used when array variables are required.

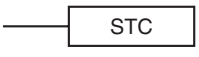
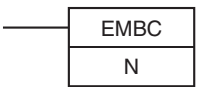
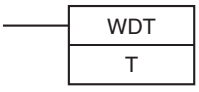
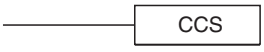
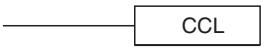
2-6-27 Debugging Instructions

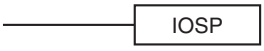
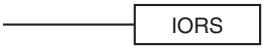
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
TRACE MEMORY SAMPLING	TRSM	045		---	---	---

2-6-28 Failure Diagnosis Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
FAILURE ALARM	FAL @FAL	006		N: FAL number	# + decimal only	---
				M: 1st message word or error code to generate(#0000 to #FFFF)	WORD	---
SEVERE FAILURE ALARM	FALS	007		N: FALS number	# + decimal only	---
				M: 1st message word or error code to generate(#0000 to #FFFF)	WORD	---
FAILURE POINT DETECTION	FPD	269	Not supported in function blocks	C: Control word	---	---
				T: Monitoring time	---	---
				R: 1st register word	---	---

2-6-29 Other Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
SET CARRY	STC @STC	040		---	---	---
CLEAR CARRY	CLC @CLC	041		---	---	---
SELECT EM BANK	EMBC @EMBC	281		N: EM bank number.	UINT	---
EXTEND MAXIMUM CYCLE TIME	WDT @WDT	094		T: Timer setting	Constants only	---
SAVE Condition FlagS	CCS @CCS	282		---	---	---
LOAD Condition FlagS	CCL @CCL	283		---	---	---

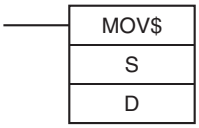
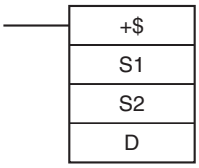
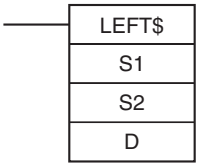
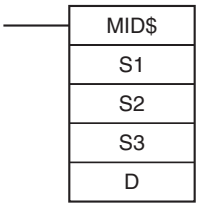
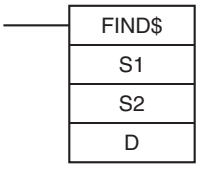
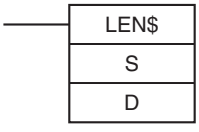
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
CONVERT ADDRESS FROM CV	FRMCV @FRMCV	284	Not supported in function blocks	S: Word containing CV-series memory address	---	---
				D: Destination Index Register	---	---
CONVERT ADDRESS TO CV	TOCV @TOCV	285	Not supported in function blocks	S: Index Register containing CS Series memory address	---	---
				D: Destination word	---	---
DISABLE PERIPHERAL SERVICING	IOSP @IOSP	287		---	---	---
ENABLE PERIPHERAL SERVICING	IORS	288		---	---	---

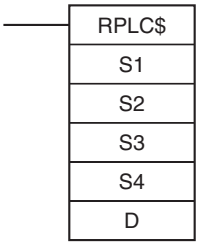
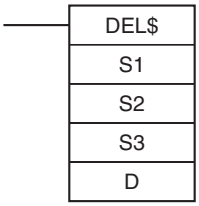
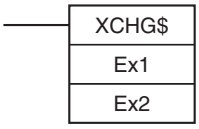
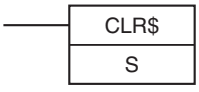
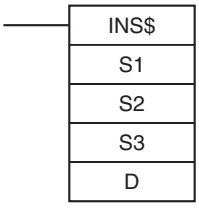
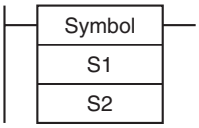
2-6-30 Block Programming Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BLOCK PROGRAM BEGIN	BPRG	096	Not supported in function blocks	N: Block program number	---	---
BLOCK PROGRAM END	BEND	801	Not supported in function blocks	---	---	---
BLOCK PROGRAM PAUSE	BPPS	811	Not supported in function blocks	N: Block program number	---	---
BLOCK PROGRAM RESTART	BPRS	812	Not supported in function blocks	N: Block program number	---	---
CONDITIONAL BLOCK EXIT	CONDITION EXIT	806	Not supported in function blocks	---	---	---
CONDITIONAL BLOCK EXIT	EXIT Bit operand	806	Not supported in function blocks	B: Bit operand	---	---
CONDITIONAL BLOCK EXIT (NOT)	EXIT NOT Bit operand	806	Not supported in function blocks	B: Bit operand	---	---
CONDITIONAL BLOCK BRANCHING	CONDITION IF	802	Not supported in function blocks	---	---	---
CONDITIONAL BLOCK BRANCHING	IF Bit operand	802	Not supported in function blocks	B: Bit operand	---	---
CONDITIONAL BLOCK BRANCHING (NOT)	IF NOT Bit operand	802	Not supported in function blocks	B: Bit operand	---	---

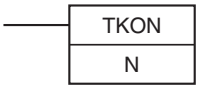
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
CONDITIONAL BLOCK BRANCHING (ELSE)	ELSE	803	Not supported in function blocks	---	---	---
CONDITIONAL BLOCK BRANCHING END	IEND	804	Not supported in function blocks	---	---	---
ONE CYCLE AND WAIT	CONDITION WAIT	805	Not supported in function blocks	---	---	---
ONE CYCLE AND WAIT	WAIT Bit operand	805	Not supported in function blocks	B: Bit operand	---	---
ONE CYCLE AND WAIT (NOT)	WAIT NOT Bit operand	805	Not supported in function blocks	B: Bit operand	---	---
TIMER WAIT	TIMW (BCD)	813	Not supported in function blocks	N: Timer number	---	---
				SV: Set value	---	---
	TIMWX (BIN)	816	Not supported in function blocks	N: Timer number	---	---
				SV: Set value	---	---
COUNTER WAIT	CNTW (BCD)	814	Not supported in function blocks	N: Counter number	---	---
				SV: Set value	---	---
				I: Count input	---	---
	CNTWX (BIN)	817	Not supported in function blocks	N: Counter number	---	---
				SV: Set value	---	---
				I: Count input	---	---
HIGH-SPEED TIMER WAIT	TMHW (BCD)	815	Not supported in function blocks	N: Timer number	---	---
				SV: Set value	---	---
	TMHWX (BIN)	818	Not supported in function blocks	N: Timer number	---	---
				SV: Set value	---	---
LOOP	LOOP	809	Not supported in function blocks	---	---	---
LEND	LEND	810	Not supported in function blocks	---	---	---
LEND	LEND Bit operand	810	Not supported in function blocks	B: Bit operand	---	---
LEND NOT	LEND NOT Bit operand	810	Not supported in function blocks	B: Bit operand	---	---

2-6-31 Text String Processing Instructions

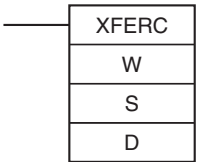
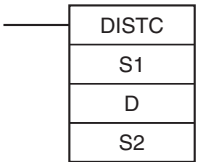
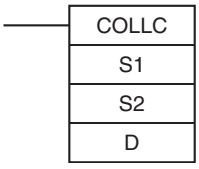
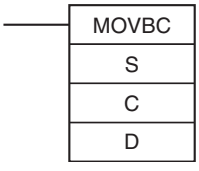
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
MOV STRING	MOV$ @MOV$	664		S: 1st source word	UINT	Yes (not fixed)
				D: 1st destination word	UINT	Yes (not fixed)
CONCATENATE STRING	+$ @+$	656		S1: Text string 1	INT	Yes (not fixed)
				S2: Text string 2	INT	Yes (not fixed)
				D: First destination word	INT	Yes (not fixed)
GET STRING LEFT	LEFT$ @LEFT$	652		S1: Text string first word	UINT	Yes (not fixed)
				S2: Number of characters	UINT	---
				D: First destination word	UINT	Yes (not fixed)
GET STRING RIGHT	RGHT$ @RGHT$	653		S1: Text string first word	UINT	Yes (not fixed)
				S2: Number of characters	UINT	---
				D: First destination word	UINT	Yes (not fixed)
GET STRING MIDDLE	MID$ @MID$	654		S1: Text string first word	UINT	Yes (not fixed)
				S2: Number of characters	UINT	---
				S3: Beginning position	UINT	---
				D: First destination word	UINT	Yes (not fixed)
FIND IN STRING	FIND$ @FIND$	660		S1: Source text string first word	UINT	Yes (not fixed)
				S2: Found text string first word	UINT	Yes (not fixed)
				D: First destination word	UINT	---
STRING LENGTH	LEN$ @LEN$	650		S: Text string first word	UINT	Yes (not fixed)
				D: 1st destination word	UINT	---

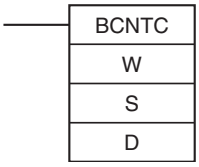
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
REPLACE IN STRING	RPLC$ @RPLC$	661		S1: Text string first word	UINT	Yes (not fixed)
				S2: Replacement text string first word	UINT	Yes (not fixed)
				S3: Number of characters	UINT	---
				S4: Beginning position	UINT	---
				D: First destination word	UINT	Yes (not fixed)
DELETE STRING	DEL$ @DEL$	658		S1: Text string first word	UINT	Yes (not fixed)
				S2: Number of characters	UINT	---
				S3: Beginning position	UINT	---
				D: First destination word	UINT	Yes (not fixed)
EXCHANGE STRING	XCHG$ @XCHG$	665		Ex1: 1st exchange word 1	UINT	Yes (not fixed)
				Ex2: 1st exchange word 2	UINT	Yes (not fixed)
CLEAR STRING	CLR$ @CLR$	666		S: Text string first word	UINT	Yes (not fixed)
INSERT INTO STRING	INS$ @INS$	657		S1: Base text string first word	UINT	Yes (not fixed)
				S2: Inserted text string first word	UINT	Yes (not fixed)
				S3: Beginning position	UINT	---
				D: First destination word	UINT	Yes (not fixed)
String Comparison	LD,AND, OR + =$,<>$,<$,<=$,>$,>=$	670 (=$) 671 (<>$) 672 (<$) 673 (<=$) 674 (>$) 675 (>=$)		S1: Text string 1	UINT	Yes (not fixed)
				S2: Text string 2	UINT	Yes (not fixed)

2-6-32 Task Control Instructions

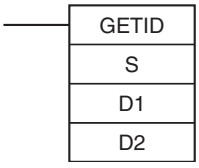
Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
TASK ON	TKON @TKON	820		N: Task number	# + decimal only	---
TASK OFF	TKOF @TKOF	821		N: Task number	# + decimal only	---

2-6-33 Model Conversion Instructions

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BLOCK TRANSFER	XFERC @XFERC	565		W: Number of words (in BCD)	WORD	---
				S: 1st source word	WORD	Yes (not fixed)
				D: 1st destination word	WORD	Yes (not fixed)
SINGLE WORD DISTRIBUTE	DISTC @DISTC	566		S1: Source word	WORD	---
				D: Destination base address	WORD	Yes (not fixed)
				S2: Offset (in BCD)	WORD	---
DATA COLLECT	COLLC @COLLC	567		S1: Source base address	WORD	Yes (not fixed)
				S2: Offset (in BCD)	WORD	---
				D: Destination word	WORD	---
MOVE BIT	MOVBC @MOVBC	568		S: Source word or data	WORD	---
				C: Control word (in BCD)	WORD	---
				D: Destination word	WORD	---

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
BIT COUNTER	BCNTC @BCNTC	621		W: Number of words (in BCD)	WORD	---
				S: 1st source word	UINT	Yes (not fixed)
				D: Result word	WORD	---

2-6-34 Special Instructions for Function Blocks

Instruction	Mnemonic	Function code	Symbol	Operands	Supported variable data types	AT setting or array variable required (Required word data size shown in parentheses.)
GET VARIABLE ID	GETID @GETIC	286		S: Source variable or address	WORD	---
				D1: Variable type (I/O memory area) code	WORD	---
				D2: Word address	WORD	---

2-7 CPU Unit Function Block Specifications

The specifications of the functions blocks used in CS/CJ-series CS1-H, CJ1-H, and CJ1M CPU Units with version 3.0 or later are as follows. Refer to the other Operation Manuals for the CS/CJ Series for other specifications.

2-7-1 Specifications

CS1-H CPU Unit

Item	Specification								
Model	CS1H-CPU67H	CS1H-CPU66H	CS1H-CPU65H	CS1H-CPU64H	CS1H-CPU63H	CS1G-CPU45H	CS1G-CPU44H	CS1G-CPU43H	CS1G-CPU42H
I/O bits	5,120						1,280	960	
Program capacity (steps)	250K	120K	60K	30K	20K	60K	30K	20K	10K
Data memory	32K words								

Item		Specification								
Extended Data Memory		32K words × 13 banks E0_00000 to EC_32767	32K words × 7 banks E0_00000 to E6_32767	32K words × 3 banks E0_00000 to E2_32767	32K words × 1 bank E0_00000 to E0_32767		32K words × 3 banks E0_00000 to E2_32767	32K words × 1 bank E0_00000 to E0_32767		
Function blocks	Maximum number of definitions	1,024	1,024	1,024	1,024	128	1,024	1,024	128	128
	Maximum number of instances	2,048	2,048	2,048	2,048	256	2,048	2,048	256	256
Flash memory	Function block program memory (Kbytes)	1,664	1,664	1,024	512	512	1,024	512	512	512
	Comment files (Kbytes)	128	128	64	64	64	64	64	64	64
	Program index files (Kbytes)	128	128	64	64	64	64	64	64	64
	Variable tables (Kbytes)	128	128	128	64	64	128	64	64	64

CJ1-H CPU Unit

Item		Specification						
Model		CJ1H-CPU67H	CJ1H-CPU66H	CJ1H-CPU65H	CJ1G-CPU45H	CJ1G-CPU44H	CJ1G-CPU43H	CJ1G-CPU42H
I/O bits		2,560			1,280		960	
Program capacity (steps)		250K	120K	60K	60K	30K	20K	10K
Data memory		32K words						
Extended Data Memory		32K words × 13 banks E0_00000 to EC_32767	32K words × 7 banks E0_00000 to E6_32767	32K words × 3 banks E0_00000 to E2_32767	32K words × 3 banks E0_00000 to E2_32767	32K words × 1 bank E0_00000 to E0_32767		
Function blocks	Maximum number of definitions	1,024	1,024	1,024	1,024	1,024	128	128
	Maximum number of instances	2,048	2,048	2,048	2,048	2,048	256	256

Item		Specification						
Flash memory	Function block program memory (Kbytes)	1,664	1,664	1,024	1,024	512	512	512
	Comment files (Kbytes)	128	128	64	64	64	64	64
	Program index files (Kbytes)	128	128	64	64	64	64	64
	Variable tables (Kbytes)	128	128	128	128	64	64	64

CJ1M CPU Unit

Item	Specification					
	Units with internal I/O functions			Units without internal I/O functions		
Model	CJ1M-CPU23	CJ1M-CPU22	CJ1M-CPU21	CJ1M-CPU13	CJ1M-CPU12	CJ1M-CPU11
I/O bits	640	320	160	640	320	160
Program capacity (steps)	20K	10K	5K	20K	10K	5K
Number of Expansion Racks	1 max.	Expansion not supported		1 max.	Expansion not supported	
Data memory	32K words					
Extended Data Memory	None					
Pulse start times	46 μs (without acceleration/deceleration) 70 μs (with acceleration/deceleration)		63 μs (without acceleration/deceleration) 100 μs (with acceleration/deceleration)	---		
Number of scheduled interrupts	2		1	2		1
PWM outputs	2		1	None		
Maximum value of subroutine number	1,024		256	1,024		256
Maximum value of jump number in JMP instruction	1,024		256	1,024		256
Internal inputs	10 points • 4 interrupt inputs (pulse catch) • 2 high-speed counter inputs (50-kHz phase difference or 100-kHz single-phase)			---		
Internal outputs	6 points • 2 pulse outputs (100 kHz) • 2 PWM outputs		6 points • 2 pulse outputs (100 kHz) • 1 PWM output	---		

Item		Specification	
		Units with internal I/O functions	Units without internal I/O functions
Function blocks	Maximum number of definitions	128	
	Maximum number of instances	256	
Flash memory	Function block program memory (Kbytes)	256	
	Comment files (Kbytes)	64	
	Program index files (Kbytes)	64	
	Variable tables (Kbytes)	64	

2-7-2 Operation of Timer Instructions

There is an option called *Apply the same spec as TO-2047 to T2048-4095* in the PLC properties of CPU Units. This setting affects the operation of timers as described in this section.

Selecting the Option (Default)

If this option is selected, all timers will operate the same regardless of timer number, as shown in the following table.

Timer Operation for Timer Numbers T0000 to T4095

Refresh	Description
When instruction is executed	The PV is refreshed each time the instruction is executed. If the PV is 0, the Completion Flag is turned ON. If it is not 0, the Completion Flag is turned OFF.
When execution of all tasks is completed	All PV are refreshed once each cycle.
Every 80 ms	If the cycle time exceeds 80 ms, all PV are refreshed once every 80 ms.

Not Selecting the Option

If this option is not selected, the refreshing of timer instructions with timer numbers T0000 to T2047 will be different from those with timer numbers T2048 to T4095, as given below. This behavior is the same for CPU Units that do not support function blocks. (Refer to the descriptions of individual instruction in the *CS/CJ Series Instruction Reference* for details.)

Timer Operation for Timer Numbers T0000 to T2047

Refresh	Description
When instruction is executed	The PV is refreshed each time the instruction is executed. If the PV is 0, the Completion Flag is turned ON. If it is not 0, the Completion Flag is turned OFF.
When execution of all tasks is completed	All PV are refreshed once each cycle.
Every 80 ms	If the cycle time exceeds 80 ms, all PV are refreshed once every 80 ms.

Timer Operation for Timer Numbers T2048 to T4095

Refresh	Description
When instruction is executed	The PV is refreshed each time the instruction is executed. If the PV is 0, the Completion Flag is turned ON. If it is not 0, the Completion Flag is turned OFF
When execution of all tasks is completed	PV are not updated.
Every 80 ms	PV are not updated even if the cycle time exceeds 80 ms.

Select the *Apply the same spec as TO-2047 to T2048-4095* Option to ensure consistent operation when using the timer numbers allocated by default to function block variables (T3072 to T4095).

2-8 Number of Function Block Program Steps and Instance Execution Time

2-8-1 Number of Function Block Program Steps (CPU Units with Unit Version 3.0 or Later)

Use the following equation to calculate the number of program steps when function block definitions have been created and the instances copied into the user program using CS/CJ-series CPU Units with unit version 3.0 or later.

Number of steps

= Number of instances × (Call part size m + I/O parameter transfer part size n × Number of parameters) + Number of instruction steps in the function block definition p (See note.)

Note The number of instruction steps in the function block definition (p) will not be diminished in subsequence instances when the same function block definition is copied to multiple locations (i.e., for multiple instances). Therefore, in the above equation, the number of instances is not multiplied by the number of instruction steps in the function block definition (p).

Contents			CS/CJ-series CPU Units with unit version 3.0 or later
m	Call part		57 steps
n	I/O parameter transfer part The data type is shown in parentheses.	1-bit I/O variable (BOOL)	6 steps
		1-word I/O variable (INT, UINT, WORD)	6 steps
		2-word I/O variable (DINT, UDINT, DWORD, REAL)	6 steps
		4-word I/O variable (LINT, ULINT, LWORD, LREAL)	12 steps
p	Number of instruction steps in function block definition	The total number of instruction steps (same as standard user program) + 27 steps.	

Example:
Input variables with a 1-word data type (INT): 5
Output variables with a 1-word data type (INT): 5
Function block definition section: 100 steps
Number of steps for 1 instance = 57 + (5 + 5) × 6 steps + 100 steps + 27 steps = 244 steps

2-8-2 Function Block Instance Execution Time (CPU Units with Unit Version 3.0 or Later)

Use the following equation to calculate the effect of instance execution on the cycle time when function block definitions have been created and the instances copied into the user program using CS/CJ-series CPU Units with unit version 3.0 or later.

Effect of Instance Execution on Cycle Time
= Startup time (A)
+ I/O parameter transfer processing time (B)
+ Execution time of instructions in function block definition (C)

The following table shows the length of time for A, B, and C.

Operation			CPU Unit model		
			CS1H-CPU6□H CJ1H-CPU6□H	CS1G-CPU4□H CJ1G-CPU4□H	CJ1M-CPU□□
A	Startup time	Startup time not including I/O parameter transfer	6.8 μs	8.8 μs	15.0 μs
B	I/O parameter transfer processing time The data type is indicated in parentheses.	1-bit I/O variable (BOOL)	0.4 μs	0.7 μs	1.0 μs
		1-word I/O variable (INT, UINT, WORD)	0.3 μs	0.6 μs	0.8 μs
		2-word I/O variable (DINT, UDINT, DWORD, REAL)	0.5 μs	0.8 μs	1.1 μs
		4-word I/O variable (LINT, ULINT, LWORD, LREAL)	1.0 μs	1.6 μs	2.2 μs
C	Function block definition instruction execution time	Total instruction processing time (same as standard user program)			

Example: CS1H-CPU63H
Input variables with a 1-word data type (INT): 3
Output variables with a 1-word data type (INT): 2
Total instruction processing time in function block definition section: 10 μs
Execution time for 1 instance = 6.8 μs + (3 + 2) × 0.3 μs + 10 μs = 18.3 μs

Note The execution time is increased according to the number of multiple instances when the same function block definition has been copied to multiple locations.

SECTION 3
Creating Function Blocks

This section describes the procedures for creating function blocks on the CX-Programmer.

3-1 Procedural Flow 112
3-2 Procedures 114
3-2-1 Creating a Project 114
3-2-2 Creating a New Function Block Definition 114
3-2-3 Defining Function Blocks Created by User 117
3-2-4 Creating Instances from Function Block Definitions 124
3-2-5 Setting Function Block Parameters Using the P Key 125
3-2-6 Setting the FB Instance Areas. 128
3-2-7 Checking Internal Address Allocations for Variables 129
3-2-8 Copying and Editing Function Block Definitions 131
3-2-9 Checking the Source Function Block Definition from an Instance . 131
3-2-10 Checking the Size of the Function Block Definition 131
3-2-11 Compiling Function Block Definitions (Checking Program). 131
3-2-12 Printing Function Block Definition 132
3-2-13 Saving and Reusing Function Block Definition Files 133
3-2-14 Downloading/Uploading Programs to the Actual CPU Unit 134
3-2-15 Monitoring and Debugging Function Blocks 134

3-1 Procedural Flow

The following procedures are used to create function blocks, save them in files, transfer them to the CPU Unit, monitor them, and debug them.

Creating Function Blocks

Create a Project

Refer to *3-2-1 Creating a Project* for details.

■ **Creating a New Project**

1,2,3...

1. Start the CX-Programmer and select ***New*** from the File Menu.
2. Select a *Device type*: CS1G-H, CS1H-H, CJ1G-H, CJ1H-H, or CJ1M.

■ **Reusing an Existing CX-Programmer Project**

1,2,3...

1. Start the CX-Programmer, and read the existing project file (.cxp) created using CX-Programmer Ver. 4.0 or earlier by selecting the file from the File Menu.
2. Select a *Device type*: CS1H-H, CS1G-H, CJ1G-H, CJ1H-H, or CJ1M.

Create a Function Block Definition

Refer to *3-2-2 Creating a New Function Block Definition* for details.

1,2,3...

1. Select *Function Blocks* in the project workspace and right-click.
2. Select ***Insert Function Block - Ladder*** or ***Insert Function Blocks - Structured Text*** from the popup menu.

Define the Function Block

Refer to *3-2-3 Defining Function Blocks Created by User* for details.

■ **Registering Variables before Inputting the Ladder Program or ST Program**

1,2,3...

1. Register variables in the variable table.
2. Create the ladder program or ST program.

■ **Registering Variables as Necessary while Inputting the Ladder Program or ST Program**

1,2,3...

1. Create the ladder program or ST program.
2. Register a variable in the variable table whenever required.

Create an Instance from the Function Block Definition

Refer to *3-2-4 Creating Instances from Function Block Definitions* for details.

■ **Inserting Instances in the Ladder Section Window and then Inputting the Instance Name**

1,2,3...

1. Place the cursor at the location at which to create an instance (i.e., a copy) of the function block and press the **F** Key.
2. Input the name of the instance.
3. Select the function block definition to be copied.

■ **Registering Instance Names in the Global Symbol Table and then Selecting the Instance Name when Inserting**

1,2,3...

1. Select *Function Block* as the data type for the variable in the global symbol table.
2. Press the **F** Key in the Ladder Section Window.
3. Select the name of the instance that was registered from the pull-down menu on the *FB Instance* Field.

Allocate External I/O to the Function Block

Refer to *3-2-5 Setting Function Block Parameters Using the P Key* for details.

1,2,3...

1. Place the cursor at the position of the input variable or output variable and press the **P** Key.
2. Input the source address for the input variable or the destination address for the output variable.

Set the Function Block Memory Allocations (Instance Areas)

Refer to *3-2-6 Setting the FB Instance Areas* for details.

1,2,3...

1. Select the instance and select ***Function Block Memory - Function Block Memory Allocation*** from the PLC Menu.
2. Set the function block memory allocations.

Printing, Saving, and Reusing Function Block Files

Compile the Function Block Definition and Save It as a Library File

Refer to *3-2-11 Compiling Function Block Definitions (Checking Program)* and *3-2-13 Saving and Reusing Function Block Definition Files* for details.

1,2,3...

1. Compile the function block that has been saved.
2. Print the function block.
3. Save the function block as a function block definition file (.cxf).
4. Read the file into another PLC project.

Transferring the Program to the PLC

Refer to *3-2-14 Downloading/Uploading Programs to the Actual CPU Unit*.

Monitoring and Debugging the Function Block

Refer to *3-2-15 Monitoring and Debugging Function Blocks*.

3-2 Procedures

3-2-1 Creating a Project

Creating New Projects with CX-Programmer

1,2,3...

1. Start the CX-Programmer and select ***New*** from the File Menu.
2. In the Change PLC Window, select a *Device Type* that supports function blocks. These are listed in the following table.

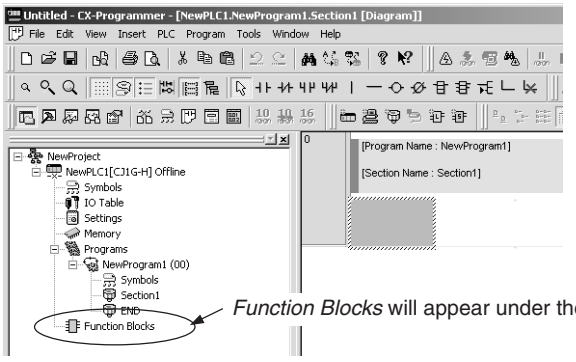
Device	CPU
CS1G-H	CPU42H/43H/44H/45H
CS1H-H	CPU63H/64H/65H/66H/67H
CJ1G-H	CPU42H/43H/44H/45H
CJ1H-H	CPU65H/66H/67H
CJ1M	CPU11/12/13/21/22/23

Press the Settings Button and select the *CPU Type.* For details on other settings, refer to the *CX-Programmer Ver. 5.0 Operation Manual (W414).*

3-2-2 Creating a New Function Block Definition

1,2,3...

1. When a project is created, a *Function Blocks* icon will appear in the project workspace as shown below.



2. Function block definitions are created by inserting function block definitions after the Function Blocks icon.

Creating Function Block Definitions

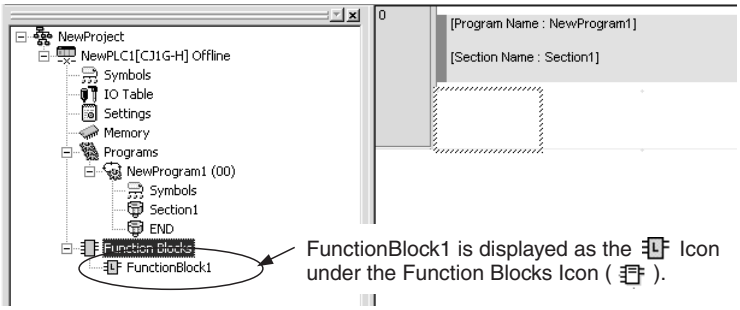
Function blocks can be defined by the user using either ladder programming or structured text.

Creating (Inserting) Function Block Definitions with Ladders

1. Select ***Function Blocks*** in the project workspace, right-click, and select ***Insert Function Blocks - Ladder*** from the popup menu. (Or select ***Function Block - Ladder*** from the Insert Menu.)

Creating (Inserting) Function Block Definitions with Structured Text

1. Select ***Function Blocks*** in the project workspace, right-click, and select ***Insert Function Blocks - Structured Text*** from the popup menu. (Or select ***Function Block - Structured Text*** from the Insert Menu.)

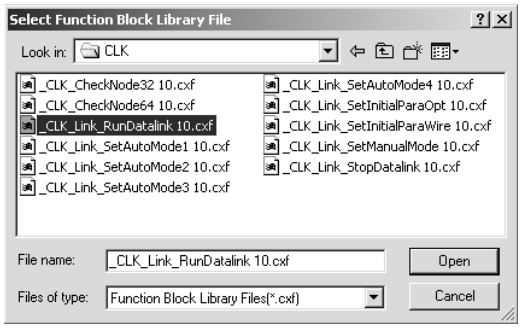


2. A function block called FunctionBlock1 will be automatically inserted either after the for ladder programming language (default) or the for ST language. This icon contains the definitions for the newly created (inserted) function block.
3. Whenever a function block definition is created, the name FunctionBlock□ will be assigned automatically, where □ is a serial number. These names can be changed. All names must contain no more than 64 characters.

Using OMRON FB Library Files

Use the following procedure to insert OMRON FB Library files (.cxf).

1. Select ***Function Blocks*** in the project workspace, right-click, and select ***Insert Function Blocks - Library File*** from the popup menu. (Or select ***Function Block - Library File*** from the Insert Menu.)
2. The following Select Function Block Library File Dialog Box will be displayed.



Note To specify the default folder (file location) in the Function Block Library File Dialog Box, select ***Tools - Options***, select the **General** Tab and the select the default file in the *OMRON FB library storage location* field.

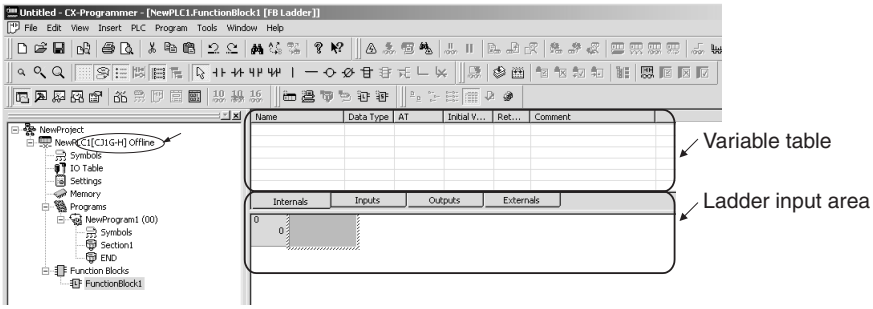
3. Specify the folder in which the OMRON FB Library file is located, select the library file, and click the **Open** Button. The library file will be inserted as a function block definition after the .

Function Block Definitions

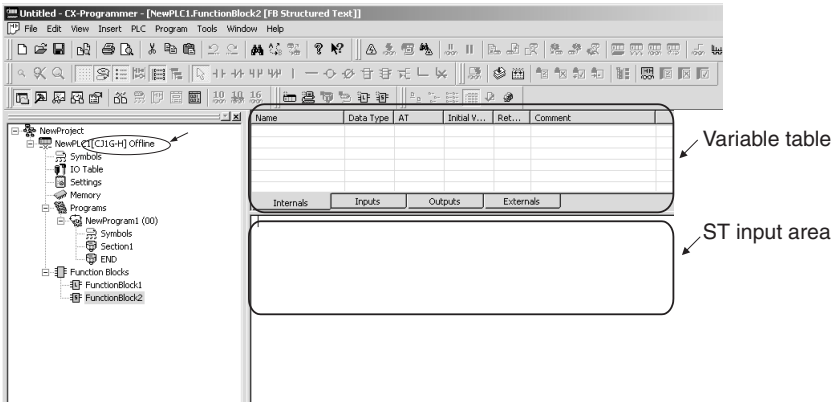
Creating Function Block Definitions

One of the following windows will be displayed when the newly created Function Block 1 icon is double-clicked (or if it is right-clicked and ***Open*** is selected from the popup menu). A variable table for the variables used in the function block is displayed on top and an input area for the ladder program or structured text is displayed on the bottom.

Ladder Program



Structured Text



As shown, a function block definition consists of a variable table that serves as an interface and a ladder program or structured text that serves as an algorithm.

Variable Table as an Interface

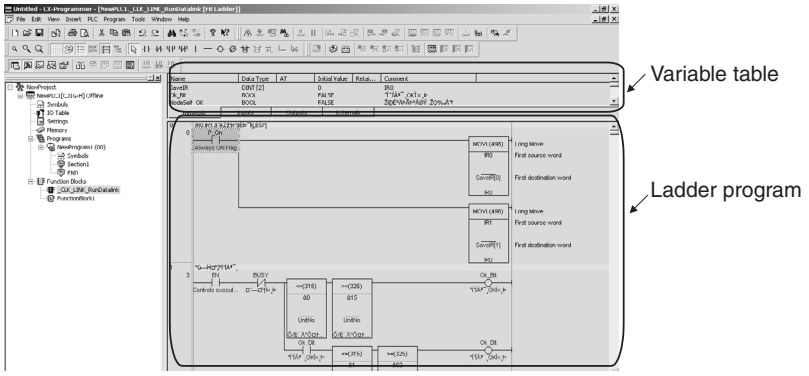
At this point, the variable table is empty because there are no variables allocated for I/O memory addresses in the PLC.

Ladder Program or Structure Text as an Algorithm

- With some exceptions, the ladder program for the function block can contain any of the instructions used in the normal program. Refer to *2-3 Restrictions on Function Blocks* for restrictions on the instructions that can be used.
- Structured text can be input according to the ST language defined in IEC61131-3.

Using OMRON FB Library Files

Double-click the inserted function block library (or right-click and select ***Open*** from the pop-up menu) to display the variable table that has finished being created at the top right window, and the ladder program that has finished being created in the bottom right window. Both windows are displayed in gray and cannot be edited.



Note Function block definitions are not displayed in the default settings for OMRON FB Library files (.cxf). To display definitions, select the ***Display the inside of FB*** option in the function block properties. (Select the OMRON FB Library file in the project workspace, right-click, select ***Properties***, and select the ***Display the inside of FB*** option in the General Tab.)

3-2-3 Defining Function Blocks Created by User

A function block is defined by registering variables and creating an algorithm. There are two ways to do this.

- Register the variables first and then input the ladder program or structure text.
- Register variables as they are required while inputting input the ladder program or structure text.

Registering Variables First

Registering Variables in the Variable Table

The variables are divided by type into four sheets in the variable table: Internals, Inputs, Outputs, and Externals.

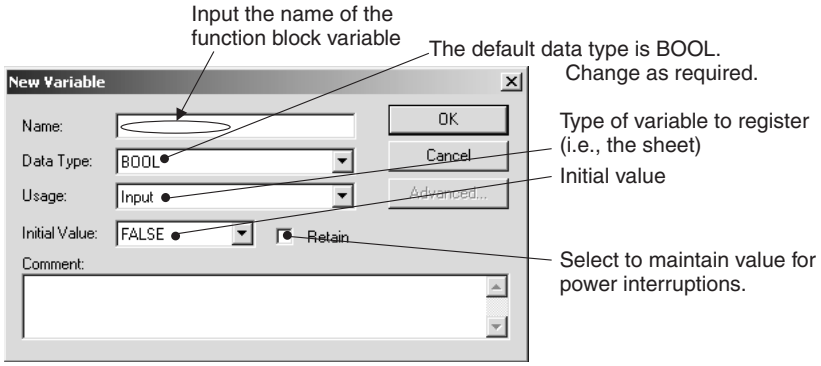
These sheets must be switched while registering or displaying the variables.

1,2,3...

1. Make the sheet for the type of variable to be registered active in the variable table. (See note.) Place the cursor in the sheet, right-click, and perform either of the following operations:
   - To add a variable to the last line, select ***Insert Variable*** from the popup menu.
   - To add the variable to the line above or below a line within the list, select ***Insert Variable - Above*** or ***Below*** from the popup menu.

   Note The sheet where a variable is registered can also be switched when inserting a variable by setting the usage (N: Internals, I: Inputs, O: Outputs, E: Externals).

   The New Variable Dialog Box shown below will be displayed.
   - **Name:** Input the name of the variable.
   - **Data Type:** Select the data type.
   - **Usage:** Select the variable type.
   - **Initial Value:** Select the initial value of the variable at the start of operation.
   - **Retain:** Select if the value of the variable is to be maintained when the power is turned ON or when the operating mode is changed from PROGRAM or MONITOR mode to RUN mode. The value will be cleared at these times if *Retain* is not selected.

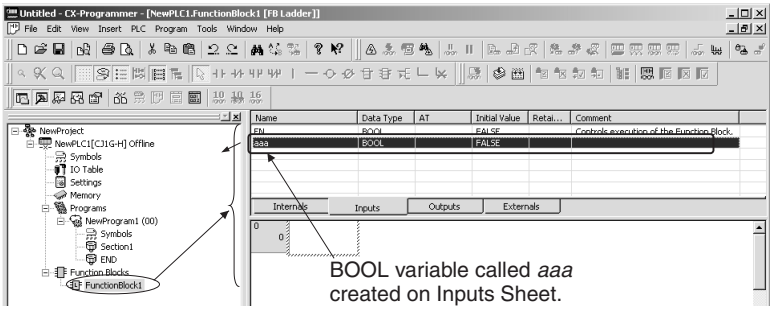


Note (a) For user-defined external variables, the global symbol table can be browsed by registering the same variable name in the global symbol table.

(b) External variables defined by the system are registered in the external variable table in advance.

2. For example, input “aaa” as the variable name and click the **OK** Button.

   As shown below, a BOOL variable called *aaa* will be created on the Inputs Sheet of the Variable Table.



Note (1) After a variable is added, it can be selected to display in reverse video, then moved to another line by dragging and dropping. To select a variable for dragging and dropping, select the variable in any of the columns except the *Name* field.

(2) After inputting a variable, to move to the registered sheet, double-click, and switch the setting in the *Usage* field (N: Internals, I: Inputs, O: Outputs, E: Externals). The variable can also be copied or moved between the sheets for internal, external, input, and output variables. Select the variable, right-click, and select ***Copy*** or ***Cut*** from the pop-up menu, and then select ***Paste***.

(3) Variable names must also be input for variables specified with AT (allocating actual address) settings.

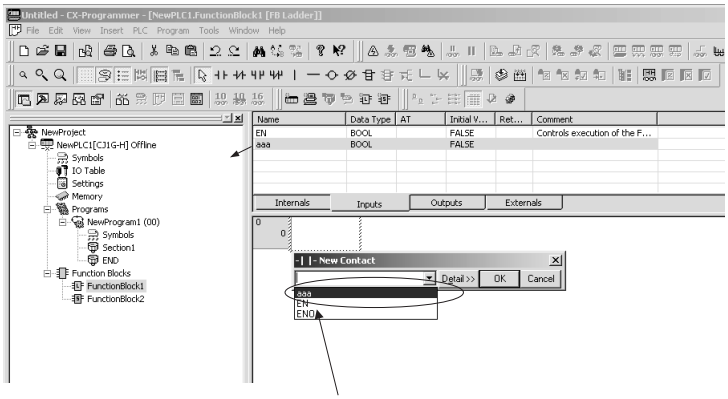
(4) The following text is used to indicate I/O memory addresses in the PLC and thus cannot be input as variable names in the function block variable table.

- A, W, H, HR, D, DM, E, EM, T, TIM, C, or CNT followed by a numeric value

Creating the Algorithm

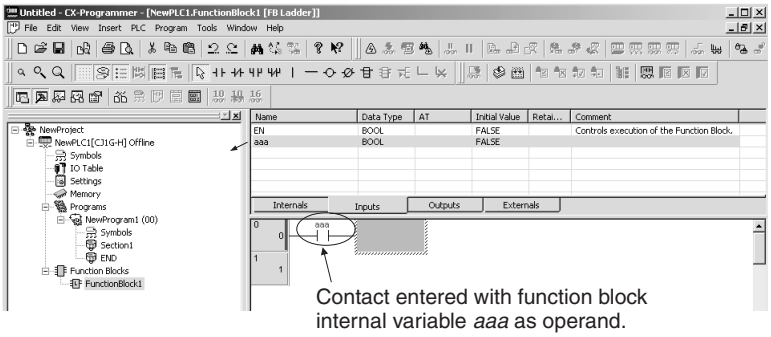
Using a Ladder Program

1,2,3... 1. Press the **C** Key and select *aaa* registered earlier from the pull-down menu in the New Contact Dialog Box.



Press the C Key and select *aaa* registered earlier from the pull-down menu in the New Contact Dialog Box.

2. Click the **OK** Button. A contact will be entered with the function block internal variable *aaa* as the operand (variable type: internal).



Contact entered with function block internal variable *aaa* as operand.

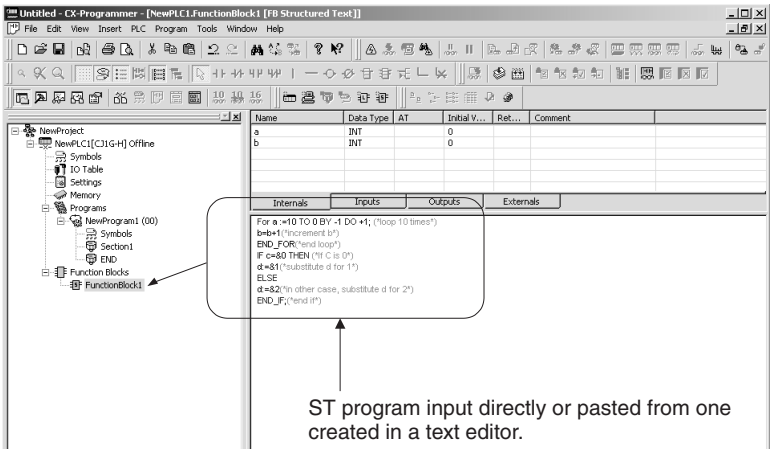
The rest of the ladder program is input in exactly the same way as for standard programs with CX-Programmer.

Note Addresses cannot be directly input for instruction operands within function blocks. Only Index Registers (IR) and Data Registers (DR) can be input directly as follows (not as variables): Addresses DR0 to DR5, direct specifications IR0 to IR15, and indirect specifications ,IR0 to ,IR15.

Using Structured Text

An ST language program (see note) can either be input directly into the ST input area or a program input into a general-purpose text editor can be copied and then pasted into the ST input area using the *Paste* Command on the Edit Menu.

Note The ST language conforms to IEC61131-3. Refer to *Appendix B Structured Text (ST Language) Specifications* for details.



Note

(1) Tabs or spaces can be input to create indents. They will not affect the algorithm.

(2) The display size can be changed by holding down the **Ctrl** Key and turning the scrolling wheel on a wheel mouse.

(3) When an ST language program is input or pasted into the ST input area, syntax keywords reserved words will be automatically displayed in blue, comments in green, errors in red, and everything else in black.

(4) To change the font size or colors, select ***Options*** from the Tools Menu and then click the **ST Font** Button on the Appearance Tab Page. The font names, font size (default is 8 point) and color can be changed.

(5) For details on structured text specifications, refer to *Appendix B Structured Text (ST Language) Specifications.*

Registering Variables as Required

The ladder program or structured text program can be input first and variable registered as they are required.

Using a Ladder Program

When using a ladder diagram, a dialog box will be displayed to register the variable whenever a variable name that has not been registered is input. The variable is registered at that time.

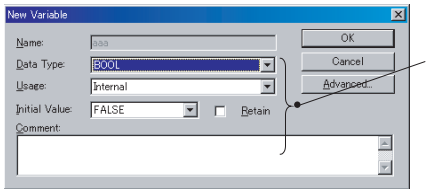
Use the following procedure.

1,2,3...

1. Press the **C** Key and input a variable name that has not been registered, such as *aaa,* in the New Contact Dialog Box.

   Note Addresses cannot be directly input for instruction operands within function blocks. Only Index Registers (IR) and Data Registers (DR) can be input directly as follows (not as variables): Addresses DR0 to DR5, direct specifications IR0 to IR15, and indirect specifications ,IR0 to ,IR15.

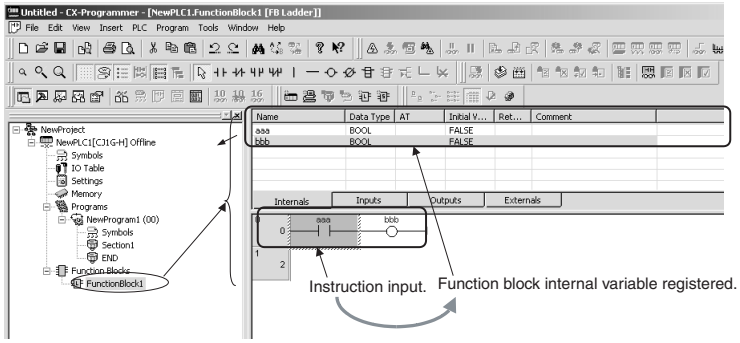
2. Click the **OK** Button. The New Variable Dialog Box will be displayed. With special instructions, a New Variable Dialog Box will be display for each operand in the instruction.



The properties for all input variables will initially be displayed as follows:

- Usage: Internal
- Data Type: BOOL for contacts and WORD for channel (word)
- Initial Value: The default for the data type.
- Retain: Not selected.

3. Make any required changes and click the **OK** Button.
4. As shown below, the variable that was registered will be displayed in the variable table above the program.



5. If the type or properties of a variable that was input are not correct, double-click the variable in the variable table and make the required corrections.

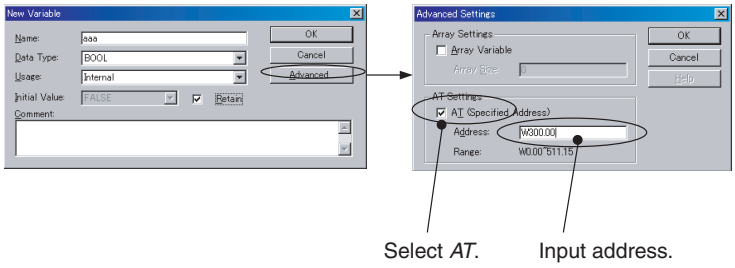
■ Reference Information

AT Settings (Specified Address)

AT settings can be made in the variable properties to specify allocation addresses for Basic I/O Units, Special I/O Units, or CPU Bus Units, or Auxiliary Area addresses not registered using the CX-Programmer. A variable name is required to achieve this. Use the following procedure to specify an address.

1,2,3...

1. After inputting the variable name in the New Variable Dialog Box, click the **Advanced** Button. The Advanced Settings Dialog Box will be displayed.
2. Select *AT (Specified Address)* under *AT Settings* and input the desired address.



The variable name is used to enter variables into the algorithm in the function block definition even when they have an address specified for the AT settings (the same as for variables without a specified address).
For example, if a variable named *Restart* has an address of A50100 specified for the AT settings, *Restart* is specified for the instruction operand.

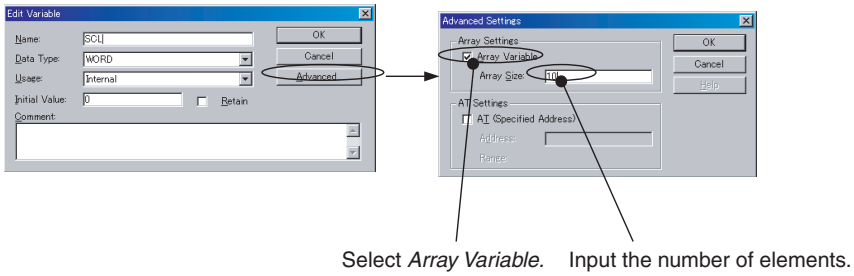
Array Settings

An array can be specified to use the same data properties for more than one variable and manage the variables as a group.

Use the following procedure to set an array.

1,2,3...

1. After inputting the variable name in the New Variable Dialog Box, click the **Advanced** Button. The Advanced Settings Dialog Box will be displayed.
2. Select *Array Variable* in the *Array Settings* and input the maximum number of elements in the array.



When the name of an array variable is entered in the algorithm in the function block definition, square brackets surrounding the index will appear after the array name.

For example, if you create a variable named PV with a maximum of 3 elements, PV[0], PV[1], and PV[2] could be specified as instruction operands.

There are three ways to specify indices.

- Directly with numbers, e.g., PV[1] in the above example (for ladder programming or ST language programming)
- With a variable, e.g., PV[a] in the above example, where “a” is the name of a variable with a data type of INT (for ladder programming or ST language programming)
- With an equation, e.g., PV[a+b] or PV[a+1} in the above example, where “a” and “b” are the names of variables with a data type of INT (for ST language programming only)

Using Structured Text

When using structured text, a dialog box will not be displayed to register the variable whenever a variable name that has not been registered is input. Be sure to always register variables used in standard text programming in the variable table, either as you need them or after completing the program. (Place the cursor in the tab page on which to register the variable, right-click, and select ***Insert Variable*** from the popup menu.

Note For details on structured text specifications, refer to *Appendix B Structured Text (ST Language) Specifications.*

Copying User Program Circuits and Pasting in Ladder Programming of Function Block Definitions

A single circuit or multiple circuits in the user program can be copied and pasted in the ladder programming of function block definitions. This operation, however, is subject to the following restrictions.

Source Instruction Operand: Address Only

Addresses are not registered in the function block definition variable tables. After pasting, the addresses will be displayed in the operand in red. Double-click on the instruction and input the variable name into the operand.

Note Index Registers (IR) and Data Registers (DR), however, do not require modification after pasting and function in the operand as is.

Source Instruction Operand: Address and I/O Comment

***Automatically generate symbol name* Option Selected in Symbols Tab under Options in Tools Menu**

The user program symbol names (in the global symbol table only) will be generated automatically as AutoGen_ + *Address* (if the option is deselected, the symbol names will be removed).

Example 1: For address 100.01, the symbol name will be displayed as AutoGen_100_01.

Example 2: For address D0, the symbol name will be displayed as AutoGen_D0.

If circuits in the user program are copied and pasted into the function block definition program as is, the symbols will be registered automatically in the function block definition symbol table (at the same time as copying the circuits) as the symbol name AutoGen_*Address* and I/O comments as *Comment*. This function enables programmed circuits to be easily reused in function blocks as addresses and I/O comments.

Note The prefix AutoGen_ is not added to Index Registers (IR) and Global Data Registers (DR), and they cannot be registered in the original global symbol table.

***Automatically generate symbol name* Option Not Selected in Symbols Tab under Options in Tools Menu**

Addresses and I/O comments are not registered in the function block definition variable tables. Addresses are displayed in the operand in red. I/O comments will be lost. Double-click on the instruction and input the symbol name into the operand.

Index Registers (IR) and Data Registers (DR), however, do not require modification after pasting and function in the operand as is.

Source Instruction Operand: Symbol

The user program symbol is automatically registered in the internal variables of the function block definition variable table. This operation, however, is subject to the following restrictions.

Addresses

Symbol addresses are not registered. Use AT settings to specify the same address.

Symbol Data Types

The symbol data types are converted when pasted from the user program into the function block definition, as shown in the following table.

Symbol data type in user program	→	Variable data type after pasting in function block program
CHANNEL	→	WORD
NUMBER	→	The variable will not be registered, and the value (number) will be pasted directly into the operand as a constant.
UINT BCD	→	WORD
UDINT BCD	→	DWORD
ULINT BCD	→	LWORD

Symbol data types CHANNEL, NUMBER, UINT BCD, UDINT BCD, or ULINT BCD, however, cannot be copied from the symbol table (not the program) and then pasted into the variable table in the function block definition.

Note Symbols with automatically generated symbol names (AutoGen_ + *Address*) cannot be copied from a global symbol table and pasted into the function block definition symbol table.

3-2-4 Creating Instances from Function Block Definitions

If a function block definition is registered in the global symbol table, either of the following methods can be used to create instances.

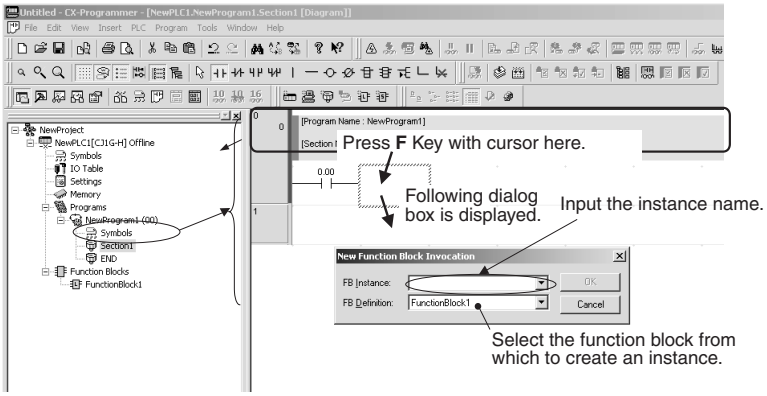
Method 1:Select the function block definition, insert it into the program, and input a new instance name. The instance will automatically be registered in the global symbol table.

Method 2: Set the data type in the global symbol table to "FUNCTION BLOCK," specify the function block definition to use, and input the instance name to register it.

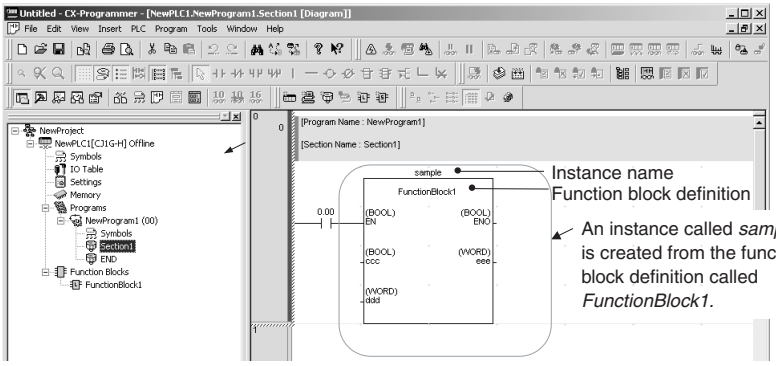
■ Method 1: Using the F Key in the Ladder Section Window and Inputting the Instance Name

1,2,3...

1. In the Ladder Section Window, place the cursor in the program where the instance is to be inserted and press the **F** Key. (Alternately, select ***Function Block Invocation*** from the Insert Menu.) The New Function Block Invocation Dialog Box will be displayed.
2. Input the instance name, select the function block from which to create an instance, and click the **OK** Button.



3. As an example, set the instance name in the *FB Instance* Field to ***sample***, set the function block in the *FB Definition* Field to ***FunctionBlock1***, and click the **OK** Button. As shown below, a copy of the function block definition called *FunctionBlock1* will be created with an instance name of *sample.*



The instance will be automatically registered in the global symbol table with an instance name of *sample* and a data type of *FUNCTION BLOCK.*

■ **Method 2: Registering the Instance Name in the Global Symbol Table in Advance and Then Selecting the Instance Name**

If the instance name is registered in the global symbol table in advance, the instance name can be selected from the global symbol table to create other instances.

1,2,3... 1. Select a data type of *Function block* in the global symbol table, input the instance name, and registered the instance.
2. Press the **F** Key in the Ladder Section Window. The Function Block Invocation Dialog Box will be displayed.
3. Select the instance name that was previously registered from the pulldown menu on the *FB Instance* Field. The instance will be created.

Restrictions

Observe the following restrictions when creating instances. Refer to *2-3 Restrictions on Function Blocks* for details.

- No more than one function block can be created in each program circuit.
- The rung cannot be branched to the left of an instance.
- Instances cannot be connected directly to the left bus bar, i.e., an EN must always be inserted.

Note If changes are made in the I/O variables in a variable table for a function block definition, the bus bar to the left of all instances that have been created from that function block definition will be displayed in red to indicate an error. When this happens, select the function block, right-click, and select ***Update Invocation***. The instance will be updated for any changes that have been made in the function block definition and the red bus bar display indicating an error will be cleared.

3-2-5 Setting Function Block Parameters Using the P Key

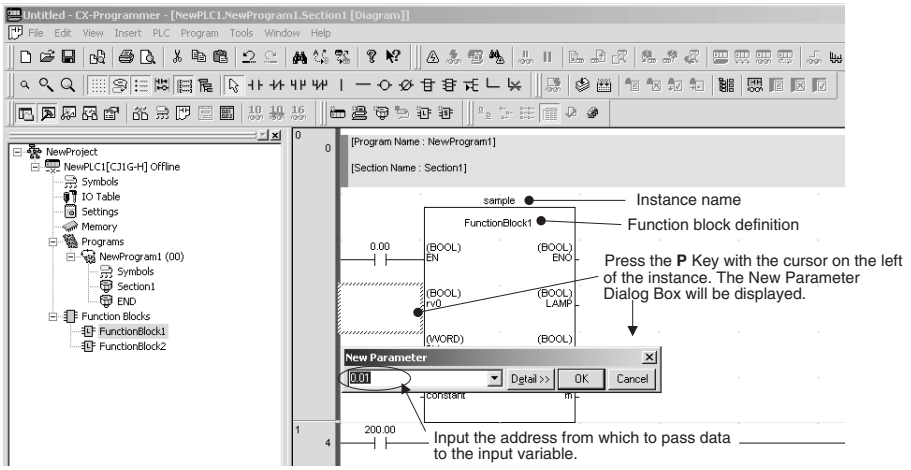
After an instance of a function block has been created, input parameters must be set for input variables and output parameters must be set for output variables to enable external I/O.

- Values, addresses, and program symbols (global symbols and local symbols) can be set in input parameters. (See note a.)
- Addresses and program symbols (global symbols and local symbols) can be set in output parameters. (See note b.)

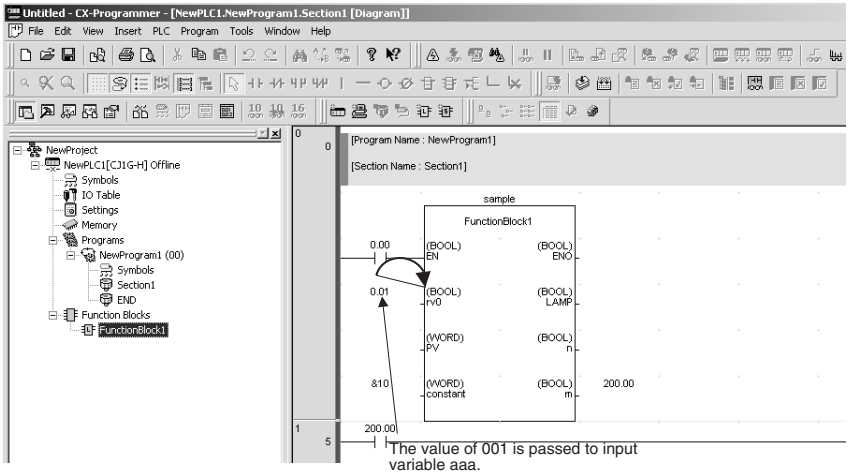
Note (a) The function block's input variable data size and the program's symbol data size must match.

(b) The function block's output variable data size and the program's symbol data size must match.

1,2,3... 1. Inputs are located on the left of the instance and outputs on the right. Place the cursor where the parameter is to be set and press the **P** Key. (Alternately, select ***Function Block Parameter*** from the Insert Menu.) The New Parameter Dialog Box will be displayed as shown below.



2. Set the source address from which to pass the address data to the input variable. Also set the destination address to which the address data will be passed from the output variable.



Note Set the data in all the input parameters. If even a single input parameter remains blank, the left bus bar for the instance will be displayed in red to indicate an error. If this happens, the program cannot be transferred to the CPU Unit.

Inputting Values in Parameters

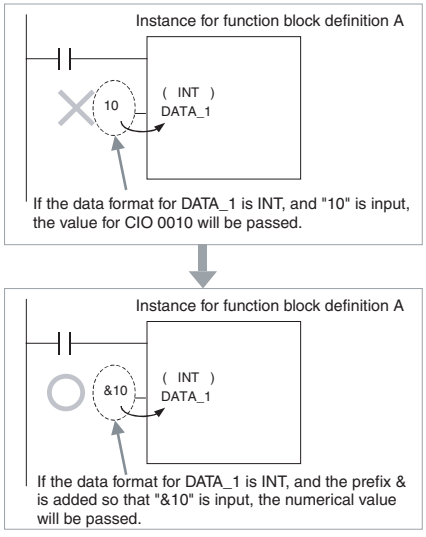
The following table lists the methods for inputting values in parameters.

Input variable data type	Content	Size	Input method	Setting range
BOOL	Bit data	1 bit	P_Off, P_On	0 (FALSE), 1 (TRUE)
INT	Integer	16 bits	Positive value: & or + followed by integer Negative value: − followed by integer	−32768 to +32767
DINT	Double integer	32 bits		−2147483648 to +2147483647
LINT	Long (4-word) integer	64 bits		−9223372036854775808 to +9223372036854775807
UINT	Unsigned integer	16 bits	Positive value: & or + followed by integer	&0 to 65535
UDINT	Unsigned double integer	32 bits		&0 to 4294967295
ULINT	Unsigned long (4-word) integer	64 bits		&0 to 18446744073709551615

Input variable data type	Content	Size	Input method	Setting range
REAL	Real number	32 bits	Positive value: & or + followed by real number (with decimal point) Negative value: − followed by real number (with decimal point)	-3.402823×10^{38} to $-1.175494 \times 10^{-38}$, 0, $+1.175494 \times 10^{-38}$ to $+3.402823 \times 10^{38}$
LREAL	Long real number	64 bits		$-1.79769313486232 \times 10^{308}$ to $-2.22507385850720 \times 10^{-308}$, 0, $+2.22507385850720 \times 10^{-308}$ to $+1.79769313486232 \times 10^{308}$
WORD	16-bit data	16 bits	# followed by hexadecimal number (4 digits max.) & or + followed by decimal number	#0000 to FFFF or &0 to 65535
DWORD	32-bit data	32 bits	# followed by hexadecimal number (8 digits max.) & or + followed by decimal number	#00000000 to FFFFFFFF or &0 to 4294967295
LWORD	64-bit data	64 bits	# followed by hexadecimal number (16 digits max.) & or + followed by decimal number	#0000000000000000 to FFFFFFFFFFFFFFFF or &0 to 18446744073709551615

Note If a non-boolean data type is used for the input variable and only a numerical value (e.g., 20) is input, the value for the CIO Area address (e.g, CIO 0020) will be passed, and not the numerical value. To set a numerical value, always insert an &, #, + or − prefix before inputting the numerical value.

Example Programs:



If the input variable data type is boolean and a numerical value only (e.g., 0 or 1) is input in the parameter, the value for CIO 000000 (0.00) or CIO 000001 (0.01) will be passed. Always input P_Off for 0 (OFF) and P_On for 1 (ON).

3-2-6 Setting the FB Instance Areas

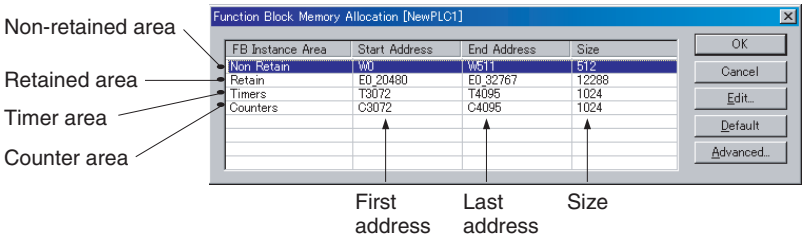
The areas where addresses for variables used in function blocks are allocated can be set. These areas are called the function block instance areas.

1,2,3...

1. Select the instance in the Ladder Section Window or in the global symbol table, and then select ***Function Block Memory - Function Block Memory Allocation*** from the PLC Menu.

   The Function Block Memory Allocation Dialog shown below will appear.

2. Set the FB instance areas.



The non-retained and retained areas are set in words. The timer and counter areas are set by time and counter numbers.

The default values are as follows:

FB Instance Area	Default value			Applicable memory areas
	Start Address	End Address	Size	
Non Retain (See notes 1 and 3.)	H512 (See note 2.)	H1407 (See note 2.)	896	CIO, WR, HR, DM, EM
Retain (See note 1.)	H1408 (See note 2.)	H1535 (See note 2.)	128	HR, DM, EM
Timers	T3072	T4095	1024	TIM
Counters	C3072	C4095	1024	CNT

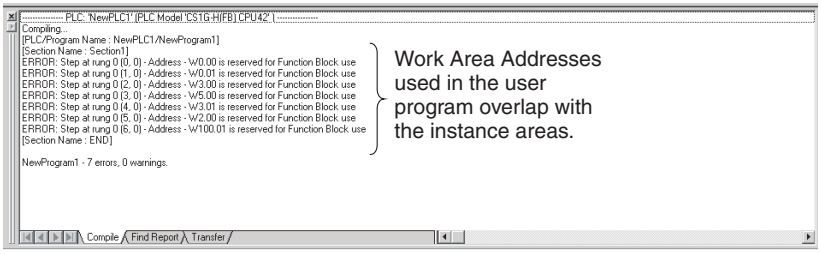
Note

(1) Bit data can be accessed even if the DM or EM Area is specified for the non-retained area or retained area.

(2) The Function Block Holding Area words are allocated in H512 to H1535. These words cannot be specified in instruction operands in the user program. These words can also not be specified in the internal variable's AT settings.

(3) Words H512 to H1535 are contained in the Holding Area, but the addresses set as non-retained will be cleared when the power is turned OFF and ON again or when operation is started.

(4) To prevent overlapping of instance area addresses and addresses used in the program, set H512 to H1535 (Function Block Holding Area words) for the non-retained area and retained area. If another area is set, the addresses may overlap with addresses that are used in the user program. If the addresses in the function block instance areas overlap with any of the addresses used in the user program, an error will occur when compiling. This error will also occur when a program is downloaded, edited online, or checked by the user.



If addresses are duplicated and an error occurs, either change the function block instance areas or the addresses used in the user program.

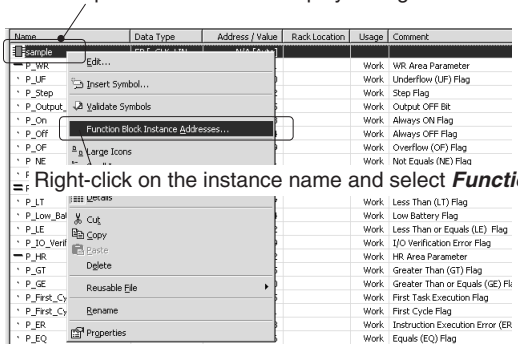
3-2-7 Checking Internal Address Allocations for Variables

The following procedure can be used to check the I/O memory addresses internally allocated to variables.

1,2,3... 1. Select ***View - Symbols - Global.***

2. Select the instance in the global symbol table, right-click, and select ***Function Block Memory Address*** from the popup menu. (Alternately, select ***Function Block Memory - Function Block Memory Address*** from the PLC Menu.)

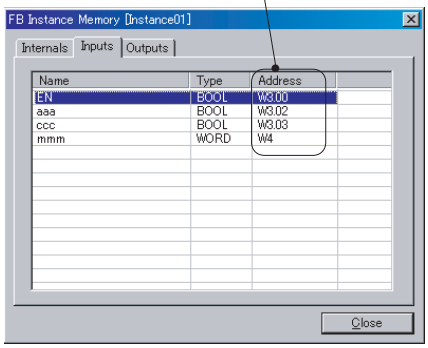
Example: Instance name displayed in global variable table (automatically registered)



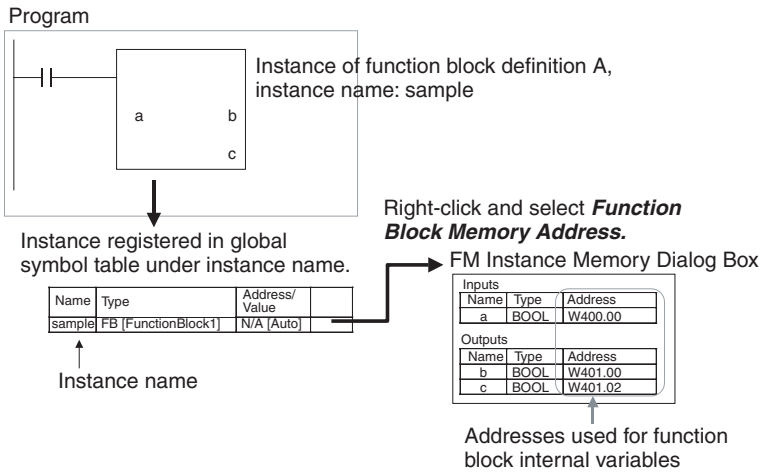
Right-click on the instance name and select ***Function Block Instance Address.***

3. The FB Interface Memory Dialog Box will be displayed. Check the I/O memory addresses internally allocated to variables here.

Example: Addresses used internally for the input variables.



Method Used for Checking Addresses Internally Allocated to Variables

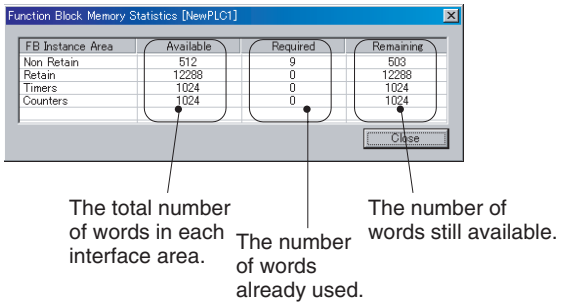


Checking the Status of Addresses Internally Allocated to Variables

The following procedure can be used to check the number of addresses allocated to variables and the number still available for allocation in the function block instance areas.

1,2,3...

1. Select the instance in the Ladder Section Window, right-click, and select ***Function Block Memory - Function Block Memory Statistics*** from the PLC Menu.
2. The Function Block Memory Statistics Dialog Box will be displayed as shown below. Check address usage here.



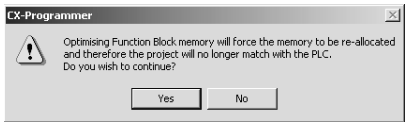
Optimizing Function Memory

When a variable is added or deleted, addresses are automatically re-allocated in the variables' instance area. Consecutive addresses are required for each instance, so all of the variables will be allocated to a different block of addresses if the original block of addresses cannot accommodate the change in variables. This will result in an unused block of addresses. The following procedure can be used to eliminate the unused areas in memory so that memory is used more efficiently.

1,2,3...

1. Select the instance in the Ladder Section Window, right-click, and select ***Function Block Memory - Optimize Function Memory*** from the PLC Menu.

   The following dialog box will be displayed.



2. Click the **OK** Button. Allocations to the function block instance areas will be optimized.

3-2-8 Copying and Editing Function Block Definitions

Use the following operation to copy and edit the function block definition that has been created.

1. Select the function block to copy, right-click, and select ***Copy*** from the pop-up menu.
2. Position the cursor over the function block item under the PLC in the project directory, right-click and select ***Paste*** from the pop-up menu.
3. The function block definition will be copied ("copy" is indicated before the name of the function block definition at the copy source).
4. To change the function block name, left-click or right-click and select ***Rename*** from the pop-up menu.
5. Double-click the function block definition to edit it.

3-2-9 Checking the Source Function Block Definition from an Instance

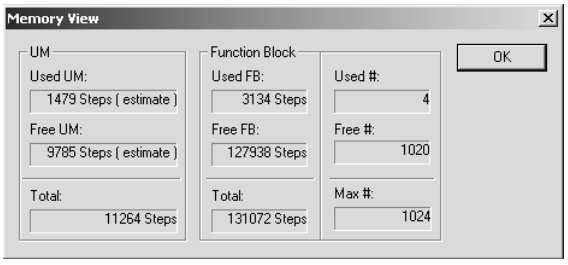
Use the following procedure to check the function block definition from which an instance was created.

1,2,3... Right-click the instance and select ***Go To - Function Block Definition*** from the popup menu. The function block definition will be displayed.

3-2-10 Checking the Size of the Function Block Definition

CX-Programmer can be used to check the size of the function block definition being created using a similar method to checking the program capacity. The procedure is as follows:

1. Select ***Memory View*** from the View Menu.
2. The function block definition size and number of function block definitions will be displayed in the Memory View Dialog Box as shown below.



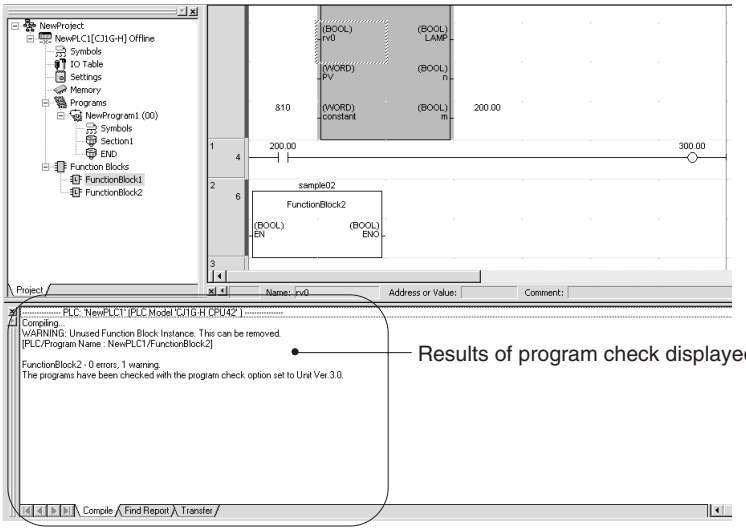
- The *Used FB*, *Free FB*, and *Total* fields under *Function Block* refer to the size of the function block definitions. The values are displayed in step units. 1 step = 4 bytes, so the function block source memory capacity (K bytes) in the CPU Unit's internal flash memory is the value multiplied by 1,024 and divided by 4.
- The *Used #*, *Free #*, and *Max #* fields under *Function Block* refer to the number of function block definitions.

3-2-11 Compiling Function Block Definitions (Checking Program)

A function block definition can be compiled to perform a program check on it. Use the following procedure.

1,2,3... Select the function block definition, right-click, and select ***Compile*** from the popup menu. (Alternately, press the **Ctrl + F7** Keys.)

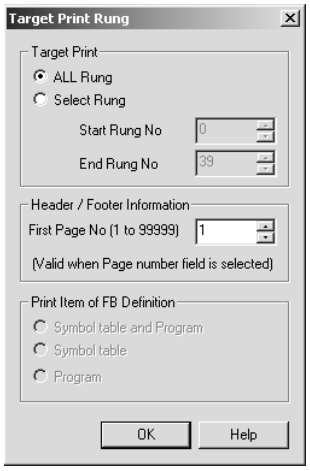
The function block will be compiled and the results of the program check will be automatically displayed on the Compile Table Page of the Output Window.



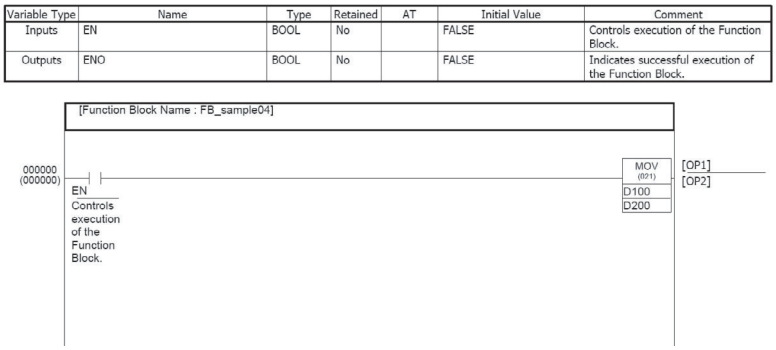
3-2-12 Printing Function Block Definition

Use the following procedure to print function block definitions.

1,2,3... 1. Double-click the function block definition to be printed, and with the variable table and algorithm displayed, select ***Print*** from the File Menu. The following Target Print Rung Dialog Box will be displayed.



2. Select the ***All Rung*** or ***Select Rung*** option. When the Select Rung option is selected, specify the start rung and end rung numbers. When a page number has been specified in the header and footer fields in *File - Page Setup*, the first page number can be specified.
3. Select either of the following options for the function block printing range.
   - Symbol table and program (default)
   - Symbol table
   - Program
4. Click the **OK** Button, and display the Print Dialog Box. After setting the printer, number of items to print and the paper setting, click the OK button.
5. The following variable table followed by the algorithm (e.g, ladder programming language) will be printed.



Note For details on print settings, refer to the section on printing in the *CX-Programmer Ver. 5.0 Operation Manual* (W437).

3-2-13 Saving and Reusing Function Block Definition Files

The function block definition that has been created can be saved independently as a function block library file (*.cxf) to enable reusing it in other projects.

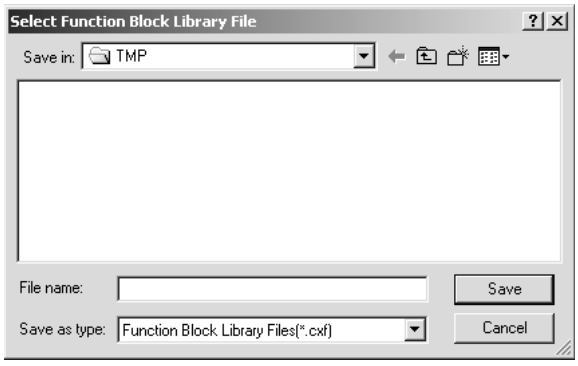
Note Before saving to file, or reusing in another project, compile the function block definition and perform a program check.

Saving a Function Block Library File

Use the following procedure to save a function block definition to a function block library file.

1,2,3...

1. Select the function block definition, right-click, and select ***Save Function Block to File*** from the popup menu. (Alternately, select ***Function Block - Save Function Block to File*** from the File Menu.)
2. The following dialog box will be displayed. Input the file name. *Function Block Library Files (*.cxf)* should be selected as the file type.

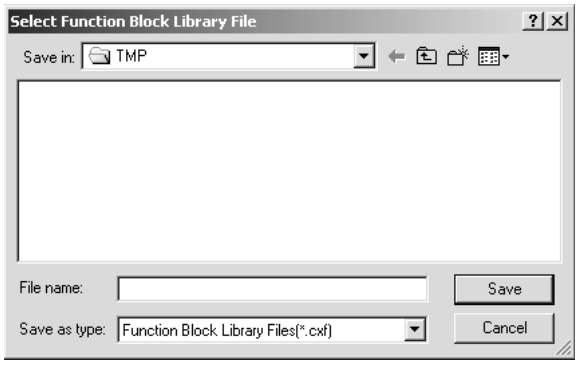


Reading Function Block Library Files into Other Projects

Use the following procedure to read a function block library file (*.cxf) into a project.

1,2,3...

1. Select the function block definition item under the PLC directory in the Project Workspace, right-click, and select ***Insert Function Block - From File*** from the popup menu (or select ***File - Function Block - Load Function Block from File***).
2. The following dialog box will be displayed. Select a function block library file (*.cxf) and click the **Open** Button.



3. A function block called FunctionBlock1 will be automatically inserted after the Function Blocks icon. This icon contains the definition of the function block.
4. Double-click the **FunctionBlock1** Icon. The variable table and algorithm will be display.

3-2-14 Downloading/Uploading Programs to the Actual CPU Unit

After a program containing function blocks has been created, it can be downloaded from the CX-Programmer to an actual CPU Unit that it is connected to online. Programs can also be uploaded from the actual CPU Unit. It is also possible to check if the programs on the CX-Programmer Ver. 5.0 (personal computer) and in the actual CPU Unit are the same. When the program contains function blocks, however, downloading in task units is not possible (uploading is possible).

3-2-15 Monitoring and Debugging Function Blocks

The following procedures can be used to monitor programs containing function blocks.

Monitoring Programs in Function Block Definitions

Use the following procedure to check the program in the function block definition for an instance during monitoring.

1,2,3... Right-click the instance and select ***Go To - Function Block Definition*** from the popup menu. The function block definition will be displayed.

Monitoring Instance Variables in the Watch Window

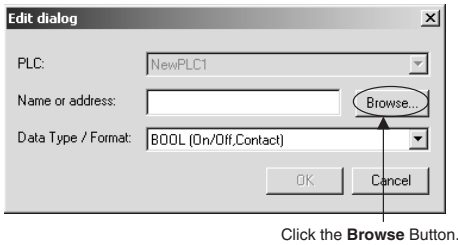
Use the following procedure to monitor instance variables.

1,2,3... 1. Select ***View - Window - Watch.***

A Watch Window will be displayed.

2. Double-click the watch window.

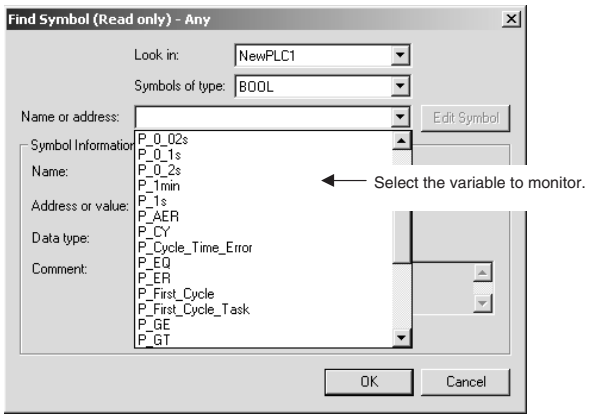
The Edit Dialog Box will be displayed as shown below.



Click the **Browse** Button.

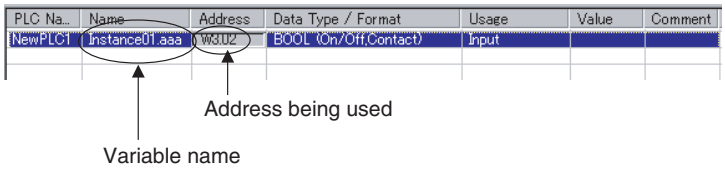
Note The instance variable is displayed as the instance name and variable name.

3. Click the **Browse** Button, select the variable to be monitored, and click the **OK** Button.



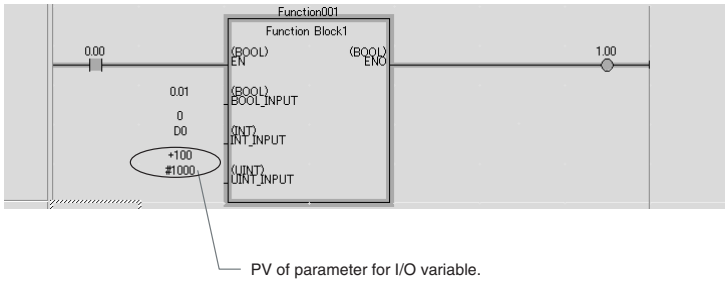
Note Instance variables are displayed as *instance_name*, *variable_name*.

4. Click the **OK** Button. Variable values will be display in the Watch Window as shown below.



Monitoring Instance I/O Variables

The present values of parameters for I/O variables are displayed below the parameters.



Editing Function Block Definition Programs Online

Programs using function blocks can be edited online. Changes can also be made around instances.

- Instance parameters can be changed, instances can be deleted, and instructions other than those in instances can be changed.
- Instances cannot be added, instance names cannot be changed, and algorithms and variable tables in function block definitions cannot be changed.

Appendix A
Data Types

Basic Data Types

Data type	Content	Size	Range of values
BOOL	Bit data	1	0 (FALSE), 1 (TRUE)
INT	Integer	16	–32,768 to +32,767
DINT	Double integer	32	–2,147,483,648 to +2,147,483,647
LINT	Long (8-byte) integer	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
UINT	Unsigned integer	16	&0 to 65,535
UDINT	Unsigned double integer	32	&0 to 4,294,967,295
ULINT	Unsigned long (8-byte) integer	64	&0 to 18,446,744,073,709,551,615
REAL	Real number	32	-3.402823×10^{38} to $-1.175494 \times 10^{-38}$, 0, $+1.175494 \times 10^{-38}$ to $+3.402823 \times 10^{38}$
LREAL	Long real number	64	$-1.79769313486232 \times 10^{308}$ to $-2.22507385850720 \times 10^{-308}$, 0, $2.22507385850720 \times 10^{-308}$ to $1.79769313486232 \times 10^{308}$
WORD	16-bit data	16	#0000 to FFFF or &0 to 65,535
DWORD	32-bit data	32	#00000000 to FFFFFFFF or &0 to 4,294,967,295
LWORD	64-bit data	64	#0000000000000000 to FFFFFFFFFFFFFFFF or &0 to 18,446,744,073,709,551,615
TIMER (See note.)	Timer (See note.)	Flag: 1 bit PV: 16 bits	Timer number: 0 to 4095 Completion Flag: 0 or 1 Timer PV: 0 to 9999 (BCD), 0 to 65535 (binary)
COUNTER (See note.)	Counter (See note.)	Flag: 1 bit PV: 16 bits	Counter number: 0 to 4095 Completion Flag: 0 or 1 Counter PV: 0 to 9999 (BCD), 0 to 65535 (binary)

Note The TIMER and COUNTER data types cannot be used in structured text function blocks.

Derivative Data Types

Array	1-dimensional array; 32,000 elements max.

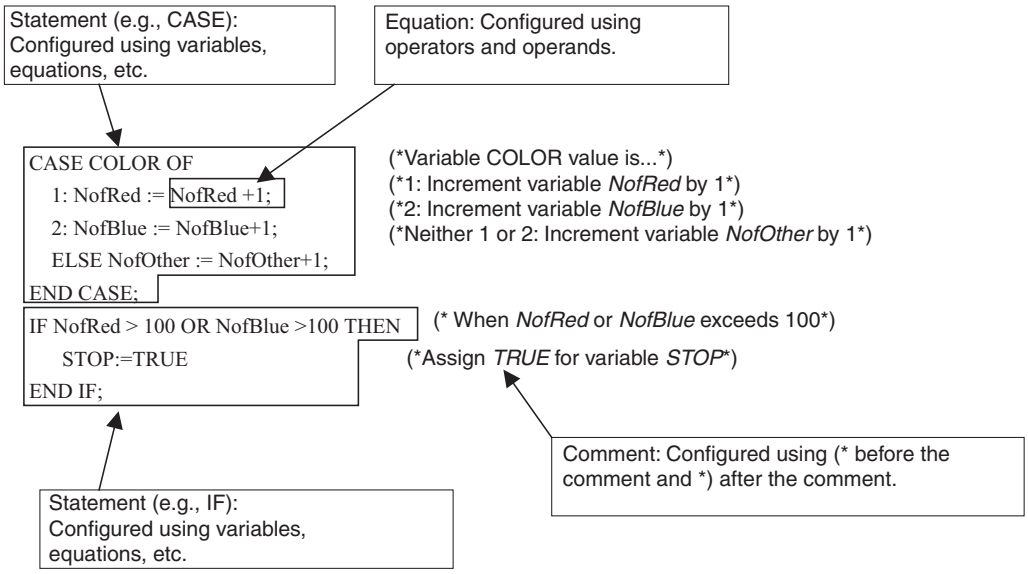
Appendix B
Structured Text (ST Language) Specifications

Structured Text

Structured text (also referred to as ST language) is a high-level programming language similar to PASCAL that uses language structures such as selection and iteration statements. Programs written using structured text are configured using statements. Statements are configured from variables and equations.

- Equations are sequences containing operators and operands (variables or constants). Operators include arithmetic operators, comparison operators, and logical operators.
- Statements are either assignment or control statements. Assignment statements store calculation results from the equation in the variable. Control statements include selection statements and iteration statements.

Example of Structured Text



Restrictions

Statement Delimiters

- Statements (assignment and control statements) must always end in a semicolon (;). The statement cannot be completed by simply using a carriage return.
- Do not use a semicolon (;) as a delimiter within a statement such as following reserved words, values, or equations. Inserting a semicolon within a statement, except at the end of a statement, will result in a syntax error.

Comments

- Comments are enclosed in parentheses and asterisks, i.e., (**comment**). Any characters except parentheses and asterisks can be used within a comment. Nesting within comments is not supported.

Notation	Example
(**comment**)	(*this is the comment*)

Note: Nesting in comments is not possible, i.e., (*(*this type of nesting is not supported*)*)

Spaces, Carriage Returns, Tabs

- Any number of spaces, carriage returns, and tabs, or combinations of these can be used anywhere within statements. Therefore, use spaces, carriage returns, and tabs between reserved words and equations to make them easier to read.
- Spaces, carriage returns, and tabs cannot be used between the following tokens (the smallest meaningful unit for compiling), in which case they are referred to as token separators.
  Tokens: Reserved words, variable names, special characters, constants (numerical values)

  Reserved words (upper or lower case): AND, CASE, DO, ELSE, FOR, IT, NOT, OF, OR, REPEAT, THEN, TO, UNTIL, WHILE, XOR, TRUE, FALSE, ELSIF, BY, EXIT, RETURN

  Variable names: Any text that is not a reserved word will be recognized as a variable name.

  Special characters: <=, >=, <>, :=, .., &, (*, *)

  Constants (numerical values): Numerical value only for decimal numbers
  16# followed by numerical value for hexadecimal numbers
  2# followed by numerical value for binary numbers
  8# followed by numerical value for octal numbers

  If a space, carriage return, or tab is used between any of the above tokens, the parts of the token on either side will be treated as separate tokens. Therefore, make sure that spaces, carriage returns, or tabs are not used within a single token.
- Always use a space, carriage return, tab, or other token separator between reserved words and variable names. Using token separators between other token combinations is optional.
  In the following example, the box (□) indicates where a space, carriage return, tab, or other token separator is required.

```
IF□A>0THEN□X=10;
ELSE□
  X:=0;
END_IF;
```

Upper and Lower Case

- Reserved words and variable names do not distinguish between upper and lower case (either can be used).

Prohibited Characters for Variable Names

- The following characters enclosed in square brackets cannot be used in variable names.
  [!], [“], [#], [$], [%], [&], [‘], [(],, [)], [-], [=], [^], [~], [\], [|], [@], [‘], [[], [{], [;], [+], [:], [*], []], [}], [,], [<], [.], [>], [/], [?]
- The numbers 0 to 9 cannot be used as the first character of variable names.
- An underscore cannot be followed immediately by another underscore in variable names.
- Spaces cannot be used in variable names.

An error message will occur if any of these characters are used in this way.

Inputting Constants (Numerical Values)

- Numerical values can be expressed in decimal, hexadecimal, octal, or binary, as shown in the following examples.

	Notation method	Example (for the decimal value 12)
Decimal:	Numerical value only	12
Hexadecimal:	16# followed by numerical value	16#C
Octal:	8# followed by numerical value	8#14
Binary:	2# followed by numerical value	2#1100

Operator Priority

- Consider the operator priority in the structured text syntax, or enclose operations requiring priority in parentheses.
  Example: AND takes priority over OR. Therefore, in the example X OR Y AND Z, priority will be given to Y AND Z.

CX-Programmer's ST Input Screen Display

Text Display Color

The CX-Programmer automatically displays text in the following colors when it is input or pasted in the ST Input Screen.

- Text keywords (reserved words): Blue
- Comments: Green
- Errors: Red
- Other: Black

Changing Fonts

To change font sizes or display colors, select ***Tools - Options - Display***, and then click the **ST Font** Button. The font name, font size (default is 8 point), and color can be changed.

Statements

Statement	Function	Example
End of statement	Ends the statement	;
Comment	All text between (* and *) is treated as a comment.	(**comment**)
Assignment	Substitutes the results of the expression, variable, or value on the right for the variable on the left.	A:=B;

IF, THEN, ELSIF, ELSE, END_IF	Evaluates an expression when the condition for it is true.	IF (*condition_1*) THEN
		(*expression 1*);
		ELSIF (*condition_2*) THEN
		(*expression 2*);
		ELSE
		(*expression 3*);
		END_IF;

CASE, ELSE, END_CASE	Evaluates an express based on the value of a variable.	CASE (*variable*) OF 1: (*expression 1*); 2: (*expression 2*); 3: (*expression 3*); ELSE (*expression 4*); END_CASE;
FOR, TO, BY, DO, END_FOR	Repeatedly evaluates an expression according to the initial value, final value, and increment.	FOR (*identifier*) := (*initial_value*) TO (*final_value*) BY (*increment*) DO (*expression*); END_FOR;
WHILE, DO, END_WHILE	Repeatedly evaluates an expression as long as a condition is true.	WHILE (*condition*) DO (*expression);* END_WHILE;
REPEAT, UNTIL, END_REPEAT	Repeatedly evaluates an expression until a condition is true.	REPEAT (*expression);* UNTIL (*condition*) END_REPEAT;

Statement	Function	Example
EXIT	Stops repeated processing.	EXIT;
RETURN	Returns to the point in the program from which a function block was called.	RETURN;

Operators

Operation	Symbol	Data types supported by operator	Priority 1: Lowest 11: Highest
Parentheses and brackets	(*expression*), *array*[*index*]		1
Function evaluation	*identifier*	Depends on the function (refer to *2-6 Instruction Support and Operand Restrictions*)	2
Exponential	**	REAL, LREAL	3
Complement	NOT	BOOL, WORD, DWORD, LWORD	4
Multiplication	*	INT, DINT, UINT, UDINT, ULINT, REAL, LREAL	5
Division	/	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL	5
Addition	+	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL	6
Subtraction	–	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL	6
Comparisons	<, >, <=, >=	BOOL, INT, DINT, LINT, UINT, UDINT, ULINT, WORD, DWORD, LWORD, REAL, LREAL	7
Equality	=	BOOL, INT, DINT, LINT, UINT, UDINT, ULINT, WORD, DWORD, LWORD, REAL, LREAL	8
Non-equality	<>	BOOL, INT, DINT, LINT, UINT, UDINT, ULINT, WORD, DWORD, LWORD, REAL, LREAL	8
Boolean AND	&	BOOL, WORD, DWORD, LWORD	9
Boolean AND	AND	BOOL, WORD, DWORD, LWORD	9
Boolean exclusive OR	XOR	BOOL, WORD, DWORD, LWORD	10
Boolean OR	OR	BOOL, WORD, DWORD, LWORD	11

Note Operations are performed according to the data type.
Therefore, the addition result for INT data, for example, must be a variable using the INT data type. Particularly care is required when a carry or borrow occurs in an operation for integer type variables. For example, using integer type variables A=3 and B= 2, if the operation (A/B)*2 is performed, the result of A/B is 1 (1.5 with the value below the decimal discarded), so (A/B)*2 = 2.

Functions

Function	Syntax
Numerical Functions	Absolute values, trigonometric functions, etc.
Arithmetic Functions	Exponential (EXPT)
Data type conversion Functions	*Source data type*_TO_*New data type* (*Variable name*)

Numerical Functions

The following numerical functions can be used in structured text.

Numerical functions	Argument data type	Return value data type	Contents	Example
ABS (*argument*)	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL	INT, DINT, LINT, UINT, UDINT, ULINT, REAL, LREAL	Absolute value [*argument*]	a: = ABS (b) (*absolute value of variable *b* stored in variable *a**)
SQRT (*argument*)	REAL, LREAL	REAL, LREAL	Square root: $\sqrt{}$ argument	a: = SQRT (b) (*square root of variable *b* stored in variable *a**)
LN (*argument*)	REAL, LREAL	REAL, LREAL	Natural logarithm: LOG_e argument	a: = LN (b) (*natural logarithm of variable *b* stored in variable *a**)
LOG (*argument*)	REAL, LREAL	REAL, LREAL	Common logarithm: LOG_{10} argument	a: = LOG (b) (*common logarithm of variable *b* stored in variable *a**)
EXP (*argument*)	REAL, LREAL	REAL, LREAL	Natural exponential: $e^{argument}$	a: = EXP (b) (*natural exponential of variable *b* stored in variable *a**)
SIN (*argument*)	REAL, LREAL	REAL, LREAL	Sine: SIN argument	a: = SIN (b) (*sine of variable *b* stored in variable *a**)
COS (*argument*)	REAL, LREAL	REAL, LREAL	Cosine: COS argument	a: = COS (b) (*cosine of variable *b* stored in variable *a**)
TAN (*argument*)	REAL, LREAL	REAL, LREAL	Tangent: TAN argument	a: = TAN (b) (*tangent of variable *b* stored in variable *a**)
ASIN (*argument*)	REAL, LREAL	REAL, LREAL	Arc sine: SIN^{-1} argument	a: = ASIN (b) (*arc sine of variable *b* stored in variable *a**)
ACOS (*argument*)	REAL, LREAL	REAL, LREAL	Arc cosine: COS^{-1} argument	a: = ACOS (b) (*arc cosine of variable *b* stored in variable *a**)
ATAN (*argument*)	REAL, LREAL	REAL, LREAL	Arc tangent: TAN^{-1} argument	a: = ATAN (b) (*arc tangent of variable *b* stored in variable *a**)

Note The data type returned for numerical functions is the same as that used in the argument. Therefore, variables substituted for function return values must be the same data type as the argument.

Arithmetic Functions

The following general exponential function can be used in structured text.

Exponential function	Argument data type	Return value data type	Contents	Example
EXPT (*base, exponent*)	Base: REAL, LREAL Exponent: INT, DINT, LINT, UINT, UDINT, ULINT	REAL, LREAL	Exponential: $Base^{exponent}$	a: = EXPT (b, c) (*Exponential with variable *b* as the base and variable *c* as the exponent is stored in variable *a**)

Note The data type returned for the general exponential function is the same as that used in the argument. Therefore, variables substituted for function return values must be the same data type as the argument.

Data Type Conversion Functions

The following data type conversion functions can be used in structured text.

Syntax

*Source data type*_TO_*New data type* (*Variable name*)

Example: REAL_TO_INT (C)

In this example, the data type for variable *C* will be changed from REAL to INT.

Data Type Combinations

The combinations of data types that can be converted are given in the following table.
(YES = Conversion possible, No = Conversion not possible)

FROM	TO											
	BOOL	INT	DINT	LINT	UINT	UDINT	ULINT	WORD	DWORD	LWORD	REAL	LREAL
BOOL	No	No	No	No	No	No	No	No	No	No	No	No
INT	No	No	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
DINT	No	YES	No	YES	YES	YES	YES	YES	YES	YES	YES	YES
LINT	No	YES	YES	No	YES	YES	YES	YES	YES	YES	YES	YES
UINT	No	YES	YES	YES	No	YES	YES	YES	YES	YES	YES	YES
UDINT	No	YES	YES	YES	YES	No	YES	YES	YES	YES	YES	YES
ULINT	No	YES	YES	YES	YES	YES	No	YES	YES	YES	YES	YES
WORD	No	YES	YES	YES	YES	YES	YES	No	YES	YES	No	No
DWORD	No	YES	YES	YES	YES	YES	YES	YES	No	YES	No	No
LWORD	No	YES	YES	YES	YES	YES	YES	YES	YES	No	No	No
REAL	No	YES	YES	YES	YES	YES	YES	No	No	No	No	YES
LREAL	No	YES	YES	YES	YES	YES	YES	No	No	No	YES	No

Statement Details

Assignment

Summary

The left side of the statement (variable) is substituted with the right side of the statement (equation, variable, or constant).

Reserved Words

:=

Combination of colon (:) and equals sign (=).

Statement Syntax

Variable: = Equation, variable, or constant;

Usage

Use assignment statements for inputting values in variables. This is a basic statement for use before or within control statements. This statement can be used for setting initial values, storing calculation results, and incrementing or decrementing variables.

Description

Substitutes (stores) an *equation*, *variable*, or *constant* for the *variable*.

Examples

Example 1: Substitute variable A with the result of the equation X+1.

```
A:=X+1;
```

Example 2: Substitute variable A with the value of variable B.

```
A:=B;
```

Example 3: Substitute variable A with the constant 10.

```
A:=10;
```

Precautions

The data type of the equation, variable, or constant to be assigned must be the same as the data type of the variable to be substituted. Otherwise, a syntax error will occur.

Control Statements

IF Statement (Single Condition)

Summary

This statement is used to execute an expression when a specified condition is met. If the condition is not met, a different expression is executed.

Reserved Words

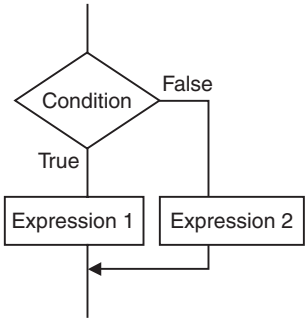
IF, THEN, (ELSE), END_IF

Note ELSE can be omitted.

Statement Syntax

```
IF <condition> THEN
    <expression_1>;
ELSE
    <expression_2>;
END_IF;
```

Process Flow Diagram



Usage

Use the IF statement to perform a different operation depending on whether a single condition (condition equation) is met.

Description

Condition = If true, execute *expression_1*

Condition = If false, execute *expression_2*

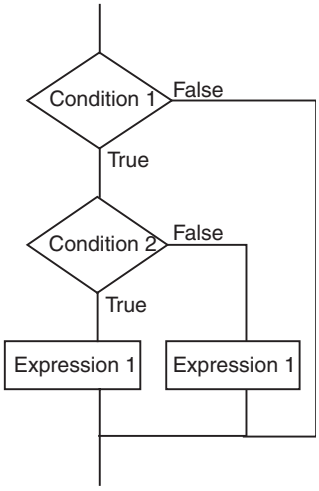
Precautions

- IF must be used together with END_IF.
- The *condition* must include a true or false equation for the evaluation result.
  Example: IF(A>10)
  The *condition* can also be specified as a boolean variable only rather than an equation. As a result, the variable value is 1 (ON) = True result, 0 (OFF) = False result.
- Statements that can be used in *expression_1* and *expression_2* are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.

Example:

```
IF <condition_1> THEN
 IF <condition_2> THEN
  <expression_1>;
 ELSE
  <expression_2>:
  END_IF;
END_IF;
```

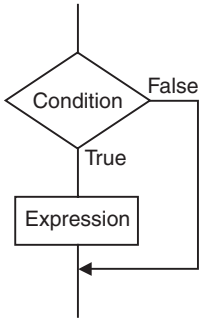
The processing flow diagram is as follows:



ELSE corresponds to THEN immediately before it, as shown in the above diagram.

- Multiple statements can be executed within *expression_1* and *expression_2.* Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- The ELSE statement can be omitted. When ELSE is omitted, no operation is executed if the result of the *condition* equation is false.

Processing Flow Diagram



Examples

Example 1: If variable A>0 is true, variable X will be substituted with numerical value 10. If A>0 is false, variable X will be substituted with numerical value 0.

```
IF A>0 THEN
  X:=10;
ELSE
  X:=0;
END_IF;
```

Example 2: If variable A>0 and variable B>1 are both true, variable X will be substituted with numerical value 10, and variable Y will be substituted with numerical value 20. If variable A>0 and variable B>1 are both false, variable X and variable Y will both be substituted with numerical value 0.

```
IF A>0 AND B>1 THEN
  X:=10; Y:=20;
ELSE
  X:=0; Y:=0;
END_IF;
```

Example 3: If the boolean (BOOL data type) variable A=1(ON), variable X will be substituted with numerical value 10. If variable A=0(OFF), variable X will be substituted with numerical value 0.

```
IF A THEN X:=10;
ELSE X:=0;
END_IF;
```

IF Statement (Multiple Conditions)

Summary

This statement is used to execute an expression when a specified condition is met. If the first condition is not met, but another condition is met, a corresponding expression is executed. If none of the conditions is met, a different expression is executed.

Reserved Words

IF, THEN, ELSIF, (ELSE)

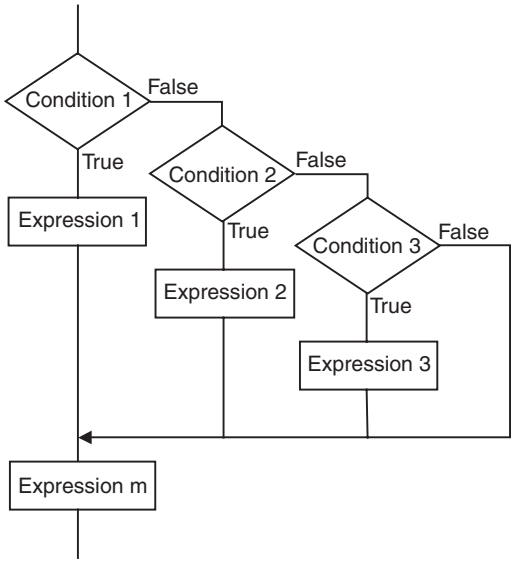
Note ELSE can be omitted.

Statement Syntax

```
IF <condition_1> THEN <expression_1>;
     ELSIF <condition_2> THEN <expression_2>;
     ELSIF <condition_3> THEN <expression_3>;

             ...
     ELSIF <condition_n> THEN <expression_n>;
ELSE <expression_m>;
END_IF;
```

Processing Flow Chart



Usage

Use the IF statement to perform different operations depending which of multiple conditions (*condition* equation) is met.

Description

Condition 1 = If true, execute *expression 1*

Condition 1 = If false,

Condition 2 = If true, execute *expression 2*

Condition 2 = If false,

Condition 3 = If true, execute *expression 3*

etc.

Condition n = If true, execute *expression n*

If none of these conditions are met, *condition m* is executed.

Precautions

- IF must be used together with END_IF.
- *Condition_*□ contains the true or false result of the equation (e.g., IF(A>10)).
  A boolean (BOOL data type) variable only can also be specified as the *condition* rather than an equation. For boolean conditions, the result is true when the variable value is 1 (ON) and false when it is 0 (OFF).
- Statements that can be used in *expression_*□ are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.
- Multiple statements can be executed in *expression_*□. Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- The ELSE statement can be omitted. When ELSE is omitted, no operation is executed if the result of any *condition* equation is false.

Examples

Example 1: If variable A>0 is true, variable X will be substituted with numerical value 10.
If A>0 is false, but variable B=1, variable X will be substituted with numerical value 1.
If A>0 is false, but variable B=2, variable X will be substituted with numerical value 2.
If either of these conditions is met, variable X will be substituted with numerical value 0.

```
IF A>0 THEN X:=10;
          ELSIF B=1 THEN X:=1;
          ELSIF B=2 THEN X:=2;
ELSE X:=0;
END_IF;
```

CASE Statement

Summary

This statement executes an expression containing a selected integer that matches the value from an integer equation. If the selected integer value is not the same, either no expression or a specified expression is executed.

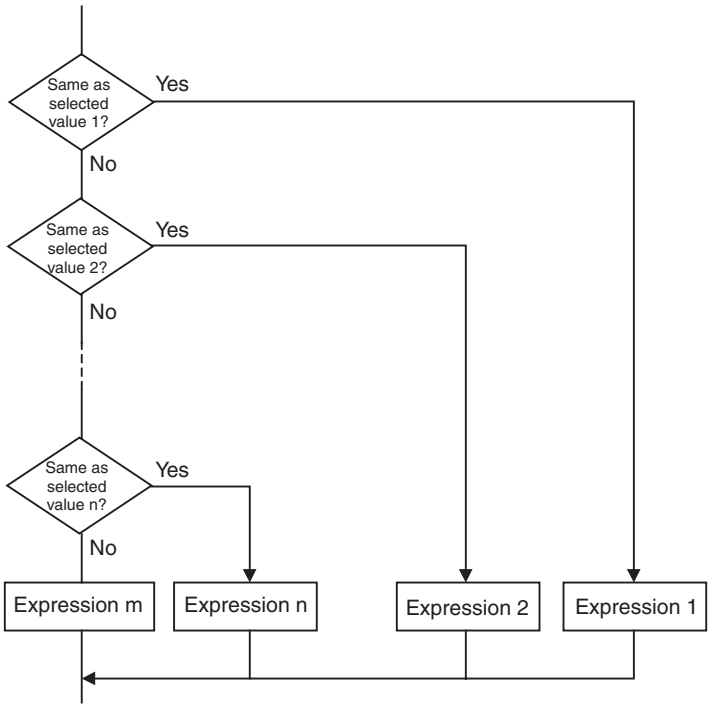
Reserved Word

CASE

Statement Syntax

```
CASE <integer_equation> OF
      <integer_equation_value_1 >:<expression_1>;
      <integer_equation_value_2>:<expression_2>;
                    ...
      <integer_equation_value_n>:<expression_n>;
ELSE <expression_m>;
END_CASE;
```

Processing Flow Chart



Usage

Use the CASE statement to execute different operations depending on specified integer values.

Description

If the *integer_equation* matches *integer_equation_value_n*, *expression_n* is executed.
if the *integer_equation* does not match any of *integer_equation_value_n*, *expression_m* is executed.

Precautions

- CASE must be used together with END_CASE.
- The result of the *integer_equation* must be in integer format (INT, DINT, LINT, UINT, UDINT, or ULINT).
- Statements that can be used in *expression_□* are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.
- Multiple statements can be executed in *expression_□*. Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- Variables in integer format (INT, DINT, LINT, UINT, UDINT, or ULINT), or equations that return integer values can be specified in the *integer_equation*.
- When OR logic is used for multiple integers in the *integer_equation_value_n*, separate the numerical value using a comma delimiter. To specify a sequence of integers, use two periods (..) as delimiters between the first and last integers.

Examples

Example 1: If variable A is 1, variable X is substituted with numerical value 1. If variable A is 2, variable X is substituted with numerical value 2. If variable A is 3, variable X is substituted with numerical value 3. If neither of these cases matches, variable Y will be substituted with 0.

```
CASE A OF
        1:X:=1;
        2:X:=2;
        3:X:=3;
ELSE Y:=0;
END_CASE;
```

Example 2: If variable A is 1, variable X is substituted with numerical value 1. If variable A is 2 or 5, variable X is substituted with numerical value 2. If variable A is a value between 6 and 10, variable X is substituted with numerical value 3. If variable A is 11, 12, or a value between 15 and 20, variable X is substituted with numerical value 4. If neither of these cases matches, variable Y will be substituted with 0.

```
CASE A OF
1:X:=1;
2,5:X:=2;
6..10:X:=3;
11,12,15..20:X:=4;
ELSE Y:=0;
END_CASE;
```

FOR Statement

Summary

This statement is used to execute a specified expression repeatedly until a variable (referred to here as an iteration variable) reaches a specified value.

Reserved Words

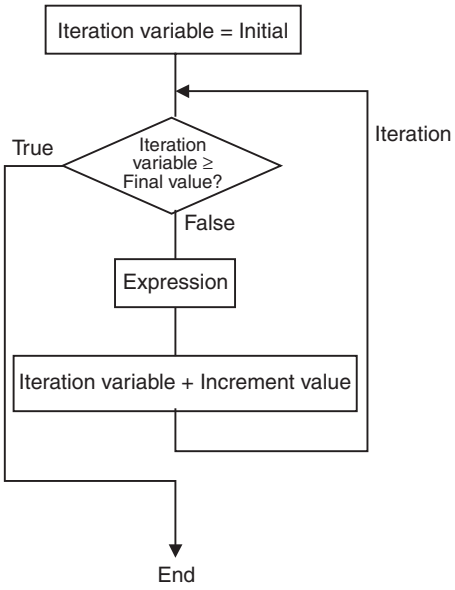
FOR, TO, (BY), DO, END_FOR

Note BY can be omitted.

Statement Syntax

FOR <*iteration_variable*>:= <*initial_value*> TO <*final_value_equation*> BY <*increment_value_equation*>
DO
 <*expression*>;
END_FOR;

Processing Flow Chart



Usage

Use the FOR statement when the number of iterations has been determined beforehand. FOR is particularly useful when switching the number of elements in an array variable according to the value of a specified iteration variable.

Description

When the *iteration_variable* is the *initial_value*, the *expression* is executed. After execution, the value obtained from the *increment_equation* is added to the *iteration_variable*, and if the *iteration_variable* < *final_value_equation* (see note 1), the *expression* is executed. After execution, the value obtained from the *increment_equation* is added to the *iteration_variable*, and if the *iteration_variable* < *final_value_equation* value (see note 1), the *expression* is executed. This process is repeated. If the *iteration_variable* ≥ *final_value_equation* (see note 2), the processing ends.

Note (1) If the value from the *increment_equation* is negative, the condition is *iteration_variable* > *final_value_equation* value.

(2) If the value from the *increment_equation* is negative, the condition is *iteration_variable* ≤ *final_value_equation.*

Precautions

- A negative value can be specified in the *increment_equation*
- FOR must be used in combination with END_FOR.
- The *initial_value*, *final_value_equation*, and *final_value_equation* must be an integer data type (INT, DINT, LINT, UINT, UDINT, or ULINT).
- Do not reference an iteration variable that is outside the FOR statement, where possible. The value of the iteration variable after completing execution of the FOR statement depends on the actual model specifications, and may prevent use of the program for general-purpose applications.
  Example: In the following structured text, whether the value of *a* is TRUE or FALSE depends on the actual model being used.

```
FOR i:=0 TO 100 DO
  array[i]:=0;
END_FOR;


IF i=101 THEN
   a:=TRUE;
ELSE
   a:=FALSE;
END_IF;
```

- Do not use a FOR statement in which an iteration variable is changed directly. Doing so may result in unexpected operations.
  Example:

```
FOR i:=0 TO 100 BY 1 DO
  array[i]:=0;
  i:=i+5;
END_FOR;
```

- Statements that can be used in the *expression* are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.
- Multiple statements can be executed in the *expression.* Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- BY *increment_equation* can be omitted. When omitted, BY is taken as 1.
- Variables with integer data types (INT, DINT, LINT, UINT, UDINT, or ULINT), or equations that return integer values can be specified in the *initial_value, final_value_equation,* and *increment_equation.*
  Example 1: The iteration is performed when the iteration variable n = 0 to 50 in increments of 5, and the array variable SP[n] is substituted with 100.

```
FOR n:=0     TO 50  BY 5 DO
  SP[n]:=100;
END_FOR;
```

Example 2: The total value of elements DATA[1] to DATA[50] of array variable DATA[n] is calculated, and substituted for the variable SUM.

```
FOR n:=0     TO 50  BY 1 DO
  SUM:=SUM+DATA[n];
END_FOR;
```

Example 3: The maximum and minimum values from elements DATA[1] to DATA[50] of array variable DATA[n] are detected. The maximum value is substituted for variable MAX and the minimum value is substituted for variable MIN. The value for DATA[n] is between 0 and 1000.

```
MAX:=0;
MIN:=1000;
FOR n:=1     TO 50  BY 1 DO
  IF DATA[n]>MAX THEN
  MAX:=DATA[n];
  END IF;
  IF DATA[n]<MIN THEN
  MIN:=DATA[n];
  END IF;
END_FOR;
```

WHILE Statement

Summary

This statement is used to execute a specified expression repeatedly for as long as a specified condition is true.

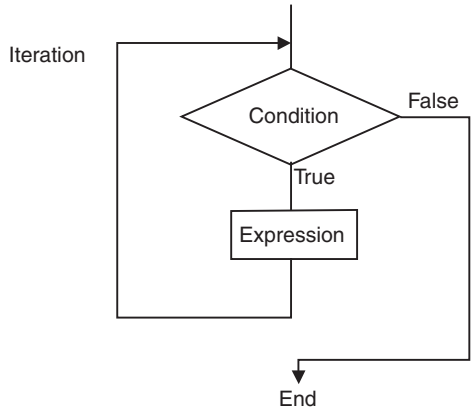
Reserved Words

WHILE, DO, END_WHILE

Statement Syntax

```
WHILE <condition> DO
 <expression>;
END_WHILE;
```

Processing Flow Chart



Usage

Use the WHILE statement when the number of iterations has not been determined beforehand (depends on the condition being met) to repeat specified processing for the duration that the condition is met. This statement can be used to execute processing while the condition equation is true only (pretest loop).

Description

Before the *expression* is executed, the *condition* is evaluated.
If the *condition* is true, the *expression* is executed. Afterwards, the *condition* is evaluated again. This process is repeated. If the *condition* is false, the *expression* is not executed and the *condition* evaluation ends.

Precautions

- WHILE must be used in combination with END_WHILE.
- Before executing the *expression*, if the *condition* equation is false, the process will end without executing the *expression.*
- Statements that can be used in the *expression* are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.
- Multiple statements can be executed in the *expression*. Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- The *condition* can also be specified as a boolean variable (BOOL data type) only rather than an equation.

Examples

Example 1: The value exceeding 1000 in increments of 7 is calculated and substituted for variable A.

```
A:=0;
WHILE A>=1000 DO
 A:=A+7;
END_WHILE;
```

Example 2: While X<3000, the value of X is doubled, and the value is substituted for the array variable DATA[1]. The value of X is then multiplied by 2 again, and the value is substituted for the array variable DATA[2]. This process is repeated.

```
n:=1'
WHILE X<3000 DO
 X:=X*2;
 DATA[n]:=X;
 n:=n+1;
END_WHIE;
```

REPEAT Statement

Summary

This statement is used to repeatedly execute an expression until a specified condition is true.

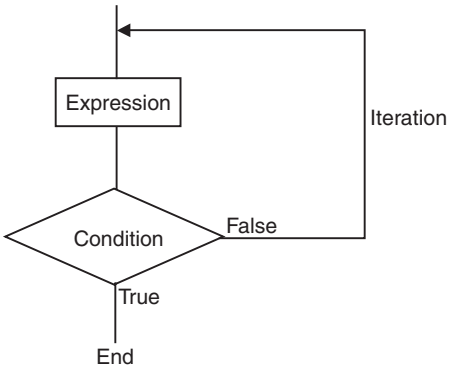
Reserved Words

REPEAT, UNTIL, END_REPEAT

Statement Syntax

```
REPEAT
 <expression>;
UNTIL <condition>
END_REPEAT
```

Processing Flow Chart



Usage

Use the REPEAT statement to repeat processing for as long as a condition is met after specified processing, when the number of iterations is undetermined beforehand (depends on whether the condition is met). This statement can be used to determine whether to repeat processing according to the results of specified processing execution (post-test loop).

Description

The *expression* will execute the first time without a condition. Thereafter, the *condition* equation will be evaluated. If the *condition* is false, the *expression* will be executed again. If the *condition* is true, processing will end without executing the *expression.*

Precautions

- REPEAT must be used together with END_REPEAT.
- Even if the *condition* equation is true before the *expression* has been executed, the *expression* will be executed.
- Statements that can be used in the *expression* are assignment statements, IF, CASE, FOR, WHILE, or REPEAT.
- Multiple statements can be executed in the *expression*. Be sure to use a semicolon (;) delimiter between multiple statements in an *expression*.
- The *condition* can also be specified as a boolean variable (BOOL data type) only rather than an equation.

Examples

Example 1: Numeric values from 1 through 10 are incremented and the total is substituted for the variable TOTAL.

```
A:=1;
TOTAL:=0;
REPEAT
  TOTAL:=TOTAL+A;
  A:=A+1;
UNTIL A>10
END_REPEAT;
```

EXIT Statement

Summary

This statement is used within iteration statements (FOR, WHILE, REPEAT) only to force an iteration statement to end. This statement can also be used within an IF statement to force an iteration statement to end when a specified condition is met.

Reserved Words

EXIT

Statement Syntax (Example: Using within IF Statement)

```
FOR (WHILE, REPEAT) expression
                                ...
IF <condition> THEN EXIT;
END_IF;

                                ...
END_FOR (WHILE, REPEAT);
```

Usage

Use the EXIT statement to force iteration processing to end before the end condition is met.

Description (Example: Using within IF Statement)

When the *condition* equation is true, the iteration statement (FOR, WHILE, REPEAT) is forced to end, and any statements after EXIT will not be executed.

Note (1) The *condition* can also be specified as a boolean variable (BOOL data type) only rather than an equation.

(2) Even if the *condition* equation is true before the *expression* has been executed, the *expression* will be executed.

Example

Processing is repeated from when variable n = 1 until 50 in increments of 1 and n is added to array variable DATA[n]. If DATA[n] exceeds 100, however, processing will end.

```
FOR n:=1; TO 50 BY 1 DO
  DATA[n]:=DATA[n]+n;
  IF DATA[n]>100 THEN EXIT;
  END_IF;
END_FOR;
```

RETURN Statement

Summary

This statement is used to execute the next instruction following the location that called the function block in the program when the function block in the structured text must be forced to end before it has been completed.

Reserved Words

RETURN

Statement Syntax

```
RETURN;
```

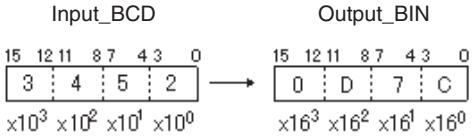
Usage

Use the RETURN statement when a function block has been forced to end.

Examples of Structured Text Programming

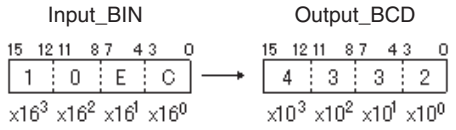
Example 1: Conversion of BCD Data (#0000-#9999) to BIN Data

```
(*Check the input parameter "Input_BCD" (BCD data) *)
IF (Input_BCD>=0 & Input_BCD<=16#9999) THEN
   ENO:=true;
ELSE
   ENO:=false;
   RETURN;
END_IF;
(*BCD data is divided by 16 four times to get each digit of the BIN data converted from the BCD data*)
DIV_1:=Input_BCD/16;
DIV_2:=DIV_1/16;
DIV_3:=DIV_2/16;
DIV_4:=DIV_3/16;
(*Calculate each digit of the BIN data converted from the BCD data*)
BIN_1:=Input_BCD-16*DIV_1;   (*a number of 16^0 digit*)
BIN_2:=DIV_1-16*DIV_2;   (*a number of 16^1 digit*)
BIN_3:=DIV_2-16*DIV_3;   (*a number of 16^2 digit*)
BIN_4:=DIV_3-16*DIV_4;   (*a number of 16^3 digit*)
(*Calculate the BIN data "Output_BIN" (output parameter) *)
Output_BIN:=BIN_1+BIN_2*10+BIN_3*10*10+BIN_4*10*10*10;
```



Example 2: Conversion of BIN Data (#0000-#FFFF) to BCD Data

```
(*check the input parameter "Input_BIN" (BIN data) *)
IF (Input_BIN>=0 & Input_BIN<=16#FFFF) THEN
   ENO:=true;
ELSE
   ENO:=false;
   RETURN;
END_IF;
(*BIN data is divided by 10 four times to get each digit of the BCD data converted from the BIN data*)
DIV_1:=Input_BIN/10;
DIV_2:=DIV_1/10;
DIV_3:=DIV_2/10;
DIV_4:=DIV_3/10;
(*Calculate each digit of the BCD data converted from the BIN data*)
BCD_1:=Input_BIN-10*DIV_1;   (*a number of 10^0 digit*)
BCD_2:=DIV_1-10*DIV_2;   (*a number of 10^1 digit*)
BCD_3:=DIV_2-10*DIV_3;   (*a number of 10^2 digit*)
BCD_4:=DIV_3-10*DIV_4;   (*a number of 10^3 digit*)
(*Calculate the BCD data "Output_BCD" (output parameter) *)
Output_BCD:=BCD_1+BCD_2*16+BCD_3*16*16+BCD_4*16*16*16;
```



Restrictions

Nesting

There is no restriction on the number of nests that can be used in IF, CASE, FOR, WHILE, or REPEAT statements.

Data Type Restrictions

- Integers can only be allocated to variables with data types WORD, DWORD, INT, DINT, UINT, UDINT, or ULINT. For example, if A is an INT data type, A:=1; it possible. If the value is not an integer data type, a syntax error will occur. For example, if A is an INT data type, a syntax error will occur for A:=2.5;.
- If a real number (floating point decimal data) can only be allocated to variables with data types REAL and UREAL. For example, if A is a REAL data type, A:=1.5; is possible. If the value is not an real data type, a syntax error will occur. For example, if A is a REAL data type, a syntax error will occur for A:=2;. Use A:=2.0;.
- Bits (TRUE, FALSE) can only be allocated to variables with the BOOL data type. For example, if A is a BOOL data type, A:=FALSE; is possible. If a BOOL data type is not used, a syntax error will occur. For example, if A is an INT data type, a syntax error will occur for A:=FALSE;.
- Data types must all be consistent within the structured text. For example, if A, B, and C are INT data types, A:=B+C; is possible. If, however, A and B are INT data types, but C is a REAL data type or LINT data type, a syntax error will occur for A:=B+C;.

Structured Text Errors

Error Messages

Error Message	Cause of error	Example
%s' Input variables cannot be assigned a value	A value was substituted for an input variable.	
%s' operator not supported by %s data type	A numerical value or variable for a data type that is not supported by the operator was used.	A:=B+1; (*A and B are WORD type variables*))
%s' variable had a read only memory AT Address and cannot be assigned a value	A value was substituted for a variable allocated to a read-only memory address (read-only Auxiliary Area address or Condition Flag).	
Array index out of range	An array index larger than the array size was specified.	Array[100]:=10; (**Array* is an array variable with an array size of 100*)
Conversion cannot convert from %s to %s	A numeric equation in which the data type of the operation result does not match the variable at the substitution destination and a variable that is different from the data type was substituted.	Y:=ABS(X); (*X is an INT type variable, Y is a UINT type variable*)
Division by Zero	The numeric expression contains division by 0.	
End of comment not found	The comment does not have a closing parenthesis and asterisk "*)" corresponding to the opening parenthesis and asterisk "(*" of the comment.	(**comment*
Invalid Literal Format '%s'	The numeric format is illegal.	X:=123_; (*There is no numeral after underscore*) X:=1__23; (*The underscore is followed immediately by another underscore*) X:=2#301; Y:=8#90; (*A numeral that cannot be used with binary or octal values has been used*) **Note** The underscore can be inserted between numerals to make them easier to read. Placing 2#, 8#, and 16# at the beginning of the numeral expresses the numerals as binary, octal, and hexadecimal values, respectively.
Invalid Literal Value	The numeric value is illegal.	X:=1e2; (*an index was used for a numeric value that was not a REAL data type*) **Note** "e" indicates an exponent of 10.
Invalid array index	A numeric equation with a non-integer type operation result or a non-integer variable has been specified in the array index.	Array[Index]:=10; (**Index* is a WORD type variable*)
Invalid constant	A numeric equation with a non-integer type operation result, or a non-integer variable or numeric value has been specified in the integer equation of a CASE statement.	CASE A OF (*A is a REAL type variable*) 1: X:=1; 2: X:=2; END_CASE;

Error Message	Cause of error	Example
Invalid expression	The numeric equation is illegal. For example, the integer equation or condition equation is illegal or has not been specified in the syntax (IF, WHILE, REPEAT, FOR, CASE).	WHILE DO (*The WHILE statement does not contain a condition equation*) X:=X+1; END_WHILE;
Invalid parameter in FOR loop declaration	A variable with data type other than INT, DINT, LINT, UINT, UDINT, or ULINT has been used for variables in a FOR statement.	FOR I:=1 TO 100 DO (*I is a WORD type variable*) X:=X+1; END_FOR;
Invalid statement	The statement is illegal. E.g., The statement (IF, WHILE, REPEAT, FOR, CASE, REPEAT) does not contain an IF, WHILE, REPEAT, FOR, CASE, or REPEAT in the syntax, respectively.	X:=X+1; (*There is no REPEAT in the syntax*) UNTIL X>10 END_REPEAT;
Invalid variable for Function output	The specified variable for the function output is illegal (A non-boolean (BOOL) variable or numeral has been specified as the ENO transfer destination.)	Y:=SIN(X1, ENO=>1);
Missing (	The call for a data format conversion instruction or function does not contain a “(“ (opening parenthesis).	Y:=INT_TO_DINT X);
Missing)	The operator parentheses or the call for a data format conversion instruction or function does not contain a “)“ (closing parenthesis) corresponding to “(“ (opening parenthesis).	Y:=(X1+X2/2
Missing :	The integer equation in the CASE statement is not followed by a “:” (colon).	CASE A OF 1 X:=1; END_CASE;
Missing :=	“:=” is not included in the assignment equation.	
Missing ;	The statement is not concluded by a “;” (semicolon).	
Missing DO	“DO” is not provided in the FOR or WHILE statement.	
Missing END_CASE	“END_CASE” is not provided at the end of the CASE statement.	
Missing END_FOR	“END_FOR” is not provided at the end of the FOR statement.	
Missing END_IF	“END_IF” is not provided at the end of the IF statement.	
Missing END_REPEAT	“END_REPEAT” is not provided at the end of the REPEAT statement.	
Missing END_WHILE	“END_WHILE” is not provided at the end of the WHILE statement.	
Missing Input Parameter. All input variables must be set.	The function argument is not specified or is insufficient.	Y:=EXPT(X);

Error Message	Cause of error	Example
Missing OF	"OF" is not included in CASE statement.	
Missing THEN	"THEN" is not included in IF statement.	
Missing TO	"TO" is not included in FOR statement.	
Missing UNTIL	"UNTIL" is not included in REPEAT statement.	
Missing [	The array index for the array variable has not been specified.	X:=Array; (**Array* is an array variable*)
Missing]	The array index for the array variable has not been specified.	X:=Array[2; (**Array* is an array variable*)
Missing constant	A constant is not provided in the integer equation of the CASE statement.	CASE A OF 2..: X:=1; 2,: X:=2; END_CASE;
NOT operation not supported on a literal number	The NOT operator was used for a numeric value.	Result:=NOT 1;
Negation not supported by %s data type	A minus symbol was used before a variable with a data type that does not support negative values (UINT, UDINT, ULINT).	Y:=-X; (*X is an UINT type variable, Y is an INT type variable*)
There must be one line of valid code (excluding comments)	There is no line of valid code (excluding comments).	
Too many variables specified for Function	Too many parameter settings are specified for the function.	Y:=SIN(X1,X2);
Undefined identifier '%s'	A variable that is not defined in the variable table has been used.	
Unexpected syntax '%s'	A keyword (reserved word) or variable has been used illegally.	FOR I:=1 TO 100 DO BY -1 (*The DO position is illegal*) X:=X+1; END_FOR;
Usage mismatch in Function variable	The function parameter has been used illegally.	Y:=SIN(X1,EN=>BOOL1); (*The input parameter EN has been used as an output parameter*)
Value out of range	A value outside the range for the variable data type has been substituted in the variable.	X:=32768; (*X is an INT type variable*)
Variable '%s' is not a Function parameter	A variable that cannot be specified in the function parameter has been specified in the parameter.	Y:=SIN(Z:=X); (*X and Y are REAL type variables, and Z is not a SIN function parameter *)

Warning Messages

Warning message	Cause of warning	Example
Keyword '%s' is redundant	The keyword has been used in an invalid location. For example, use of the EXIT statement outside a loop syntax.	
Conversion from '%s' to '%s', possible loss of data	Data may be lost due to conversion of a data type with a large data size to a data type with a small data size.	Y:=DINT_TO_INT(X); (*X is a DINT type variable, Y is an INT type variable*)

Commonly Asked Questions

Q: How is a hexadecimal value expressed?

A: Add “16#” before the value, e.g., 16#123F.

The prefixes 8# and 2# can also be added to express octal numbers and binary numbers, respectively. Numbers without these prefixes will be interpreted as decimal numbers.

Q: How many times can FOR be used?

A: In the following example, the contents of the FOR statement is executed 101 times. The loop processing ends when the value of “i” is equal to 101.

```
FOR i:=0 TO 100 BY 1 DO
  a:=a+1;
END_FOR;
```

Q: What occurs when the array subscript is exceeded?

A: For the array variable INT[10] with 10 elements, an error will not be detected for the following type of statement. Operation will be unstable when this statement is executed.

```
i:=15;
INT[i]:=10;
```

Q: Are the variables in the structured text editor automatically registered in the variable tables?

A: No. Register the variables in the variable table before using them.

Q: Can ladder programming instructions be called directly?

A: No.

Appendix C
External Variables

Classification	Name	External variable in CX-Programmer	Data type	Address
Conditions Flags	Greater Than or Equals (GE) Flag	P_GE	BOOL	CF00
	Not Equals (NE) Flag	P_NE	BOOL	CF001
	Less Than or Equals (LE) Flag	P_LE	BOOL	CF002
	Instruction Execution Error (ER) Flag	P_ER	BOOL	CF003
	Carry (CY) Flag	P_CY	BOOL	CF004
	Greater Than (GT) Flag	P_GT	BOOL	CF005
	Equals (EQ) Flag	P_EQ	BOOL	CF006
	Less Than (LT) Flag	P_LT	BOOL	CF007
	Negative (N) Flag	P_N	BOOL	CF008
	Overflow (OF) Flag	P_OF	BOOL	CF009
	Underflow (UF) Flag	P_UF	BOOL	CF010
	Access Error Flag	P_AER	BOOL	CF011
	Always OFF Flag	P_Off	BOOL	CF114
	Always ON Flag	P_On	BOOL	CF113
Clock Pulses	0.02 second clock pulse bit	P_0_02s	BOOL	CF103
	0.1 second clock pulse bit	P_0_1s	BOOL	CF100
	0.2 second clock pulse bit	P_0_2s	BOOL	CF101
	1 minute clock pulse bit	P_1mim	BOOL	CF104
	1.0 second clock pulse bit	P_1s	BOOL	CF102
Auxiliary Area Flags/ Bits	First Cycle Flag	P_First_Cycle	BOOL	A200.11
	Step Flag	P_Step	BOOL	A200.12
	First Task Execution Flag	P_First_Cycle_Task	BOOL	A200.15
	Maximum Cycle Time	P_Max_Cycle_Time	UDINT	A262
	Present Scan Time	P_Cycle_Time_Value	UDINT	A264
	Cycle Time Error Flag	P_Cycle_Time_Error	BOOL	A401.08
	Low Battery Flag	P_Low_Battery	BOOL	A402.04
	I/O VerIFication Error Flag	P_IO_Verify_Error	BOOL	A402.09
	Output OFF Bit	P_Output_Off_Bit	BOOL	A500.15
OMRON FB Library words (see note)	CIO Area specification	P_CIO	WORD	A450
	HR Area specification	P_HR	WORD	A452
	WR Area specification	P_WR	WORD	A451
	DM Area specification	P_DM	WORD	A460
	EM0 to C Area specification	P_EM0 to P_EMC	WORD	A461 to A473

Note These words are external variables for the OMRON FB Library. Do not use these words for creating function blocks.

Index

A

addresses
  allocation areas, 31
  checking internal allocations, 129
  setting allocation areas, 128
algorithm
  creating, 119
applications
  precautions, xiii
array settings, 14, 28, 44, 121
AT settings, 14, 27, 121
  restrictions, 38

C

compiling, 131
computer system requirements, 3

D

data types, 14, 27, 137
  determining, 42
debugging function blocks, 134
differentiation
  restrictions, 38

E

errors
  function blocks, 41
external variables, 26
  list, 161
externals, 14

F

features, 2
files
  function block definitions, 133
  library, 5
  project text files, 5
function block definitions, 8
  checking for an instance, 131
  compiling, 131
  creating, 114
  saving to files, 133
function blocks
  advantages, 7
  application guidelines, 42
  creating, 17, 111
  debugging, 134
  defining, 117
  elements, 21
  errors, 41
  monitoring, 134
  operating specifications, 35
  outline, 7
  restrictions, 37
  reusing, 18
  setting parameters, 125
  specifications, 4, 21
  structure, 8
functions, 2
  function blocks, 4
  restrictions, 3

G

global symbol table, 12

I

IEC 61131-3, 2, 4
input variables, 23
inputs, 13
instance areas, 15, 31
  setting, 16, 128
instances
  creating, 17, 124
  multiple, 33
  number of, 10
  outline, 9
  registering in global symbol table, 12
  specifications, 30
internal variables, 25
internals, 13

L

ladder programming
  function block definition, 116
  restrictions, 40
  restrictions in function blocks, 37

M

menus, 5
  main, 5
  popup, 6
monitoring function blocks, 134

O

online editing
  restrictions, 41
output variables, 24
outputs, 14

P

parameters
  outline, 10
precautions, xi
  applications, xiii
  general, xii
  safety, xii
Programming Consoles, 41
projects
  creating, 114

S

safety precautions, xii
specifications, 19
  CX-Programmer Ver. 5.0, 3
  function block operation, 35
  instances, 30
structured text
  function block definition, 116
  restrictions, 40
symbol name
  automatically generating, 123

T

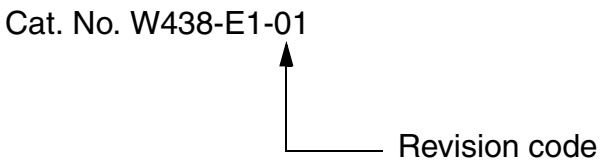
timer instructions
  operation, 107
  restrictions, 39

V

variable names, 14
variables
  address allocations, 15
  checking address allocations, 129
  creating as needed, 120
  definitions, 22
  introduction, 13
  properties, 13, 14, 27
  registering in advance, 117
  restrictions, 38
  setting allocation areas, 15
  usage, 13, 22

Revision History

A manual revision code appears as a suffix to the catalog number on the front cover of the manual.



The following table outlines the changes made to the manual during each revision. Page numbers refer to the previous version.

Revision code	Date	Revised content
01	July 2004	Original production

OMRON CORPORATION
FA Systems Division H.Q.
66 Matsumoto
Mishima-city, Shizuoka 411-8511
Japan
Tel: (81)55-977-9181/Fax: (81)55-977-9045

Regional Headquarters

OMRON EUROPE B.V.
Wegalaan 67-69, NL-2132 JD Hoofddorp
The Netherlands
Tel: (31)2356-81-300/Fax: (31)2356-81-388

OMRON ELECTRONICS LLC
1 East Commerce Drive, Schaumburg, IL 60173
U.S.A.
Tel: (1)847-843-7900/Fax: (1)847-843-8568

OMRON ASIA PACIFIC PTE. LTD.
83 Clemenceau Avenue,
#11-01, UE Square,
Singapore 239920
Tel: (65)6835-3011/Fax: (65)6835-2711

OMRON

Authorized Distributor:

Cat. No. W438-E1-01 Note: Specifications subject to change without notice Printed in Japan

This manual is printed on 100% recycled paper.

Cat. No. W438-E1-01

SYSMAC WS02-CXPC1-E-V50 CX-Programmer Ver. 5.0

OPERATION MANUAL Function Blocks

OMRON

Automatización Eléctrica
Especialistas en Automatización

Below is a list of articles with direct links to our shop Electric Automation Network where you can see:

- Quote per purchase volume in real time.
- Online documentation and datasheets of all products.
- Estimated delivery time enquiry in real time.
- Logistics systems for the shipment of materials almost anywhere in the world.
- Purchasing management, order record and tracking of shipments.

To access the product, click on the green button.

Product	Code	Reference	Product link
Control system, CPU 5120 E / S 60KW 128kW RS232C Data	135644	CS1H-CPU65H	Buy on EAN
Control system, CPU 1280 E / S 30KW 64kW RS232C Data	135642	CS1G-CPU44H	Buy on EAN
Control system, CPU 640 E / S 20Kpasos 32KW	184020	CJ1M-CPU13	Buy on EAN
Control system, CPU 160 E / S 5Kpasos 32KW	184018	CJ1M-CPU11	Buy on EAN
Control system, CPU 960 E / S 20KW 64kW RS232C Data	147099	CS1G-CPU43H	Buy on EAN
Control system, CPU 5120 E / S 120KW 256KW RS232C Data	144821	CS1H-CPU66H	Buy on EAN
Control system, CPU 320 E / S 10Kpasos 32KW	184019	CJ1M-CPU12	Buy on EAN
Control system, CPU 640 E / S 20Kpasos 32KW 16 E / S	204821	CJ1M-CPU23	Buy on EAN
Control system, CPU 320 E / S 10Kpasos 32KW 16 E / S	204820	CJ1M-CPU22	Buy on EAN
Control system, CPU 5120 E / S 30KW 64kW RS232C Data	153563	CS1H-CPU64H	Buy on EAN