USER'S MANUAL

LG PROGRAMMABLE LOGIC CONTROLLER

GLOFA-GM

Instruction / Programming

LG Industrial Systems

Table of Contents

Ch 1. Overview

Ch 2. The Structure of Software

Ch 3. Common Elements

Ch 4. SFC (Sequential Function Chart)

Ch 5. IL (Instruction List)

Ch 6. LD (Ladder Diagram)

Ch 7. Function and Function Block

Ch 8. Basic Function/Function Block Library

1. Overview

This instruction describes languages that support GM1~GM7 (GLOFA PLC).

GLOFA PLC is based on the standard language of IEC (International Electrotechnical Commission).

1.1 Characteristics of IEC 1131-3 Language

The characteristics of IEC language newly introduced are as follows:

• ••Available to support several data types.

• ••The introduction of program elements such as functions, function blocks etc. enables the bottom-up design and top-down design and the structural creation of PLC program.

• ••The program created by the user shall be stored like as a library system so that it can be used in other environment, which enables to reuse the software.

• ••Available to support various languages so that the user can select the optimal language suitable for the environment to apply.

1.2 Type of Language

The PLC language standardized by IEC consists of two illustrated languages, two character languages and SFC.

• ••Illustrated languages

a) LD (Ladder Diagram): A graphical language that is based on the relay ladder logic

b) FBD (Function Block Diagram): A graphical language for depicting signal and data flows through function blocks - re-usable software elements

• ••Character language

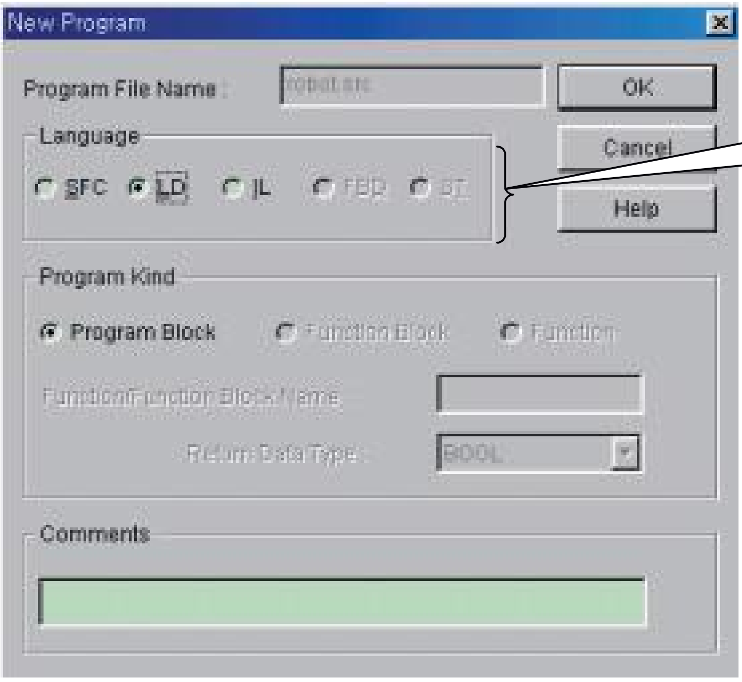
a) IL (Instruction List): A low-level 'assembler like' language that is based on similar instruction list languages.

b) ST (Structured Text): A high-level language of PASCAL type

• ••SFC (Sequential Function Chart): A graphical language for depicting sequential behavior of a control system. It is used for defining control sequences that are time- and event-driven.

• •

The languages supported by GLOFA PLC at present are IL, LD and SFC.



New Program

Program File Name : robot.src

Language: ○ SFC ● LD ○ IL ○ FBD ○ ST

Program Kind: ● Program Block ○ Function Block ○ Function

Function/Function Block Name :

Return Data Type : BOOL

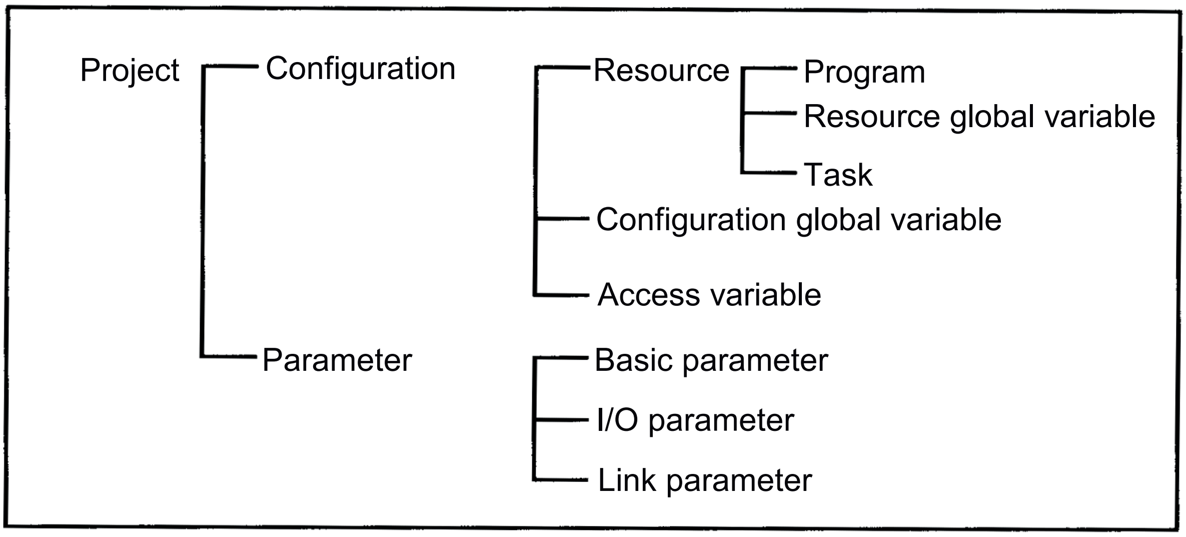
Comments

Choose the language to use

2. The Structure of Software

2.1 Overview

Before making a PLC program, you should have an overall PLC system mapped out in the aspect of software. The overall PLC system is defined as one project in GLOFA PLC. In the project, all composition elements necessary for the PLC system are defined hierarchically.



2.2 Project

- ••For a GLOFA PLC program, the first priority should be given to project configuration. To make one project means that all the elements necessary for a PLC system (scan programs, task definitions, basic parameters, I/O parameters, etc.) are programmed.
- ••A project is divided into two groups: configuration and parameter. Configuration part is for several definitions of a PLC program such as global variable, program, task definition and their interrelation. Parameter part is for setting parameters necessary for a PLC system operation. In this book, we deal with “Configuration part.” For parameter part, please refer to “GMWIN User’s Manual.”

2.3 Configuration

- ••Configuration means a PLC system. It consists of a base, a CPU module, I/O modules and special modules and so on. Generally one PLC system has one CPU module; 4 CPU modules can be installed in GM1.
- ••A PLC system has its own name called Configuration name. This becomes its unique name during communicating between PLCs. Configuration name is limited up to maximum 8 letters in alphabet and for more information, please refer to 3.1.1 Identifiers.
- ••Configuration contains resource, configuration global variables and access variables.

2.3.1 Resource

- Resource means one CPU module. And it is available to define 4 resources in the GM1 Configuration. For GM2 ~ GM5, only one resource is available to define. This resource has its own name that is also used for communication. The resource name is limited up to 8 letters in alphabet and it complies with 3.1.1 Identifiers.
- Resource has programs, resource global variables and task definitions.

2.3.1.1 Program

- It is an application program that is actually executed on PLC. In GLOFA PLC, it is available to create several application programs for one resource and set program conditions to run. For example, you can make programs as follows: program A is a general scan program; program B is a program executed once in a second; program C is an event program that is executed with certain inputs. These conditions to execute the program are called “Task.” Users should make an application program as well as set the conditions (task definitions). Unless task definitions are set, this program will be regarded as a scan program.

Reference

Scan program: application program that repeats a series of execution from the start to the end after reading input data from input modules, and writing the results in output modules.

- A program has its instance name. This instance contains data to be executed in this program.

Reference

For the instance, refer to 3.5.2. Function Block.

2.3.1.2 Resource Global Variable

- The variables defined in resource global variable can be used in any program of the resource. All the data to be shared among programs are defined in resource global variables.
- If users want to use resource global variables in their programs, variables are supposed to be declared as VAR_EXTERNAL.

Reference

For a variable type, refer to 3.3.2 Variable Declaration.

2.3.1.3 Task

- Task means a condition to execute a program. Task definitions contain designation of program execution condition and priority.
- There are 3 types of program execution conditions as follows:
  1) **Single**: executes once if the setting condition is satisfied. The condition is set as a name of **BOOL** variable.
  2) **Interval**: executes periodically per a setting time. The condition is set as elapsed time value. Refer to '3.1.2.3.1 Duration' for how to set the elapsed time value.
  3) **Interrupt**: executes once if the contact of an interrupt card is ON. The condition is set as the contact number of an interrupt card.

Execution conditions	Setting	Description
Single	%IX0.0.1	Executes once if input contact point %IX0.0.1 is ON.
Interval	T#1S	Executes per second
Interrupt	4	Executes once if the contact (#4) of an interrupt card is ON.

- The priority is from 0 to 7. Priority 0 is the highest priority. When scheduling, the task with the highest priority is executed first. And if there are some tasks with the same priority, they're executed in execution-condition-occur order.
- The task used by the reservation in system contains _ERR_SYS, _H_INIT and _INIT task.

_ERR_SYS: System Error (available in GM1, 2)

_H_INIT: Hot Restart

_INIT: Cold/Warm Restart

2.3.2 Configuration Global Variable

- • The variables defined in Configuration Global Variables can be used in any resource program. All the data to be shared among resources are defined in Configuration Global Variable.
- ••If users want to use configuration global variables in their programs, variables are supposed to be declared as VAR_EXTERNAL.

Reference

For a variable type, refer to 3.3.2 Variable Declaration.

- ••Configuration global variable can be defined only in GM1 that can have several resources.

2.3.3 Access Variable

The variable defined in Access Variable can be used in other PLC system.

Reference

For the use of access variable, refer to the User's Manual (Communication part).

3. Common Elements

The elements of **GLOFA PLC** program (programs, functions, function blocks) can be programmed in other languages such as **IL, LD, SFC**, etc., respectively. Those languages, however, have grammar elements in common.

3.1. Expression

3.1.1. Identifiers

- Alphabet and all letters starting with underline (_), and all the mixed letters with numbers and underlines can be identifiers.
- Identifiers are used as variable names.
- Blank (space) is not allowed in identifiers.
- In case of variables, identifiers are generally 16 letters of the alphabet while input/output variable and instance, 8 letters of the alphabet.
- There's no difference between small letters and capitals in alphabet; all the letters of the alphabet are recognized as capitals.

Types	Examples
Capital letters and numbers	IW210, IW215Z, QX75, IDENT
Capital letters, numbers and underline	LIM_SW_2, LIMSW5, ABCD, AB_CD
Capital letters and numbers starting with the underline (_)	_MAIN, _12V7, _ABCD

3.1.2. Data Expression

The data in **GLOFA PLC** is: numbers, a string of characters, time letters, etc.

Types	Examples
Integer	-12, 0, 123_456, +986
Real number	-12.0, 0.0, 0.456, 3.14159_26
Real number with an exponent	-1.34E-12, 1.0E+6, 1.234E6
Binary number	2#1111_1111, 2#1110_0000
Octal number	8#377 (decimal 255) 8#340 (decimal 224)
Hexadecimal number	16#FF (decimal 255) 16#E0 (decimal 224)
BOOL data	0, 1, TRUE, FALSE

3.1.2.1. Numbers

- ••There are integer and real numbers.
- ••Discontinuous underline (_) can be placed between numbers and it doesn't have any meaning.
- ••Decimal complies with general decimal literal expression and if there is a decimal point, this will be real numbers.
- ••In case of expressing exponent, plus/minus signs can be used. The letter 'E' standing for the exponent does not distinguish capitals from small letters.
- ••When using real numbers with exponents, the followings are not allowed.

  ***Ex*)** 12E-5 (•) 12.0E-5 (•)
- ••Integer includes binary, octal, hexadecimal numbers, not to mention decimal, which can be distinguished by placing # in front of each number.
- ••0 ~ 9 and A ~ F are used (including small letters a ~ f) in expressing hexadecimal.
- ••Not available to have plus/minus signs in expressing hexadecimal.
- ••Boolean data may be expressed as an integer 0 or 1.

3.1.2.2. Character String

- ••Character string covers all the letters surrounded with single inverted commas.
- ••The length is limited up to 16 letters in case of character string constant and for an initialization case it does within 30 letters.

Ex)

'CONVEYER'

3.1.2.3. Time Letters

- ••Time letters are classified into these: 1) Duration data which is calculating and controlling the elapsed time of a controlling event; 2) Time of Day and Date data which is displaying the time of the starting/ending point of a controlling event.

3.1.2.3.1. Duration

- ••Duration data starts with the reserved word, 'T#' or 't#'.
- ••Several data types such as date (d), hour (h), minute (m), second (s) and millisecond (ms) should be written in order and duration date can start with any unit among them. Millisecond (ms), the minimum unit can be omitted but don't skip the medium unit between duration units.
- ••Not allowed to use the underline (_).
- ••Duration data can overflow at the maximum unit, if any, and the data with a decimal point is available except 'ms'. It does not exceed T#49d17h2m47s295ms (32bits by 'ms' unit).
- ••The data is limited to the third decimal place in the second unit (s).
- ••Decimal point is not available at 'ms' unit.

- ••Capital and small letters are both available.

Content	Examples
Duration (no underline)	T#14ms, T#14.7s, T#14.7m, T#14.7h t#14.7d, t#25h15m, t#5d14h12m18s356ms

3.1.2.3.2. Time of Day and Date

- ••There are three types expressing ‘Time of Day and Date’ as follows: Date; Time of Day; Date and Time.

Content	Prefix as a reserved word
Date prefix	D#
Time of Day prefix	TOD#
Date and Time prefix	DT#

- ••The starting point of date is January 1st, 1984.
- ••There's a limit on 'Time of Day’ and ‘Date and Time', which is up to the third decimal place in the ‘ms’ unit.
- ••The overflow is not allowed for all the units when expressing ‘Time of Day’ and ‘Date and Time’.

Content	Examples
Date	D#1984-06-25 d#1984-06-25
Time of Day	TOD#15:36:55.36 tod#15:36:55.369
Date and Time	DT#1984-06-25-15:36:55.36 dt#1984-06-25-15:36:55.369

3.2. Data Type

Data has a data type to show its character.

3.2.1. Basic Data Type

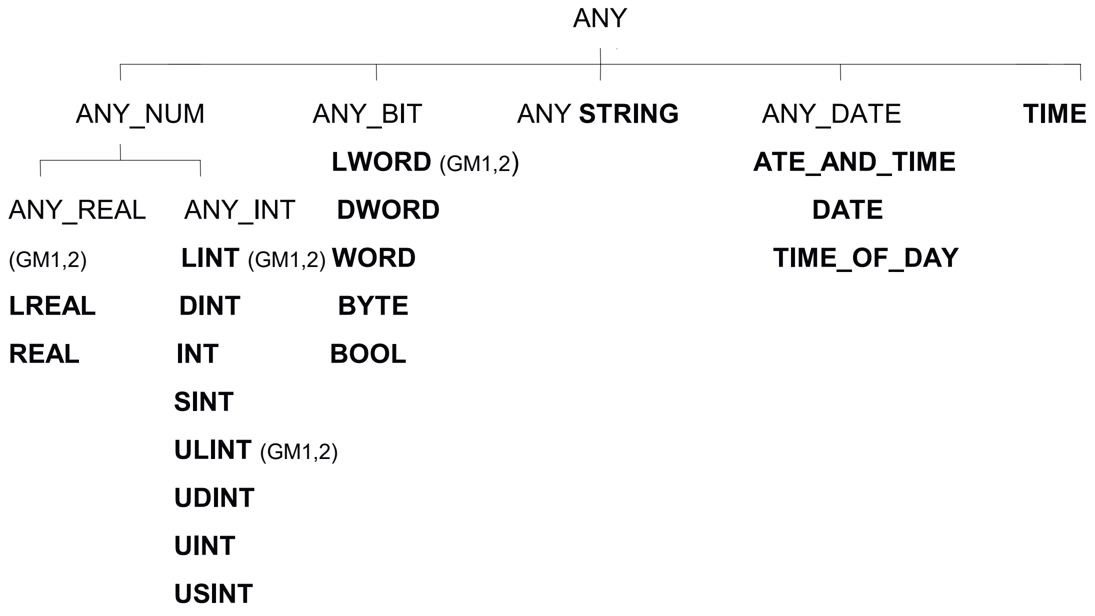
GLOFA PLC supports the following basic data types.

No	Reserved Word	Data Type	Size (bits)	Range
1	SINT	Short Integer	8	-128 ~ 127
2	INT	Integer	16	-32768 ~ 32767
3	DINT	Double Integer	32	-2147483648 ~ 2147483647
4	LINT	Long Integer	64	-2^{63} ~ 2^{63}-1
5	USINT	Unsigned Short Integer	8	0 ~ 255
6	UINT	Unsigned Integer	16	0 ~ 65535
7	UDINT	Unsigned Double Integer	32	0 ~ 4294967295
8	ULINT	Unsigned Long Integer	64	0 ~ 2^{64}-1
9	REAL	Real Numbers	32	-3.402823E38 ~ -1.401298E-45 1.401298E-45 ~ 3.402823E38
10	LREAL	Long Real Numbers	64	-1.7976931E308 ~-4.9406564E-324 4.9406564E-324 ~ 1.7976931E308
11	TIME	Duration	32	T#0S ~ T#49D17H2M47S295MS
12	DATE	Date	16	D#1984-01-01 ~ D#2163-6-6
13	TIME_OF_DAY	Time of Day	32	TOD#00:00:00 ~ TOD#23:59:59.999
14	DATE_AND_TIME	Date and Time	64	DT#1984-01-01-00:00:00 ~ DT#2163-12-31-23:59:59.999
15	STRING	Character String	30*8	Limited within 30 letters.
16	BOOL	Boolean	1	0, 1
17	BYTE	Bit String of Length 8	8	16#0 ~ 16#FF
18	WORD	Bit String of Length 16	16	16#0 ~ 16#FFFF
19	DWORD	Bit String of Length 32	32	16#0 ~ 16#FFFFFFFF
20	LWORD	Bit String of Length 64	64	16#0 ~ 16#FFFFFFFFFFFFFFFF

- • LINT, ULINT, REAL, LREAL, LWORD are available in GM1 and GM2 only.

3.2.2. Data Type Hierarchy Chart

Data types used in **GLOFA PLC** are as follows:



ANY

ANY_NUM ANY_BIT ANY **STRING** ANY_DATE **TIME**

LWORD (GM1,2) **ATE_AND_TIME**

ANY_REAL ANY_INT **DWORD** **DATE**

(GM1,2) **LINT** (GM1,2) **WORD** **TIME_OF_DAY**

LREAL **DINT** **BYTE**

REAL **INT** **BOOL**

SINT

ULINT (GM1,2)

UDINT

UINT

USINT

- ••LINT, ULINT, LWORD and ANY_REAL (LREAL, REAL) are available in **GM1** and **GM2** only.
- ••Data expressed as ANY_NUM includes LREAL, REAL, LINT, DINT, INT, SINT, ULINT, UDINT, UINT, USINT hereafter.
- ••For example, if a data type is expressed as ANY_BIT in GM3, it can use one of the following data types: DWORD, WORD, BYTE and BOOL.

3.2.3. Initial Value

If an initial value of data were not assigned, it would be automatically assigned as below.

Data Type	Initial Value
SINT, INT, DINT, LINT	0
USINT, UINT, UDINT, ULINT	0
BOOL, BYTE, WORD, DWORD, LWORD	0
REAL, LREAL	0.0
TIME	T#0s
DATE	D#1984-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#1984-01-01-00:00:00
STRING	' ' (empty string)

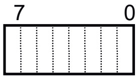
3.2.4. Data Type Structure

# Bit String

BOOL

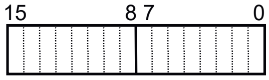
1 bit, range: 0, 1

BYTE



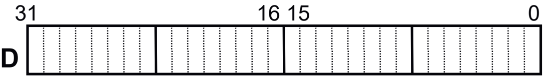
8 bits, range: 2#0000_0000 ~ 2#1111_1111, 16#00 ~ 16#FF

WORD



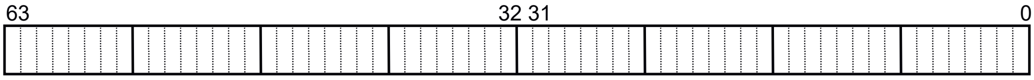
16 bits, range: 2#0000_0000 _0000_0000 ~ 2#1111_1111_1111_1111
16#0000 ~ 16#FFFF

DWORD



32 bits, range: 2#0000_...000 ~ 2#1111_...111
16#00000000 ~ 16#FFFFFFFF

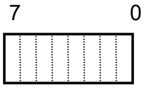
LWORD



64 bits, range: 2#0000_...000 ~ 2#1111_...111, 16#0000000000000000 ~ 16#FFFFFFFFFFFFFFFF

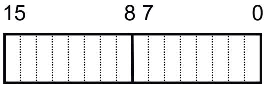
# Unsigned Integer

USINT



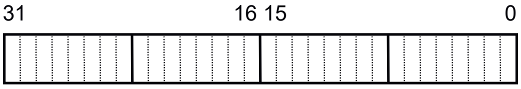
8 bits, range: 0 ~ 255

UINT



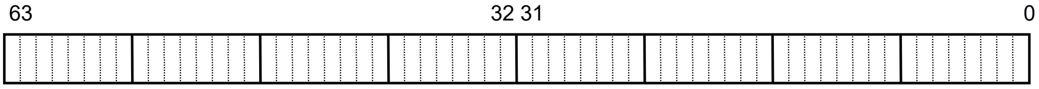
16 bits, range: 0 ~ 65,535

UDINT



32 bits, range: 0 ~ 4,294,967,295

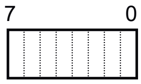
ULINT



64 bits, range: 0 ~ 2^{64}-1

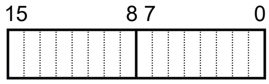
# Integer (Negative number is expressed as **2's Complement.**)

SINT



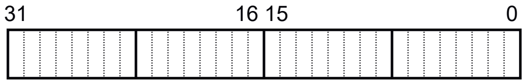
8 bits, range: -128 ~ 127

INT



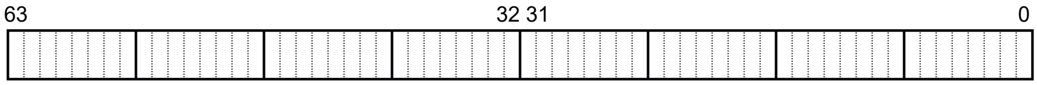
16 bits, range: -32,768 ~ 32,767

DINT



32 bits, range: -2,147,483,648 ~ 2,147,483,647

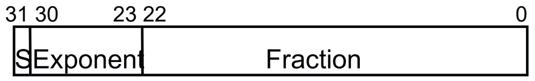
LINT



64 bits, range: -2^{63} ~ 2^{63}-1

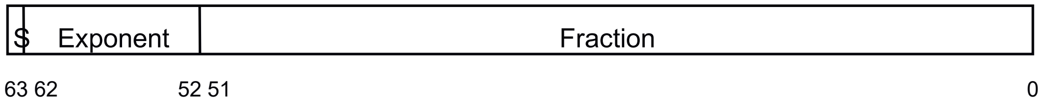
# Real (based on the **IEEE Standard 754-1984**)

REAL



32 bits, range: ±1.401298E-45 ~ ±3.402823E38

LREAL

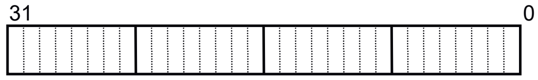


64 bits, range: ±4.9406564E-324 ~ ±1.7976931E308

- **S**: sign (If it's 0, the data is a positive number; otherwise, a negative number).
- **Exponent**: exponent of 2 (2^{e-127}: for REAL, $e=b_{30}b_{29}...b_{23}$; for LREAL, $e=b_{62}b_{61}...b_{52}$).
- **Fraction**: a decimal fraction (Fraction: for REAL, $f=b_{22}b_{21}...b_{0}$; for LREAL, $e=b_{51}b_{52}...b_{0}$).

# Time

TIME

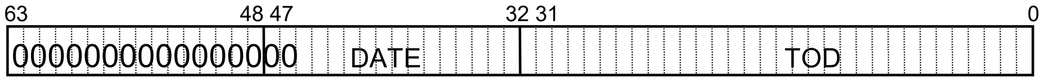


32 bits, range: 0 ~ 4,294,967,295ms

T#49d17h2m47s295ms

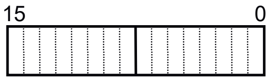
# Date

DT



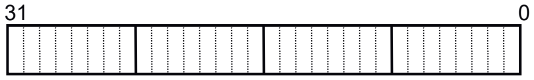
64bits, range: DT#1984-01-01-00:00:00 ~ DT#2163-12-31-23:59:59.999

DATE



16bits, range: D#1984-01-01 ~ D#2163-6-6

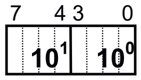
TOD



32bits, range: TOD#00:00:00 ~ TOD#23:59:59.999

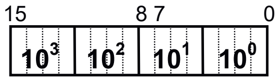
#BCD

(BYTE)



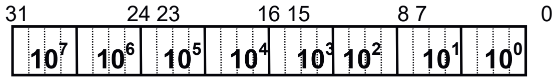
8bits, range: 0 ~ 99

(WORD)



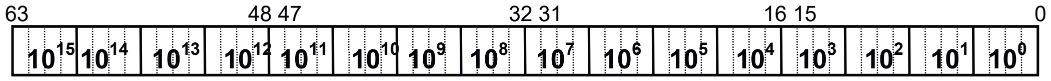
16bits, range: 0 ~ 9999

(DWORD)



32bits, range: 0 ~ 99,999,999

(LWORD)



64bits, range: 0 ~ 9,999,999,999,999,999

3.3. **Variable**

A variable, data used in the program, has its own value. ‘Variable’ means something that can vary such as an input/output of **PLC**, memory, etc.

3.3.1. Variable Expression

- ••Variables can be expressed in two ways: one is to give a name to a data element using an identifier (Variable by Identifier) and the other is to directly assign a memory address or an input/output of PLC to a data element (Direct Variable).
- ••A variable by identifier should be unique within its ‘effective scope’ (program area where the variable was declared) in order to distinguish it from other variables.
- ••A direct variable is expressed as one, which starts with the percent sign (%) followed by the ‘location prefix’, a prefix of the data size, and more than one unsigned integer numbers divided by a period (.). The prefix are shown as below:

Location prefix

No.	Prefix	Meaning
1	• •	Input Location
2	Q	Output Location
3	M	Memory Location

Size prefix

No.	Prefix	Meaning
1	X	1 bit size
2	None	1 bit size
3	B	1 BYTE (8 bits) size
4	W	1 WORD (16 bits) size
5	D	1 DOUBLE WORD (32 bits) size
6	L	1 LONG WORD (64 bits) size

Expression format

%[Location Prefix][Size Prefix] n1.n2.n3

No.	**• ••Q**	**M**
n1	Base number (starting from “0”)	n1 data according to [size prefix] (starting from “0”)
n2	Slot number (starting from “0”)	n2 bit of n1 data (starting from “0”): available to omit
n3	n3 data according to the [size prefix] (starting from “0”)	Not used.

Examples

%QX3.1.4 or **%Q3.1.4**	4th output of no.1 slot on no.3 base (1bit)
%• W2.4.1	1st word input of no.4 slot on no.2 base (16bits)
%MD48	48th double word memory
%MW40.3	3rd bit of 40th word memory
	(Internal memory doesn't have a base or slot number.)

- Small letter is not allowed as a prefix.
- A variable without a size prefix is treated as 1 bit.
- Direct variables are available to use without a variable declaration.

3.3.2. Variable Declaration

- Program elements (programs, functions, function blocks, etc) have declaration parts to edit their variables to use.
- Users should declare variables first to use them in the program elements.
- The contents of a variable declaration are as follows:

1) Variable types: how to declare variables?

Variable types	Description
VAR	General variable available to read/write
VAR_RETAIN	Retaining (data-keeping) variable
VAR_CONSTANT	Read Only Variable
VAR_EXTERNAL	Declaration to use the variable declared as **VAR_GLOBAL**

Reference

When declaring **Resource Global Variable** and **Configuration Global Variable**, variable formats are **VAR_GLOBAL, VAR_GLOBAL_RETAIN**, and **VAR_GLOBAL_CONSTANT; VAR_EXTERNAL** is not available for them.

2) Data type: sets a variable data type.

3) Memory allocation: assigns memory for a variable.

Auto: the compiler sets a variable location automatically (Automatic Allocation Variable).

Assign (AT): a user sets a variable location, using a direct variable (Direct Variable).

Reference

The location of Automatic Allocation Variable is not fixed. If variable **VAL1**, for example, was declared as **BOOL**, it is not fixed in the internal memory; the compiler and linker fix its location. If the program is compiled again after modification, the location may change.

The merit of Automatic Allocation Variable is that users don't have to care the location of the internal variables because its location is not overlapped as long as a variable name is different from others.

It is recommended not to use Direct Variable except **%•** and **%Q** because the location of a variable is fixed and it could be overlapped in a wrong-used case.

- ••Initial Value Assignment: assigns an initial value. A variable is set with an initial value as is shown in '3.2.3. Initial Value' if not assigned.

Reference

The initial value is not assigned when it comes to **VAR_EXTERNAL**.
In case of 'Variable Declaration', you cannot assign an initial value to **%•** or **%Q** variables.

- ••You can declare variable VAR_RETAIN that keeps its data in case of power failure. Rules are:
  1) 'Retention Variable' retains its data when the system is set as 'Warm Restart'.
  2) In case of 'Cold Restart', variables are initialized as the initial values set by users or the basic initial values as are shown in '3.2.3 Initial Value'.
- ••Variables, which are not declared as VAR_RETAIN, are to be initialized as the initial values set by a user or the basic initial values in case of **Warm** or **Cold Restart**'.

Reference

Variables, which are assigned as %I or %Q, are not to be declared as **VAR_RETAIN** or **VAR_CONSTANT**.

- ••Users can declare variables 'Array' with Elementary Data Type. When declaring the Array Variable, users are supposed to set Data Type and Array Size; 'String' among Elementary Data Type is not allowed.
- ••Effective scope of variable declaration, the area which is available to use the variable, is limited to the program where variables are declared. And users can't use variables declared in other program in the above area. On the contrary, users can get an access to 'Global Variable' from other program elements by declaring it as 'VAR_EXTERNAL': 'Configuration Global Variable' can be used in all program elements of all resources; 'Resource Global Variable' can be used in all program elements of the very resource.••

Examples of Variable Declaration

Variable Name	Variable Kind	Data Type	Initial Value	Memory Allocation
I_VAL	**VAR**	INT	**1234**	Auto
BIPOLAR	**VAR_RETAIN**	REAL		Auto
LIMIT_SW	**VAR**	BOOL		%IX1.0.2
GLO_SW	**VAR_EXTERNAL**	DWORD		Auto
READ_BUF	**VAR**	ARRAY OF INT[10]		Auto

3.3.3. Reserved Variable

- ‘Reserved Variable’ is the variables previously declared in the system. These variables are used for special purposes and users cannot declare other variables with the Reserved Variable names.
- Users can use these reserved variables without variable declaration.
- For further information, please refer to ‘User’s Manual’.

1) User Flag

Reserved Variable	Data Type	Description
_ERR	BOOL	Operation error contact
_LER	BOOL	Operation error latch contact
_T20MS	BOOL	20ms clock contact
_T100MS	BOOL	100ms clock contact
_T200MS	BOOL	200ms clock contact
_T1S	BOOL	1 sec. clock contact
_T2S	BOOL	2 sec. clock contact
_T10S	BOOL	10 sec. clock contact
_T20S	BOOL	20 sec. clock contact
_T60S	BOOL	60 sec. clock contact
_ON	BOOL	All time ON contact
_OFF	BOOL	All time OFF contact
_1ON	BOOL	1 scan ON contact
_1OFF	BOOL	1 scan OFF contact
_STOG	BOOL	Reversal at every scanning
_INIT_DONE	BOOL	Initial program completion
_RTC_DATE	DATE	Current date of RTC
_RTC_TOD	TOD	Current time of RTC
_RTC_WEEK	UINT	Current day of RTC

2) System Error Flag

Reserved Variable	Data Type	Description
_CNF_ER	WORD	System error (Heavy trouble)
_CPU_ER	BOOL	CPU configuration error
_IO_TYER	BOOL	Module type inconsistency error
_IO_DEER	BOOL	Module installation error
_FUSE_ER	BOOL	Fuse shortage error
_IO_RWER	BOOL	I/O module read/write error (trouble)
_SP_IFER	BOOL	Special/communication module interface error (trouble)
_ANNUN_ER	BOOL	Heavy trouble detection error of external device
_WD_ER	BOOL	Scan Watch-Dog error
_CODE_ER	BOOL	Program code error
_STACK_ER	BOOL	Stack Overflow error
_P_BCK_ER	BOOL	Program error

3) System Error Release Flag

Reserved Variable	Data Type	Description
_CNF_ER_M	BYTE	System error (heavy trouble) release

4) System Alarm Flag

Reserved variable	Data type	Description
_CNF_WAR	WORD	System Alarm (Alarm message)
_RTC_ERR	BOOL	RTC data error
_D_BCK_ER	BOOL	Data backup error
_H_BCK_ER	BOOL	Hot restart unable error
_AB_SD_ER	BOOL	Abnormal Shutdown
_TASK_ERR	BOOL	Task conflict (normal cycle, external task)
_BAT_ERR	BOOL	Battery error
_ANNUN_WR	BOOL	Light trouble detection of external device
_HSPMT1_ER	BOOL	Over high-speed link parameter 1
_HSPMT2_ER	BOOL	Over high-speed link parameter 2
_HSPMT3_ER	BOOL	Over high-speed link parameter 3
_HSPMT4_ER	BOOL	Over high-speed link parameter 4

5) Detailed System Error Flag

Reserved variable	Data type	Description
_IO_TYER_N	UINT	Module type inconsistency slot number
_IO_TYERR	ARRAY OF BYTE	Module type inconsistency location
_IO_DEER_N	UINT	Module installation slot number
_IO_DEERR	ARRAY OF BYTE	Module installation location
_FUSE_ER_N	UINT	Fuse shortage slot number
_FUSE_ERR	ARRAY OF BYTE	Fuse shortage slot location
_IO_RWER_N	UINT	I/O module read/write error slot number
_IO_RWERR	ARRAY OF BYTE	I/O module read/write error slot location
_ANC_ERR	ARRAY OF UINT	Heavy trouble detection of external device
_ANC_WAR	ARRAY OF UINT	Light trouble detection of external device
_ANC_WB	ARRAY OF BOOL	Alarm message detection bit map of external device
_TC_BMAP	ARRAY OF BOOL	Task conflict mark
_TC_CNT	ARRAY OF UINT	Task conflict counter
_BAT_ER_TM	DT	Battery voltage drop-down time
_AC_F_CNT	UINT	Shutdown counter
_AC_F_TM	ARRAY OF DT	Instantaneous service interruption history

6) Information of System Operation Status

Reserved variable•	Data type••	••••••••••••••••••••••••
•••••••••••••••	••••••••••	System Type
•••••••••	•••••••••••••••••••	PLC O/S Version number
••••••••••	•••••	••
••••••••••••	••••••	•••
••••••••	•••••	••
••••••••••	••••	•••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	••••••••••••••	••
•••••••••	•••••	•••••••••••••••••••••••••••••••••••••

7) Communication Module Information Flag **[n** is a slot number where a communication module is installed **(n = 0 ~ 7)]**

•••••••••••••••••	••••••••••••••••••••	••••••••••••••••••••••••••
•••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	•••
•••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
••••••••••	•••••	••
•••••••	•••••	••
•••••••••	•••••	••
•••••••••	•••••	••
••••••••	•••••	••
••••••••	•••••	••

8) Remote I/O Control Flag **[m** is a slot number where a communication module is installed **(m = 0 ~ 7)]**

Reserved variable	Data type	Description
_FSMm_RESET	**BOOL** (able to write)	Remote • /O station reset control (reset = 1)
_FSMm_IO_RESET	**BOOL**(able to write)	Output reset control of remote I/O station (reset = 1)
_FSMm_st_no	**USINT** (able to **write**)	Station number of corresponding remote I/O station

9) Detailed High-speed Link Information Flag **[m** is a high-speed link parameter number **(m = 1, 2, 3, 4)]**

Reserved variable	Data type	Description
_HSmRLINK	**BOOL**	HS RUN_LINK information
_HSmLTRBL	**BOOL**	Abnormal information of HS (Link Trouble)
_HSmSTATE	**ARRAY OF BOOL**	General communication status information of k data block
_HSmMOD	**ARRAY OF BOOL**	Station mode information of k data block at HS link parameter (Run = 1, Other = 0)
_HSmTRX	**ARRAY OF BOOL**	Communication status information of k data block at HS link parameter (Normal = 1, Abnormal = 0)
_HSmERR	**ARRAY OF BOOL**	Station status information of k data block at HS link parameter (Normal = 0, Error = 1)

3.4. Reserved Word

Reserved words are previously defined words to use in the system. And these reserved words cannot be used as an identifier.

Reserved words
ACTION ... END_ACTION
ARRAY ... OF
AT
CASE ... OF ... ELSE ... END_CASE **CONFIGURATION ... END_CONFIGURATION**
Name of data type
DATE#, D# **DATE_AND_TIME#, DT#**
EXIT
FOR ... TO ... BY ... DO ... END_FOR
FUNCTION ... END_FUNCTION
FUNCTION_BLOCK ... END_FUNCTION_BLOCK
Name of function block
IF ... THEN ... ELSIF ... ELSE ... END_IF
OK
Operator **(IL** language**)** Operator **(ST** language**)**
PROGRAM
PROGRAM ... END_PROGRAM
REPEAT ... UNTIL ... END_REPEAT
RESOURCE ... END_RESOURCE
RETAIN
RETURN
STEP ... END_STEP
STRUCTURE ... END_STRUCTURE
T#
TASK ... WITH
TIME_OF_DAY#, TOD#
TRANSITION ... FROM... TO ... END_TRANSITION
TYPE ... END_TYPE
VAR ... END_VAR **VAR_INPUT ... END_VAR** **VAR_OUTPUT ... END_VAR** **VAR_IN_OUT ... END_VAR** **VAR_EXTERNAL ... END_VAR**
VAR_ACCESS ... END_VAR
VAR_GLOBAL ... END_VAR
WHILE ... DO ... END_WHILE
WITH

3.5. Program Type

- There are three types of program: function, function block and program.
- It is not available to call its own program in the program (reflexive call is prohibited).

3.5.1. Function

- A function has one output.

Example

If there is function A that is to add input IN1 and IN2 and then add 100 to the sum of IN1 and IN2. and the output 1 <= IN1 + IN2 + 100, this function will be correct. However, if the above function has one more output (output 2 <= IN1 + IN2 * 100), this will not be a function because it has 2 outputs: output 1 and output 2.

- A function does not have data to preserve its state inside. This means if an input is constant, an output value should be constant, which is a function.

Example

If there is function B whose contents are

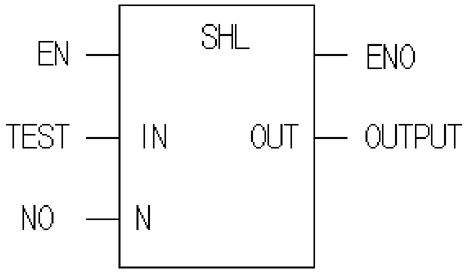
Output 1 <= IN1 + IN2 + Val

Val <= output1 (where, Val is an internal variable),

This cannot be a function as there is internal variable Val. To have an internal variable means that an output will be different even if there is a same input. Output 1 value is subject to change because of Val variable even if the value of IN1 and IN2 are constant as is shown on the above. Compared with the above function A, function A will have output 1 value (150) when IN1 is 20 and IN2 is 30. This shows that the output value will be constant if inputs are constant.

- An internal variable of a function is not available to have an initial value.
- Users can't declare a function as VAR_EXTERNAL and use it.
- It is not available to use direct variables inside the function.
- A function will be called by program elements and used.
- Data transfer from program composition elements, which call the function, to the function will be executed through an input of a function.

Example



SHL function is a basic function that shifts input IN to the left as many as N bit number and produces it as an output. Program composition elements call SHL function, assigning a value of TEST variable to input IN and a value of NO variable to input N. The result will be stored in OUTPUT variable.

- • •A function is inserted into a library for use.
- • •It is not available to call a function block or a program inside the function.
- • •A function has a variable whose name is the same as that of the function and whose data type is the same as the data type of the result of the function. This variable is automatically created when making a function, and the result value of the function will be written in the output.

Example

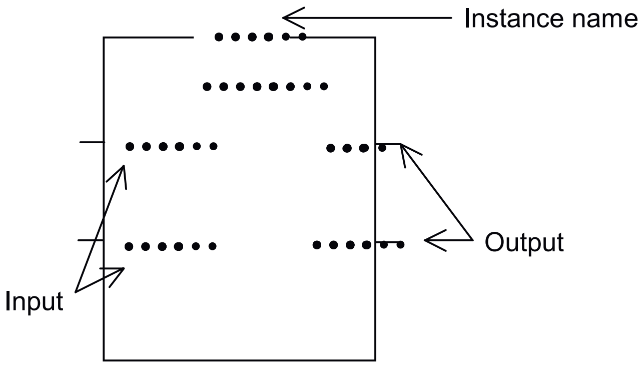
If a function name is WEIGH and a data type of a result value is WORD, a variable whose name is WEIGH and whose data type is WORD will be automatically created inside the function. Users can store the result of function in variable WEIGH.

```
ST          WEIGH              (example in IL)
```

3.5.2 Function Block

- • •A function block has several outputs.
- • •A function block has data inside. A function block should declare the instance as it declares variables before using them. Instance is a set of variables used in a function block. A function block should have its data memory to preserve the output value as well as variables used inside, which is called as “instance.” A program is a kind of a function block and also needs to declare “instance.” However, users cannot call a program inside a program or a function block for use, contrary to a function block.
- • •In order to use the output value of a function block, it is required to place a period (.) between the name of instance and the output name.

Example



General examples of a function block are Timer and Counter. On-delay timer function block is TON and this is executed if IN is ON after users declare T1 as “instance.” In order to use timer output contact and duration value, it is required to place a period (.) between the name of instance and the output name. In case of a timer function block, the output contact and the elapsed time value for the instance are T1.Q and T1.ET respectively because the output contact name is Q and the elapsed time contact name is ET. The output value of a function is a return value by calling a function while the output value of a function block is fixed for the instance.

- ••Users cannot declare a direct variable inside a function block. However, users can use a direct variable declared as Global Variable and allocated according to ‘Assign (AT)’ after declaring it as VAR_EXTERNAL.
- ••A function block is inserted into a library for use.
- ••It is not available to call a program inside the function block.

3.5.3 Program

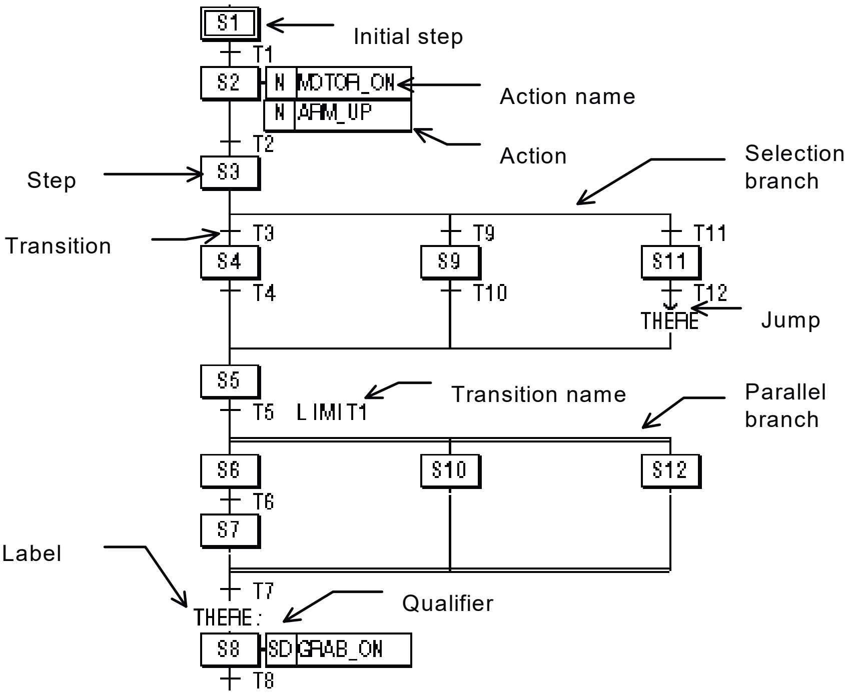
- ••Users can use a program after declaring an instance like a function block.
- ••It is available to use direct variables in the program.
- ••A program does not have input/output variables.

The calling of a program is defined in the resource.

4. SFC (Sequential Function Chart)

4.1. Overview

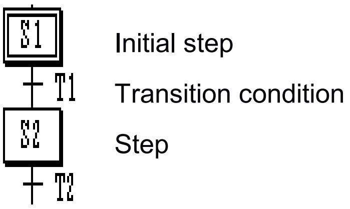
- SFC is a structured language that extends an application program in the form of flow chart according to the processing sequence, using a PLC language.
- SFC splits an application program into step and transition, and provides how to connect them each other. Each step is related to action and each transition is related to transition condition.
- As SFC should contain the state information, only program and function block among program types are available to apply this SFC.
- Type



4.2. SFC Structure

4.2.1. Step

- Step indicates a sequence control unit by connecting the action.
- When step is in an active state, the attached content of action will be executed.
- The initial step is one to be activated first.

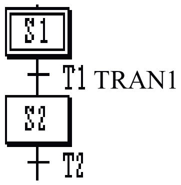


- If a next transition condition of activated initial step (S1) is established, step 1 (S1) that is currently activated becomes deactivated and Step 2 (S2) connected to S1 becomes activated.

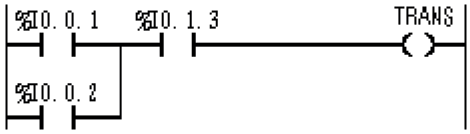
4.2.2. Transition

- ••Transition indicates the execution condition between steps.
- ••A transition condition should be described as a PLC language such as IL or LD.

  The result of a transition condition should always be a BOOL type and the variable name should be TRANS for any transition.
- ••In case that the result of transition condition is 1, the current step is deactivated and the next step is activated.
- ••There must be a transition between step and step.



The content of TRAN1



When TRANS is on, S1 will be deactivated and S2 activated.

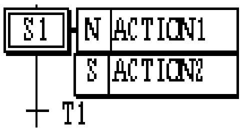
TRANS is the internally declared variable.

A transition condition of all transition should be output in TRANS variable.

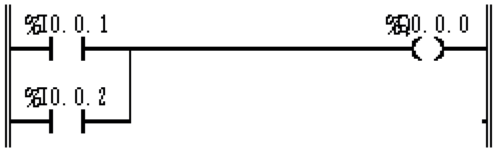
4.2.3. Action

- ••Each step is able to connect up to two actions.
- ••The step without action is regarded as a waiting action and it is required to wait until the next transition condition will be 1.
- ••Action is composed of PLC language such as IL or LD and the content of action will be executed while the step is activated.
- ••Action qualifier will be used to control action.
- ••When action becomes deactivated state after activating, the contact output in action will be 0.

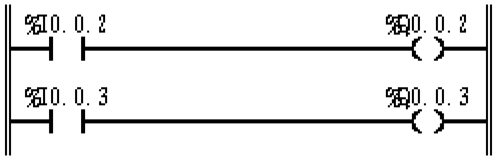
  However, S, R, function and function block output retain their state before they become non-activating.



The content of **ACTION1**



The content of **ACTION2**



- **ACTION1** will be executed only when **S1** is activated.
- **ACTION2** will be executed until **S1** meets **R** qualifier after activated.
  It goes on executing even if **S1** is deactivated.
- When action is deactivated, this action is Post Scanned and then passes to the next step.

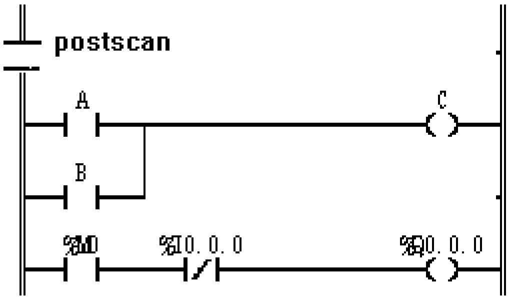
Reference

Post Scan

When action is deactivated, this action is scanned again.

As it is scanned as if there were a contact (contact with the value of 0) in the beginning part of an action program, the program output, which is composed of contacts, will be 0.

Function, function block, **S, R** output etc., are not included.



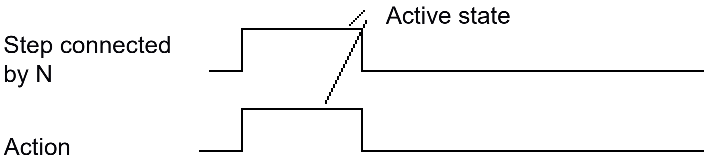
In this figure, as the contact of **postscan** is 0, C and %Q0.0.0 will be 0.

4.2.4. Action Qualifier

- ••Whenever action is used, action qualifier will be followed.
- ••The action of step defines an executing point and time according to the assigned qualifier.
- ••Types of action qualifier are as follows:

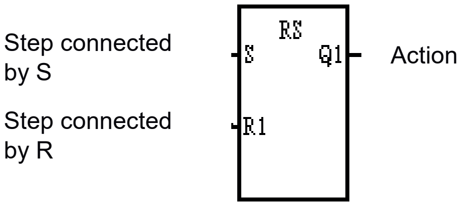
1) N (Non-Stored)

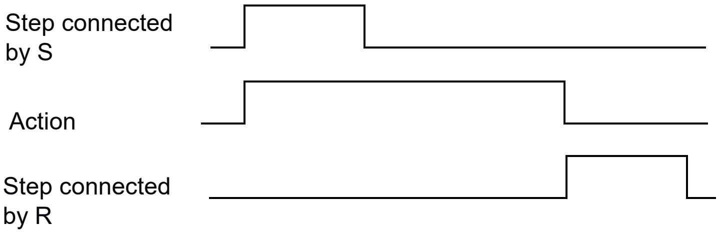
Action is executed only when the step is activated.



2) S (Set)

It continues the action after the step is deactivated (until the action is reset by R qualifier).



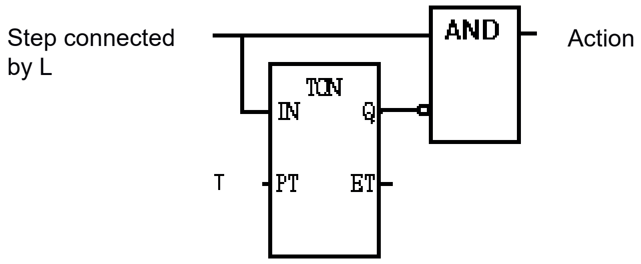


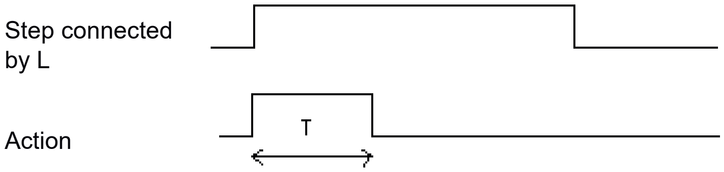
3) R (Overriding Reset)

It terminates the execution of an action previously started with the S, SD, SL or DS qualifier.

4) L (Time Limited)

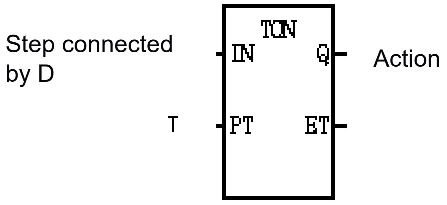
It start the action when the step becomes active and continue until the step goes inactive or a set time elapses.

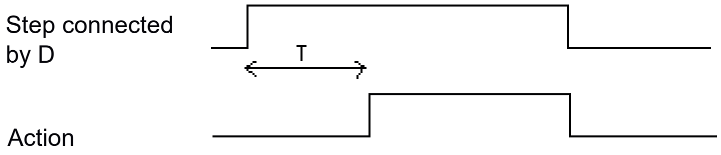




5) D (Time Delayed)

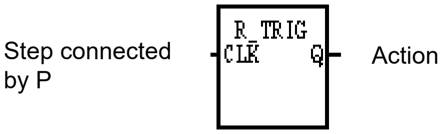
Start a delay timer when the step becomes active - after the time delay the action starts (if step still active) and continues until deactivated.

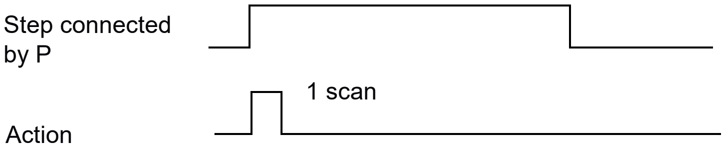




6) P (Pulse)

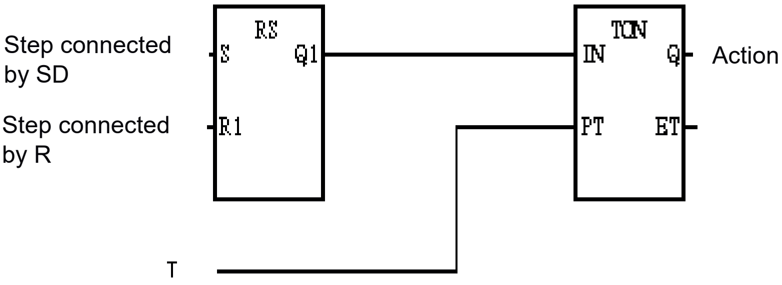
It starts the action when the step becomes active and executes the action only once.

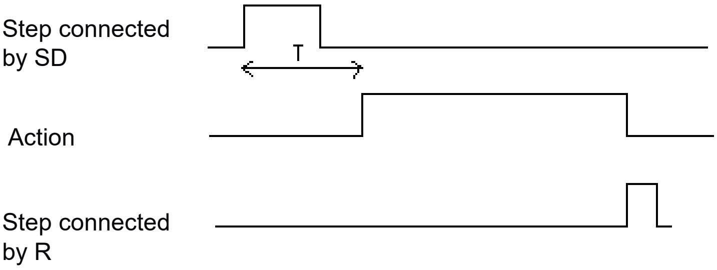




7) SD (Stored & Time Delayed)

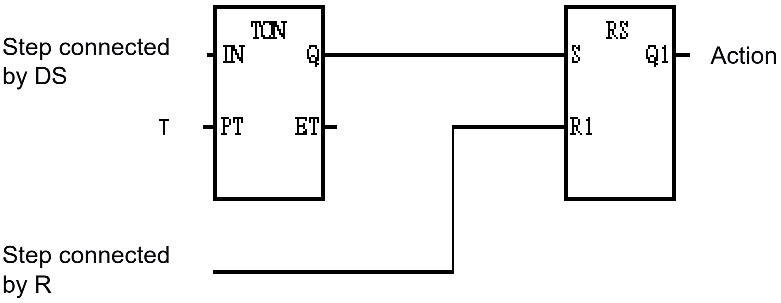
It starts a delay timer when the step becomes active - after the time delay, the action starts and continues until reset (regardless of step activation/deactivation).

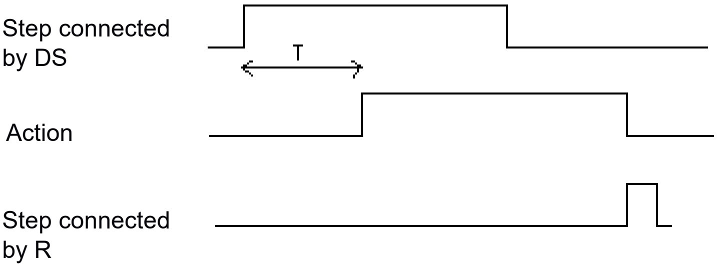




8) DS (Delayed & Stored)

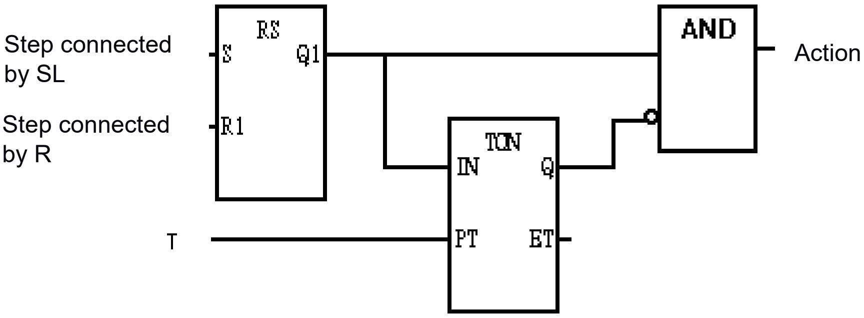
It starts a delay timer when the step becomes active - after the time delay the action starts (if step still active) and continues until reset by R qualifier.

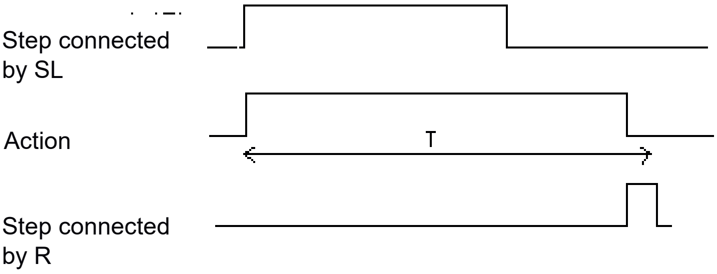




9) SL (Stored & Timed Limited)

It starts the action when the step becomes active and continues for a set time or until the action is reset (regardless of step activation/deactivation).

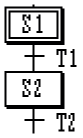


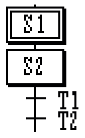


4.3. Extension Regulation

4.3.1. Serial Connection

- ••2 steps are always divided by transitions without connecting directly.
- ••Step always divides 2 transitions without connecting directly.





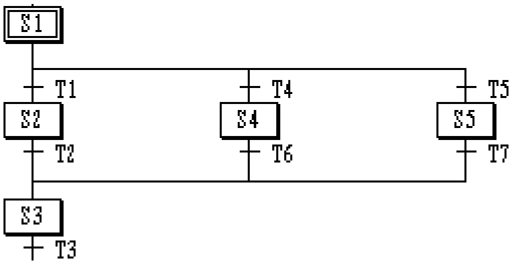
[correct example] [wrong example]

- ••For the transition between steps connected by serial, the lower step will be activated if the upper step is active and the transition condition connected to the next is 1.

4.3.2. Selection Branch

- ••When a processor executes a selection branch, the processor finds the first path with a true transition in the order of the program scan and executes the steps and transitions in that path. If more than one path in a selection branch goes true at the same time, the processor chooses the left-most path. The following example shows a typical scan sequence.

Example



* In case that the transition condition of **T1** is **1**,
the order of activation will be **S1 -> S2 -> S3**.

* In case that the transition condition of **T4** is **1**,
the order of activation will be **S1 -> S4 -> S3**.

* In case that the transition condition of **T5** is **1**,
the order of activation will be **S1 -> S5 -> S3**.

If the transition conditions are **1** at the same time, the processor chooses the left-most path.

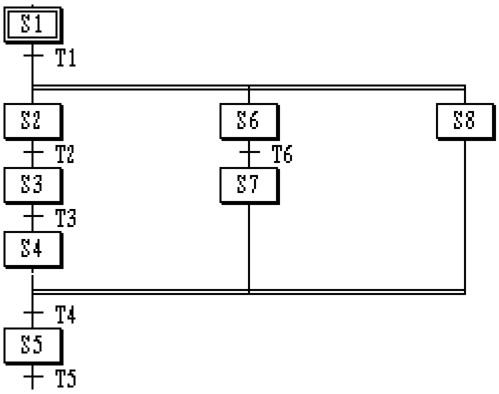
* In case that the transition condition of **T1** and **T4** is **1** at the same time,
the order of activation will be **S1 -> S2 -> S3**.

* In case that the transition condition of **T4** and **T5** is **1** at the same time,
the order of activation will be **S1 -> S4 -> S3**.

4.3.3. Parallel Branch (Simultaneous Branch)

- ••When a processor executes the parallel (simultaneous) branch, the processor scans the branch from left-to-right, top-to-bottom. It appears that the processor executes each path in the branch simultaneously.
- ••In case of connecting by parallel branch, if the transition condition connected to the next is 1, all steps tied to this transition will be activated. The extension of each branch will be the same as serial connection. At this time, the steps in the state of activation are as many as the number of branches.
- ••In case of combining in parallel branch, if the transition condition is 1 when the state of all the last steps of each branch is activated, the step connected to the next will be activated.

Example



- If the transition condition of **T1** is **1** when **S1** is active, **S2, S6** and **S8** will be activated and **S1** will be deactivated.

- If the transition condition of **T4** is **1** when **S4, S7** and **S8** are activated, **S5** will be activated and **S4, S**7 and **S8** will be deactivated.

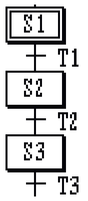
* The order of **activation**

```
S1-+->S2-->S3-->S4--+->S5
   +->S6-->S7---------+
   +->S8----------------+
```

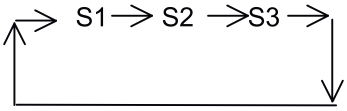
4.3.4. Jump

- ••If the transition condition connected to the next is 1 after the last step of SFC is activated, the initial step of SFC will be activated.

Example



* The order of activation

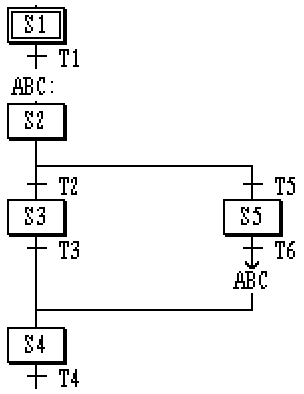


- ••It is possible to extend to the place using a jump.
- ••Jump can only be place at the end of SFC program or the end of a selection branch.

  It is not allowed to jump into the inside or outside of parallel branch; it is allowed to jump within parallel branch.

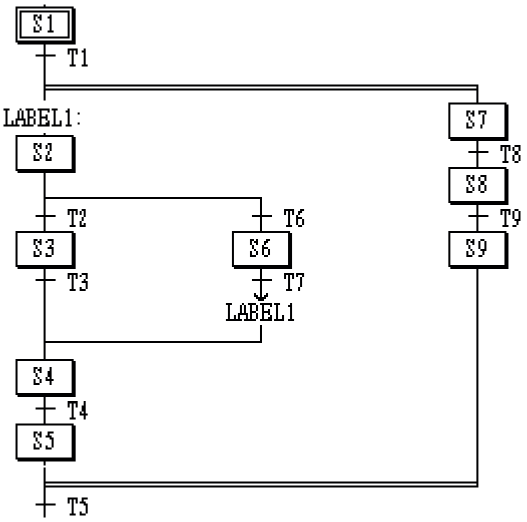
Example

1) Jump at the end of selection branch

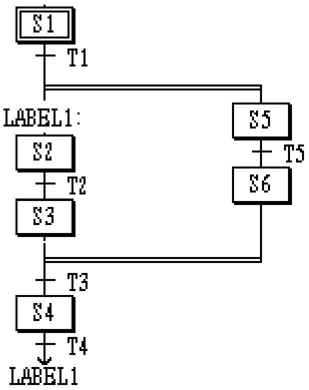


- **S2** will be activated after **S5**.

2) Jump within parallel branch



3) Not available to jump into the inside of parallel branch..

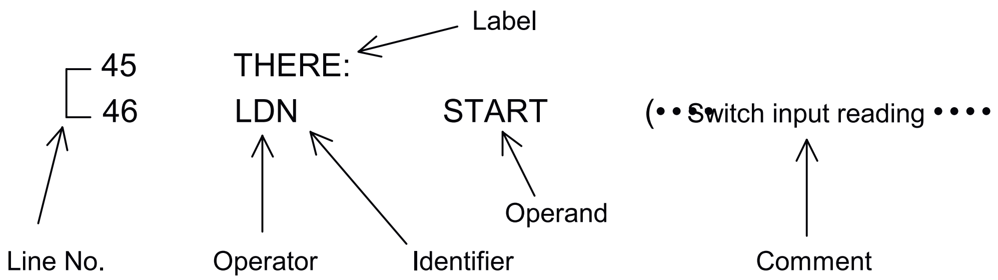


MEMO

5. IL (Instruction List)

5.1. Overview

- IL is a low-level 'assembler like' language.
- IL is applicable to simple PLC systems.
- Type



```
45    THERE:
46    LDN        START      (••Switch input reading ••••
```

5.2. Current Result: CR

- In IL, there is a register that stores an operation result by that time, which is called “CR (current result)”.
- Only one CR exists in IL.
- CR is able to be any data type.
- The operator that puts a certain value to CR and determines its data type is LD (Load).

Example

LD %IX0.0.0 is to put the value of %IX0.0.0 to the CR. Now, the data type of CR is BOOL because the data type expressed as X is BOOL. If variable VAL is declared as INT and is written as LD VAL, it writes the value of VAL to CR and the data type of CR is INT.

- ST operator stores the current result (CR) in a variable.

Example

If variable VAL is declared as INT and is written as ST VAL, this means that CR is stored in variable VAL. At this time, the data type of CR should be INT. Unless CR is an INT type, an error occurs when compiling.

Please read the following:

```
LD          %IX0.0.0
ST          VAL     (assume that variable VAL is declared as INT)
```

CR is assigned as BOOL in the first row and declared as INT in the second row, which results in an error when compiling.

```
LD          %IX0.0.0
ST          START
LD          20
ST          VAL     (assume that variable START is declared as BOOL and variable VAL as INT)
```

The above example is executed normally because the data type to store CR respectively is the same.

5.3. Instructions

- IL is a list of instructions.
- Each instruction must begin on a new line, and must contain an operator, completed with optional modifiers and, if necessary, for the specific operation, one or more operands, separated with commas (',').

5.3.1. Label

- A label followed by a colon (':') may precede the instruction.
- Labels are used as operands for some operations such as jumps.

5.3.2. Modifier

- The modifier character must complete the name of the operator, with no blank characters between them. There're three types of modifiers: N, (, C.
- The N modifier indicates a Boolean negation of the operand.

Example

ANDN %IX2.0.0 is interpreted as:

CR <= CR AND NOT %IX2.0.0

When N is attached to JMP, CAL and RET with no blank character between them, this means it executes the instruction when CR is BOOL 0.

- Modifier '(' delays the operation of an operator until it meets operator ')'.

As there is only one CR in IL, it is available to execute the delayed operation: CR is kept while other operations are executed and after that, operation will be done with the stored CR value.

Type	Characteristic	Semantics
(	Modifier	Operation is delayed.
)	Operator	Evaluation deferred operation used with '('

Example

AND(%IX1.0.0

OR   %IX2.0.0)

CR <= CR AND (%IX1.0.0 OR %IX2.0.0)

This means that the execution of AND will be delayed until ')' appears. After the operation inside the parentheses, %IX1.0.0 OR %IX2.0.0, is executed, the operation with the result will be done.

- Modifier 'C' indicates that the attached instruction must be executed only if the current result has the Boolean value 1 (TRUE).

Example

JMPC   THERE

If CR is BOOL 1, jump to THERE.

5.3.3. Basic Operator

- Basic operators are as follows:

No.	Operator	Modifier	Operand	Semantics
1	LD	N	Data	Set current results equal to operand
2	ST	N	Data	Store current results to operand
3	S R		BOOL BOOL	If CR is BOOL 1, set Boolean Operand to 1 If CR is BOOL 1, set Boolean Operand to 0
4	AND	N,(	Data	Boolean AND operation
5	OR	N,(	Data	Boolean OR operation
6	XOR	N,(	Data	Boolean XOR operation
7	ADD	(	Data	Addition operation
8	SUB	(	Data	Subtraction operation
9	MUL	(	Data	Multiplication operation
10	DIV	(	Data	Division operation
11	GT	(	Data	Comparison operation: > (greater than)
12	GE	(	Data	Comparison operation: >= (greater than or equal to)
13	EQ	(	Data	Comparison operation: = (equal to)
14	NE	(	Data	Comparison operation: <> (not equal)
15	LE	(	Data	Comparison operation: <= (less than or equal to)
16	LT	(	Data	Comparison operation: < (less than)
17	JMP	C, N	Label	Jump to label
18	CAL	C, N	Name	Call a function or function block
19	RET	C, N		Return from a function or function block
20	)			Evaluation deferred operation used with '('

• • •Operators from no. 4 to 16 execute the following functions:

***CR* <== *CR* Operation Operand**

After executing the operation made between CR and operand value is done, it stores the result in CR.

Example

AND %IX1.0.0 is interpreted as follows:

CR <= CR AND %IX1.0.0

• • •Comparison operator stores its Boolean result in CR after a comparison operation made between CR and the right operand.

Example

For GT %MW10, if CR is greater than the value of internal memory word 10, the value of CR will be BOOL 1. Otherwise it will be 0.

• • •The data type of CR is not modified by most of the operation instructions. However, in case of comparison operators, a data type of CR is changed.

Example

```
LD      VAL       (a)
EQ      GROSS     (b)
AND     %IX0.0.0  (c)
ST      START     (d)
```

(assume that variable START is declared as BOOL, and variable VAL and GROSS as INT)

At (a) row, the INT value of VAL is put in CR. At (b) row, after comparing the CR to INT value of GROSS, if the value is same, it puts BOOL 1 in CR; if not, CR is BOOL 0. At this time, a data type of CR changes from INT to BOOL. Accordingly, instructions of (c) and (d) rows are normal without making an error.

5.3.3.1. Basic Operator

(1) LD

Meaning	It loads a value in the current result. A data type of CR changes according to the operand data type.	
Modifier	N: If the operand is BOOL, it negates its value and loads it in CR.	
Operand	All the data types including constant are available.	
Examples	LD TRUE	The value of BOOL 1 is loaded in CR. The data type of CR is BOOL.
	LD INT_VALUE	The value of INT_VALUE is loaded in CR. The data type of CR is INT.
	LD T#1S	T#1S, time constant, is loaded in CR. The data type of CR is TIME.
	LDN B_VALUE	The value of B_VALUE is negated and is loaded in CR. The data type of CR is BOOL.

(2) ST

Meaning	It stores the current result (CR) in a variable (operand). The data type of both CR and operand should be the same. The current result is not modified by this operation.	
Modifier	N: If CR is BOOL, it negates its value and stores it in the operand. At this time, the value of CR does not change.	
Operand	All the data types except constant are available. Its data type should be the same as that of CR.	
Examples	LD FALSE	The value of BOOL 0 is loaded in CR. The data type of CR is BOOL.
	ST B_VALUE1	Stores the value of CR in variable B_VALUE1 of which data type is BOOL.
	STN B_VALUE2	Negates the value of CR and stores it in B_VALUE2 of which data type is BOOL.
	LD INT_VALUE	The value of INT_VALUE that is INT variable is loaded in CR. The data type of CR is INT.
	ST I_VALUE1	Stores the value of CR in variable I_VALUE1 of which data type is INT.
	LD D#1995-12-25	Date constant D#1995-12-25 is loaded in CR. At this time, a data type of CR is DATE.
	ST D_VALUE1	Stores the value of CR in variable D_VALUE1 of which data type is DATE.

(3) S (Set)

Meaning	If CR is BOOL 1, the operand value of which data type is BOOL will be 1. No operation is processed if CR is BOOL 0. The current result is not modified by this operation.	
Modifier	None	
Operand	Only BOOL data type is available. Constant is not available.	
Examples	LD FALSE	The value of BOOL 0 is loaded in CR. At this time, a data type of CR is BOOL.
	S B_VALUE1	No operation is processed because CR is 0. The value of B_VALUE1 does not change.
	LD TRUE	The value of BOOL 1 is loaded in CR. At this time, a data type of CR is BOOL.
	S B_VALUE2	As CR is 1, the value of B_VALUE2 whose data type is BOOL will be 1.

(4) R (Reset)

Meaning	If CR is BOOL 1, the operand value whose data type is BOOL will be 0. No operation is processed if CR is BOOL 0. The current result is not modified by this operation.	
Modifier	None	
Operand	Only BOOL data type is available. Constant is not available.	
Examples	LD FALSE	The value of BOOL 0 is loaded in CR. At this time, a data type of CR is BOOL.
	R B_VALUE1	No operation is processed because CR is 0. The value of B_VALUE1 does not change.
	LD TRUE	The value of BOOL 1 is loaded in CR. At this time, a data type of CR is BOOL.
	R B_VALUE2	As CR is 1, the value of B_VALUE2 whose data type is BOOL will be 0. The value of CR does not change.
	ST B_VALUE3	The value of CR (Boolean 1) is stored in B_VALUE3 whose data type is BOOL.

(5) AND

Meaning	After logical AND operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	N: If the operand data type is BOOL, logical AND operation is made between the operand value and CR after negating the operand value. (: If a data type of operand is BOOL, moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only BOOL, BYTE, WORD, DWORD, LWORD data types are available. Constant is also available.	
Examples	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	AND B_VALUE2	After logical AND operation for CR and the value of B_VALUE2 whose data type is BOOL, stores the result in CR.
	ANDN B_VALUE3	After negating the value of B_VALUE3, logical AND operation is made between CR and the value of B_VALUE3 whose data type is BOOL.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 AND B_VALUE2 AND NOT (B_VALUE3)
	LD W_VALUE1	The value of W_VALUE1 whose data type is WORD is loaded in CR. At this time, a data type of CR is WORD.
	AND W_VALUE2	After logical AND operation for CR and the value of W_VALUE2 whose data type is WORD, stores the result in CR.
	ST W_VALUE3	Stores CR value in W_VALUE3 whose data type is WORD. W_VALUE3 <== W_VALUE1 AND W_VALUE2
	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	AND(B_VALUE2	Moves CR value in other place and stores the value of B_VALUE2 whose data type is BOOL in CR.
	OR B_VALUE3	After logical OR operation for CR and the value of B_VALUE3 whose data type is BOOL, stores the result in CR.
	)	After logical AND operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 AND (B_VALUE2 OR B_VALUE3)

(6) OR

Meaning	After logical OR operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	N: If the operand data type is BOOL, logical AND operation is made between the operand value and CR after negating the operand value. (: If a data type of operand is BOOL, moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only BOOL, BYTE, WORD, DWORD, LWORD data types are available. Constant is also available.	
Examples	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	OR B_VALUE2	After logical OR operation for CR and the value of B_VALUE2 whose data type is BOOL, stores the result in CR.
	ORN B_VALUE3	After negating the value of B_VALUE3, logical OR operation is made between CR and the value of B_VALUE3 whose data type is BOOL.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 OR B_VALUE2 OR NOT (B_VALUE3)
	LD W_VALUE1	The value of W_VALUE1 whose data type is WORD is loaded in CR. At this time, a data type of CR is WORD.
	OR W_VALUE2	After logical AND operation for CR and the value of W_VALUE2 whose data type is WORD, stores the result in CR.
	ST W_VALUE3	Stores CR value in W_VALUE3 whose data type is WORD. W_VALUE3 <== W_VALUE1 OR W_VALUE2
	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	OR(B_VALUE2	Moves CR value in other place and stores the value of B_VALUE2 whose data type is BOOL in CR.
	AND B_VALUE3	After logical AND operation for CR and the value of B_VALUE3 whose data type is BOOL, stores the result in CR.
	)	After logical OR operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 OR (B_VALUE2 AND B_VALUE3)

(7) XOR

Meaning	After logical XOR operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	N: If the operand data type is BOOL, logical AND operation is made between the operand value and CR after negating the operand value. (: If a data type of operand is BOOL, moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only BOOL, BYTE, WORD, DWORD, LWORD data types are available. Constant is also available.	
Examples	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	XOR B_VALUE2	After logical XOR operation for CR and the value of B_VALUE2 whose data type is BOOL, stores the result in CR.
	XORN B_VALUE3	After negating the value of B_VALUE3, logical XOR operation is made between CR and the value of B_VALUE3 whose data type is BOOL.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 XOR B_VALUE2 XOR NOT (B_VALUE3)
	LD W_VALUE1	The value of W_VALUE1 whose data type is WORD is loaded in CR. At this time, a data type of CR is WORD.
	XOR W_VALUE2	After logical XOR operation for CR and the value of W_VALUE2 whose data type is WORD, stores the result in CR.
	ST W_VALUE3	Stores CR value in W_VALUE3 whose data type is WORD. W_VALUE3 <== W_VALUE1 XOR W_VALUE2
	LD B_VALUE1	The value of B_VALUE1 whose data type is BOOL is loaded in CR. At this time, a data type of CR is BOOL.
	XOR(B_VALUE2	Moves CR value in other place and stores the value of B_VALUE2 whose data type is BOOL in CR.
	AND B_VALUE3	After logical AND operation for CR and the value of B_VALUE3 whose data type is BOOL, stores the result in CR.
	)	After logical XOR operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST B_VALUE4	Stores CR value in B_VALUE4 whose data type is BOOL. B_VALUE4 <== B_VALUE1 XOR (B_VALUE2 AND B_VALUE3)

(8) ADD

Meaning	After addition operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	(: Moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL data types are available. Constant is also available.	
Examples	LD I_VALUE1	The value of I_VALUE1 whose data type is INT is loaded in CR. At this time, a data type of CR is INT.
	ADD I_VALUE2	After ADD operation for CR and the value of I_VALUE2 whose data type is INT, stores the result in CR.
	ST I_VALUE3	Stores CR value in I_VALUE3 whose data type is INT. I_VALUE3 <== I_VALUE1 + I_VALUE2
	LD D_VALUE1	The value of D_VALUE1 whose data type is DINT is loaded in CR. At this time, a data type of CR is DINT.
	ADD(D_VALUE2	Moves CR value in other place and stores the value of D_VALUE2 whose data type is DINT in CR.
	DIV D_VALUE3	After DIV operation for CR and the value of D_VALUE3 whose data type is DINT, stores the result in CR.
	)	After ADD operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST D_VALUE4	Stores the CR value in D_VALUE4 whose data type is DINT. D_VALUE4 <== D_VALUE1 + (D_VALUE2 / D_VALUE3)

(9) SUB

<table>
<tr><td>Meaning</td><td colspan="2">After subtraction operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.</td></tr>
<tr><td>Modifier</td><td colspan="2">(: Moves CR value in other place for a while and stores the operand value in CR (deferred operation).</td></tr>
<tr><td>Operand</td><td colspan="2">Only SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL data types are available.
Constant is also available.</td></tr>
<tr><td>Examples</td><td>LD I_VALUE1</td><td>The value of I_VALUE1 whose data type is INT is loaded in CR. At this time, a data type of CR is INT.</td></tr>
<tr><td></td><td>SUB I_VALUE2</td><td>After SUB operation for CR and the value of I_VALUE2 whose data type is INT, stores the result in CR.</td></tr>
<tr><td></td><td>ST I_VALUE3</td><td>Stores CR value in I_VALUE3 whose data type is INT.
I_VALUE3 <== I_VALUE1 - I_VALUE2</td></tr>
<tr><td></td><td>LD D_VALUE1</td><td>The value of D_VALUE1 whose data type is DINT is loaded in CR. At this time, a data type of CR is DINT.</td></tr>
<tr><td></td><td>SUB(D_VALUE2</td><td>Moves CR value in other place and stores the value of D_VALUE2 whose data type is DINT in CR.</td></tr>
<tr><td></td><td>MUL D_VALUE3</td><td>After MUL operation for CR and the value of D_VALUE3 whose data type is DINT, stores the result in CR.</td></tr>
<tr><td></td><td>)</td><td>After SUB operation for the current CR value and the moved CR value stored in other place, stores the result in CR.</td></tr>
<tr><td></td><td>ST D_VALUE4</td><td>Stores the CR value in D_VALUE4 whose data type is DINT. D_VALUE4 <== D_VALUE1 - (D_VALUE2 X D_VALUE3)</td></tr>
</table>

(10) MUL

Meaning	After multiplication operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	(: Moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL data types are available. Constant is also available.	
Examples	LD I_VALUE1	The value of I_VALUE1 whose data type is INT is loaded in CR. At this time, a data type of CR is INT.
	MUL I_VALUE2	After MUL operation for CR and the value of I_VALUE2 whose data type is INT, stores the result in CR.
	ST I_VALUE3	Stores CR value in I_VALUE3 whose data type is INT. I_VALUE3 <== I_VALUE1 X I_VALUE2
	LD D_VALUE1	The value of D_VALUE1 whose data type is DINT is loaded in CR. At this time, a data type of CR is DINT.
	MUL(D_VALUE2	Moves CR value in other place and stores the value of D_VALUE2 whose data type is DINT in CR.
	SUB D_VALUE3	After SUB operation for CR and the value of D_VALUE3 whose data type is DINT, stores the result in CR.
	)	After MUL operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST D_VALUE4	Stores the CR value in D_VALUE4 whose data type is DINT. D_VALUE4 <== D_VALUE1 X (D_VALUE2 - D_VALUE3)

(11) DIV

Meaning	After division operation for CR and the operand value, stores the operation result in CR. At this time, a data type of both CR and the operand should be the same. The operand value does not change.	
Modifier	(: Moves CR value in other place for a while and stores the operand value in CR (deferred operation).	
Operand	Only SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL data types are available. Constant is also available.	
Examples	LD I_VALUE1	The value of I_VALUE1 whose data type is INT is loaded in CR. At this time, a data type of CR is INT.
	DIV I_VALUE2	After DIV operation for CR and the value of I_VALUE2 whose data type is INT, stores the result in CR.
	ST I_VALUE3	Stores CR value in I_VALUE3 whose data type is INT. I_VALUE3 <== I_VALUE1 / I_VALUE2
	LD D_VALUE1	The value of D_VALUE1 whose data type is DINT is loaded in CR. At this time, a data type of CR is DINT.
	DIV(D_VALUE2	Moves CR value in other place and stores the value of D_VALUE2 whose data type is DINT in CR.
	ADD D_VALUE3	After ADD operation for CR and the value of D_VALUE3 whose data type is DINT, stores the result in CR.
	)	After DIV operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
	ST D_VALUE4	Stores the CR value in D_VALUE4 whose data type is DINT. D_VALUE4 <== D_VALUE1 / (D_VALUE2 + D_VALUE3)

(12) GT

<table>
<tr><td>Meaning</td><td colspan="2">After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.</td></tr>
<tr><td>Modifier</td><td colspan="2">(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).</td></tr>
<tr><td>Operand</td><td colspan="2">All the data types except ARRAY are available.
Constant is also available.</td></tr>
<tr><td>Examples</td><td></td><td>In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 70,</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GT I_VAL2</td><td>After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR.
(As I_VAL1 < I_VAL2, CR will be 0)</td></tr>
<tr><td></td><td>ST B_VAL1</td><td>Stores CR value in B_VAL1 whose data type is BOOL.
B_VAL1 &lt;== FALSE</td></tr>
<tr><td></td><td>LD I_VAL2</td><td>The value of I_VAL2 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GT I_VAL1</td><td>After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR.
(As I_VAL1 < I_VAL2, CR will be 1)</td></tr>
<tr><td></td><td>ST B_VAL2</td><td>Stores CR value in B_VAL2 whose data type is BOOL.
B_VAL2 &lt;== TRUE</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GT(I_VAL2</td><td>Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.</td></tr>
<tr><td></td><td>SUB I_VAL3</td><td>After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.</td></tr>
<tr><td></td><td>)</td><td>After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR. (As the stored CR > current CR, CR will be 1)</td></tr>
<tr><td></td><td>ST B_VAL3</td><td>Stores the CR value in B_VAL3 whose data type is BOOL.
B_VAL3 &lt;== TRUE</td></tr>
</table>

(13) GE

<table>
<tr><td>Meaning</td><td colspan="2">After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.</td></tr>
<tr><td>Modifier</td><td colspan="2">(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).</td></tr>
<tr><td>Operand</td><td colspan="2">All the data types except ARRAY are available.
Constant is also available.</td></tr>
<tr><td>Examples</td><td></td><td>In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 70,</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GE I_VAL2</td><td>After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR.
(As I_VAL1 < I_VAL2, CR will be 0)</td></tr>
<tr><td></td><td>ST B_VAL1</td><td>Stores CR value in B_VAL1 whose data type is BOOL.
B_VAL1 <== FALSE</td></tr>
<tr><td></td><td>LD I_VAL2</td><td>The value of I_VAL2 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GE I_VAL1</td><td>After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR.
(As I_VAL1 < I_VAL2, CR will be 1)</td></tr>
<tr><td></td><td>ST B_VAL2</td><td>Stores CR value in B_VAL2 whose data type is BOOL.
B_VAL2 <== TRUE</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>GE(I_VAL2</td><td>Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.</td></tr>
<tr><td></td><td>SUB I_VAL3</td><td>After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.</td></tr>
<tr><td></td><td>)</td><td>After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
(As the stored CR > current CR, CR will be 1)</td></tr>
<tr><td></td><td>ST B_VAL3</td><td>Stores the CR value in B_VAL3 whose data type is BOOL.
B_VAL3 <== TRUE</td></tr>
</table>

(14) EQ

Meaning	After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.	
Modifier	(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).	
Operand	All the data types except ARRAY are available. Constant is also available.	
Examples		In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 50,
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	EQ I_VAL2	After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR. (As I_VAL1 < I_VAL2, CR will be 0)
	ST B_VAL1	Stores CR value in B_VAL1 whose data type is BOOL. B_VAL1 <== FALSE
	LD I_VAL1	The value of I_VAL2 whose data type is INT is loaded in CR.
	EQ I_VAL3	After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR. (As I_VAL1 = I_VAL3, CR will be 1)
	ST B_VAL2	Stores CR value in B_VAL2 whose data type is BOOL. B_VAL2 <== TRUE
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	EQ(I_VAL2	Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.
	SUB I_VAL3	After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.
	)	After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR. (As the stored CR = current CR, CR will be 1)
	ST B_VAL3	Stores the CR value in B_VAL3 whose data type is BOOL. B_VAL3 <== TRUE

(15) NE

<table>
<tr><td>Meaning</td><td colspan="2">After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.</td></tr>
<tr><td>Modifier</td><td colspan="2">(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).</td></tr>
<tr><td>Operand</td><td colspan="2">All data types except ARRAY are available.
Constant is also available.</td></tr>
<tr><td>Examples</td><td></td><td>In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 50,</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>NE I_VAL3</td><td>After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR.
(As I_VAL1 = I_VAL3, CR will be 0)</td></tr>
<tr><td></td><td>ST B_VAL1</td><td>Stores CR value in B_VAL1 whose data type is BOOL.
B_VAL1 <== FALSE</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL2 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>NE I_VAL2</td><td>After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR.
(As I_VAL1 <> I_VAL2, CR will be 1)</td></tr>
<tr><td></td><td>ST B_VAL2</td><td>Stores CR value in B_VAL2 whose data type is BOOL.
B_VAL2 <== TRUE</td></tr>
<tr><td></td><td>LD I_VAL1</td><td>The value of I_VAL1 whose data type is INT is loaded in CR.</td></tr>
<tr><td></td><td>NE(I_VAL2</td><td>Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.</td></tr>
<tr><td></td><td>SUB I_VAL3</td><td>After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.</td></tr>
<tr><td></td><td>)</td><td>After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR.
(As the stored CR = current CR, CR will be 0)</td></tr>
<tr><td></td><td>ST B_VA3</td><td>Stores the CR value in B_VAL3 whose data type is BOOL.
B_VAL2 <== FALSE</td></tr>
</table>

(16) LE

Meaning	After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.	
Modifier	(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).	
Operand	All data types except ARRAY are available. Constant is also available.	
Examples		In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 70,
	LD I_VAL2	The value of I_VAL2 whose data type is INT is loaded in CR.
	LE I_VAL1	After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR. (As I_VAL1 < I_VAL2, CR will be 0)
	ST B_VAL1	Stores CR value in B_VAL1 whose data type is BOOL. B_VAL1 <== FALSE
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	LE I_VAL2	After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR. (As I_VAL1 < I_VAL2, CR will be 1)
	ST B_VAL2	Stores CR value in B_VAL2 whose data type is BOOL. B_VAL2 <== TRUE
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	LE(I_VAL2	Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.
	SUB I_VAL3	After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.
	)	After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR. (As the stored CR > current CR, CR will be 0)
	ST B_VA3	Stores the CR value in B_VAL3 whose data type is BOOL. B_VAL2 <== FALSE

(17) LT

Meaning	After comparison operation for CR and the operand value, stores the BOOL result in CR. CR will be 1 only if CR is greater than operand. A data type of both CR and the operand should be the same. The operand value does not change. After operation, a data type of CR will be BOOL regardless of the operand data type.	
Modifier	(: Moves CR value in other place for a while and stores the value of operand in CR (deferred operation).	
Operand	All data types except ARRAY are available. Constant is also available.	
Examples		In case that I_VAL1 = 50, I_VAL2 = 100 IVAL_3 = 70,
	LD I_VAL2	The value of I_VAL2 whose data type is INT is loaded in CR.
	LT I_VAL1	After comparison operation for CR and the value of I_VAL1 whose data type is INT, stores the result in CR. (As I_VAL1 < I_VAL2, CR will be 0)
	ST B_VAL1	Stores CR value in B_VAL1 whose data type is BOOL. B_VAL1 <== FALSE
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	LT I_VAL2	After comparison operation for CR and the value of I_VAL2 whose data type is INT, stores the result in CR. (As I_VAL1 < I_VAL2, CR will be 1)
	ST B_VAL2	Stores CR value in B_VAL2 whose data type is BOOL. B_VAL2 <== TRUE
	LD I_VAL1	The value of I_VAL1 whose data type is INT is loaded in CR.
	LT(I_VAL2	Moves CR value in other place and stores the value of I_VAL2 whose data type is INT in CR.
	SUB I_VAL3	After SUB operation for CR and the value of I_VAL3 whose data type is INT, stores the result in CR.
	)	After comparison operation for the current CR value and the moved CR value stored in other place, stores the result in CR. (As the stored CR > current CR, CR will be 0)
	ST B_VA3	Stores the CR value in B_VAL3 whose data type is BOOL. B_VAL2 <== FALSE

(18) JMP

Meaning	Jumps to the specified label.	
Modifier	C: If CR whose data type is BOOL is TRUE (1), it jumps to the specified label. If CR whose data type is BOOL is FALSE (0), it does not jump to the specified label but executes the next instruction. N: If CR whose data type is BOOL is FALSE (0), it jumps to the specified label. If CR whose data type is BOOL is TRUE (1), it does not jump to the specified label but executes the next instruction. If there is no modifier, it jumps to the label regardless of CR value.	
Operand	Label defined in the same IL program.	
Examples		This is a program that stores the value of I_VAL1 or I_VAL2 in I_VAL3 according to the value of B_VAL1 whose data type is BOOL.
	LD B_VAL1	The value of B_VAL1 whose data type is BOOL is loaded in CR.
	JMPC THERE1	If CR is 1, it jumps to THERE1 label; if CR is 0, it executes the next instruction.
	LD I_VAL1	CR <== I_VAL1
	JMP THERE2	Jumps to THERE2 label unconditionally.
	THERE1:	THERE1 label
	LD I_VAL2	CR <== I_VAL2
	THERE2:	THERE2 label
	ST I_VAL3	I_VAL3 <== CR
		This is a program that executes SEL function if the value of B_VAL2 whose data type is BOOL is 1.
	LD B_VAL2	CR <== B_VAL2
	JMPN THERE3	If CR is 0 (FALSE), it jumps to THERE3 label.
	LD B_VALUE	CR <== B_VALUE

	SEL
	G:= CURRENT RESULT
	IN1:= I_VAL1
	IN2:= I_VAL2
		Calls SEL function.

	ST I_VAL3	I_VAL3 <== CR
	THERE3:	THERE3 label

(19) CAL

Meaning	Calls the function block whose name is described in the operand section.	
Modifier	C: if CR whose data type is BOOL is TRUE (1), it calls a function block. If CR whose data type is BOOL is FALSE (0), it does not call a function block. N : if CR whose data type is BOOL is FALSE (0), it calls a function block. If CR whose data type is BOOL is TRUE (1), it does not call a function block. If there is no modifier, it calls a function block regardless of CR.	
Operand	Function block name	
Examples		This is a program that if the value of B_VAL1 whose data type is BOOL is 1(TRUE), calls the TON (on-delay timer).
	LD B_VAL1	The value of B_VAL1 whose data type is BOOL is loaded in CR.

	CALC TON TIMER1
	IN:= T_INPUT
	PT:= PRE_TIME
		If CR is 1, it calls the on-delay timer, TON whose instance is TIMER1.

		This is a program that calls the CTU, (up counter), if the value of B_VAL2 whose data type is BOOL is 0 (FALSE).
	LD B_VAL2	The value of B_VAL2 whose data type is BOOL is loaded in CR.

	CALN CTU COUNT1
	CU:= B_UP
	R:= B_RESET
	PV:= 100
		If CR is 1, it calls the CTU (up counter) whose instance is COUNT1.

		This is a program that calls the CTD (down-counter) regardless of CR.

	CAL CTD COUNT2
	CD:= B_DOWN
	LD:= B_LDV
	PV:= 300
		Calls the CTD (down-counter) whose instance is COUNT2.

(20) RET

Meaning	Returns from a function or function block.	
Modifier	C: if CR whose data type is BOOL is TRUE (1), it returns. If CR whose data type is BOOL is FALSE (0), it does not return. N: if CR whose data type is BOOL is FALSE (0), it returns. If CR whose data type is BOOL is TRUE (1), it does not return. If there is no modifier, it returns regardless of CR.	
Operand	None	
Examples		This is a function that stores the result in I_VAL3 after MUL operation for the value of I_VAL1 whose data type is INT and the value of I_VAL2 whose data type is INT. At this time, if an operation error occurs in MUL operation, it returns after storing 0 in I_VAL3.
	LD I_VAL1	
	MUL I_VAL2	
	ST I_VAL3	
	LD _ERR	CR <== system error flag
	RETN	If CR is 0, instance will return.
	LD 0	
	ST I_VAL3	I_VAL3 <== 0
	RET	Returns unconditionally.

(21))

Meaning	Evaluation deferred operation used with ‘(‘.	
Modifier	None	
Operand	None	
Examples	LD I_VAL1 ADD I_VAL2 MUL I_VAL3 ST I_VAL4	I_VAL4 <== (I_VAL1 + IVAL2) X I_VAL3
	LD I_VAL1 ADD(I_VAL2 MUL I_VAL3) ST I_VAL4	I_VAL4 <== I_VAL1 + (IVAL2 X I_VAL3)
	LD L_VAL1 ADD(L_VAL2 MUL(L_VAL3 SUB L_VAL4) ADD L_VAL5) DIV L_VAL6 ST L_VAL7	L_VAL7 <== (L_VAL1 + (L_VAL2 X (L_VAL3 - L_VAL4) + L_VAL5)) / L_VAL6

5.4. Calling of Function and Function Block

- ••• Calls a function using its name as an operator.
- ••• When calling a function, CR is stored as the first input.
- ••• If a function has more than one input, assign the input value and then call a function.
- ••• The output value of a function will be stored in CR.
- ••• A data type of CR will be the output data type a function.

Example

```
LD              VAL
SIN
ST              RESULT     (VAL and RESULT are regarded as a REAL data type)
```

If you store the value of VAL in CR at the first row and call SIN function at the second row, then the CR value will be stored in SIN function as a first input. And it does not need other inputs because SIN function has only one input, and the output value will be stored in CR after executing SIN function. At the third row, CR will be stored in RESULT variable.

```
LD              %IX0.0.0
SEL     G:=     CURRENT RESULT
        IN0:=   VAL1
        IN1:=   VAL2
ST      VAL3
```

This is the example of a function that has several inputs. CR is set at the first row and is loaded in SEL function as a first input value. If you assign each value for the rest inputs and call SEL function, the result will be stored in CR and CR value will be stored in variable VAL3.

- ••JMP (JMPN, JMPC) instructions are used to call a function conditionally.

Example

```
LD              %IX0.0.0
JMPN             THERE
LD              I_VAL1
ADD    IN1:=    CURRENT RESULT
       IN2:=    I_VAL2
       IN3:=    I_VAL3
ST              I_VAL4
THERE:
```

%IX0.0.0 value is loaded in CR whose data type is BOOL at the first row. And if the value is 0 at the second row, it jumps to THERE: label. If %IX0.0.0 value is 1, it does not execute JMP instruction but does the next row.

- ••When calling a function block, CAL is used as an operator and the instance name as an operand that is previously declared.
- ••CAL INSTANCE /* call a function block unconditionally. */

```
CAL      INSTANCE    /* call a function block unconditionally. */
CALN     INSTANCE    /* if CR is BOOL 0, call a function block. */
CALC     INSTANCE    /* if CR is BOOL 1, call a function block. */
```

Here, INSTANCE should be previously declared as an instance of a function block.

- ••CR is not loaded in a function block input. So it is required to assign all the input values necessary for a function block. Besides output value is not stored in CR.

Example

On-Delay Timer function block

```
LD                   %IX0.0.0
CALC        TON      TIMER0
            IN:=     %IX0.1.2
            PT:=     T#200S
LD                   TIMER0.Q
ST          %QX1.0.2
```

(assume that TIMER0 is declared as an instance of TON)

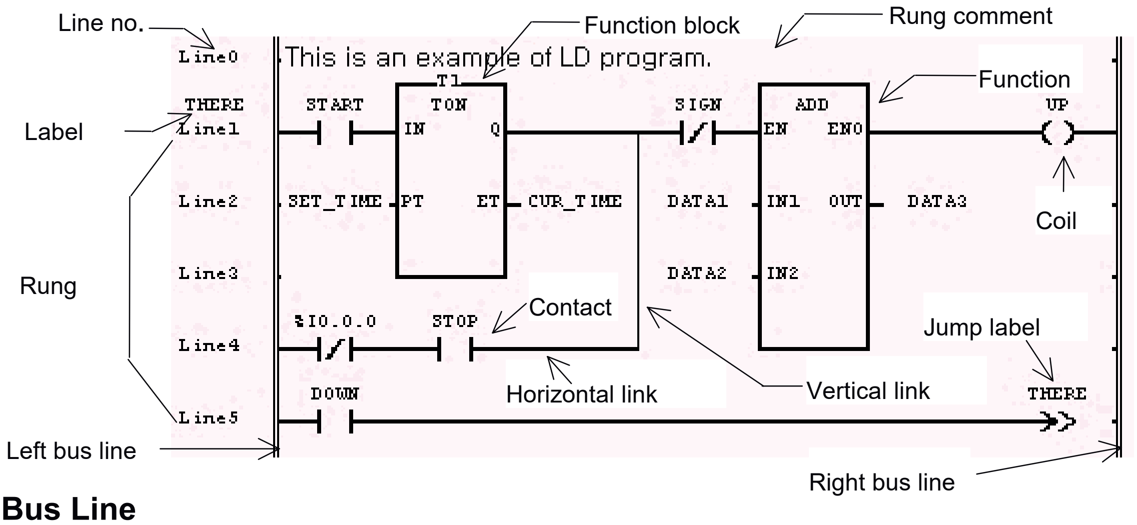
On-delay timer has 2 inputs and calls it after assigning its input values, respectively. If users want to use the result values, they can do it like the fifth row in the above program because the result values are stored in TIMER0.Q and TIMER0.ET respectively.

MEMO

6. LD (Ladder Diagram)

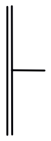
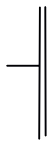
6.1. Overview

- ••LD program represents PLC program through graphic signs such as coil or contact used in relay logic diagram.
- ••Configuration



6.2. Bus Line

- ••Bus line as power line is placed vertically on both left and right sides on LD graphic diagram.

No.	Symbol	Description
1		Left bus line Its value is always 1 (BOOL).
2		Right bus line The value is not fixed.

6.3. Connection Line

- ••The value (BOOL 1) of left bus line is transmitted to the right side by the ladder diagram. The line that has the transmitted value is called as 'power flow line' or 'connection line' which is connected to a contact or coil. Power flow line has always a BOOL value and there's only one power flow line in one rung that is connected by lines.

- ••There are two types of a connection line of LD: horizontal connection line and vertical connection line.

No.	Symbol	Description
1	────	Horizontal connection line It transmits the left side value to the right side.
2	│	Vertical connection line It's logical OR of horizontal connection lines of its left side.

6.4. Contact

- ••'Contact' transmits a value to the right horizontal connection line, which is the result of logical AND operation of these: the state of left horizontal connection line, Boolean input/output related to the current contact, or memory variables. It does not change the value of variable itself related to the contact. Standard contact symbols are as follows:

Static contact		
No.	Symbol	Description
1	*** —| |—	Normally open contact When the addressed memory bit (marked with ***) is ON, the instruction is TRUE, which transmits the state of the left connection line to the right one. Otherwise the state of the right connection line is OFF.
2	*** —|/|—	Normally closed contact When the addressed memory bit (marked with ***) is OFF, the instruction is TRUE, which transmits the state of the left connection line to the right one. Otherwise the state of the right connection line is OFF.
State transition-sensing contact		
3	*** —|P|—	Positive transition-sensing contact When the addressed memory bit (marked with ***) that was OFF in the previous scan is ON, it maintains ON state during one scan (current scan).
4	*** —|N|—	Negative transition-sensing contact When the addressed memory bit (marked with ***) that was ON in the previous scan is OFF, it maintains ON state during just one scan (current scan).

••

6.5. Coil

- ••Coil stores the state of the left connection line or the processing result of state transition in the associated BOOL variable. Standard coil symbols are as follows:

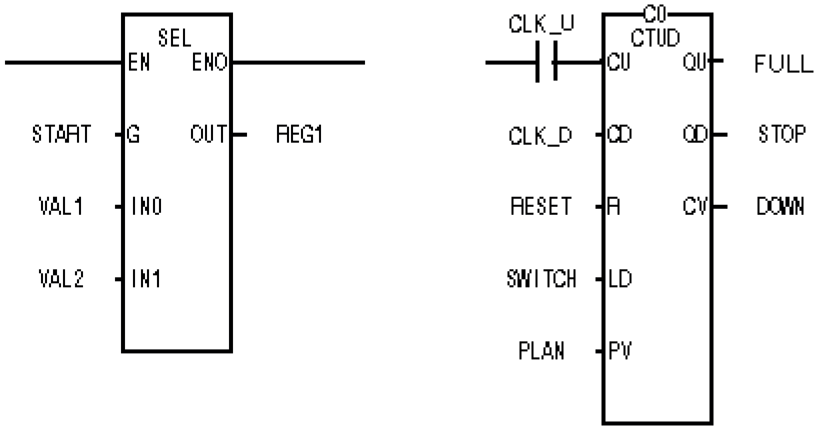
No.	Symbol	Description
		Momentary Coils
1	*** —( )—	Coil When the rung is TRUE, the addressed memory bit (marked with ***) is set ON. If the bit controls an output device, that output device will be ON.
2	*** —(/)—	Negated coil When the rung is TRUE, the addressed memory bit (marked with ***) is set OFF. That is, if the state of left connection line is OFF, the associated variable is ON and if the state of left connection line is ON, the associated variable is OFF. If the bit controls an output device, that output device will be OFF.
		Latched Coils
3	*** —(S)—	Set coil It sets the associated variable (marked with ***) to ON when the left link is in the ON state or TRUE and remains set until reset by a Reset coil. When the left link is OFF or FALSE, the associated variable is not affected by the Set coil element.
4	*** —(R)—	Reset coil It sets the associated variable (marked with ***) to OFF when the left link is in the ON state or TRUE and remains reset until set by a Set coil. When the left link is OFF or FALSE, the associated variable is not affected by the Reset coil element.
		State Transition-sensing Coils
5	*** —(P)—	Positive transition-sensing coil If the state of its left connection that was OFF in the previous scan is ON in the current scan, the associated variable (marked with ***) is ON during the current scan.
6	*** —(N)—	Negative transition-sensing coil If the state of its left connection that was ON in the previous scan is OFF in the current scan, the associated variable (marked with ***) is ON during the current scan.

- ••Coils are placed in the rightmost side of LD, of which right side is a right bus line.

6.6. Calling of Function and Function Block

- The connection to a function and function block will be done by putting suitable data or variable to their input/output.

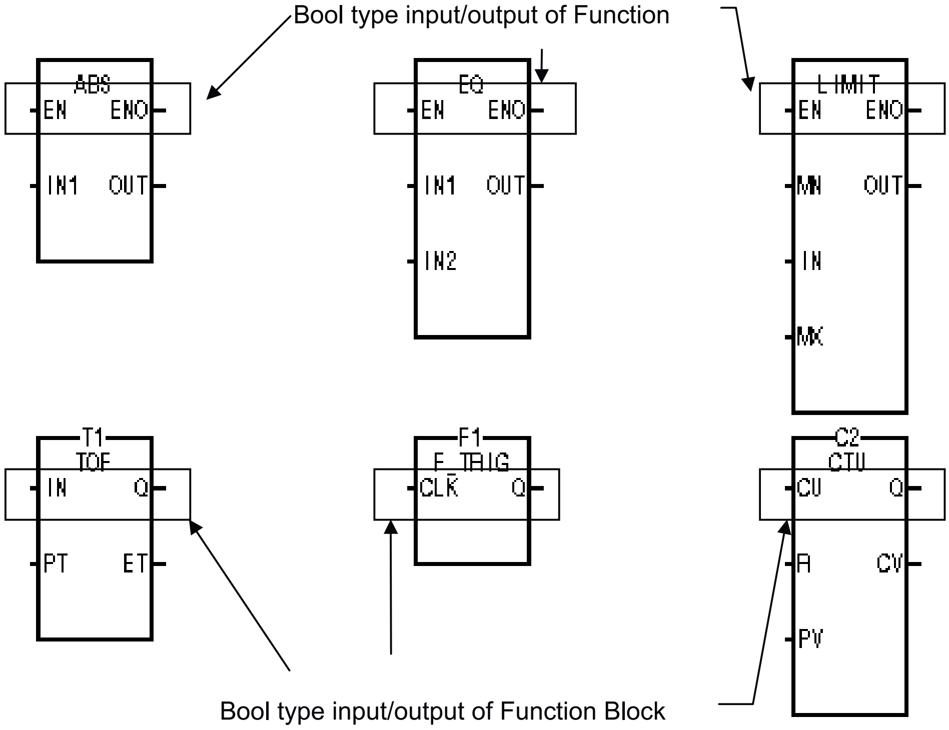
Example



Function Function block

- There should be at least one BOOL-type input and BOOL-type output in a function or function block if you want to enable them. EN and ENO are BOOL-type input/output in a function while a data type of the first input and first output are BOOL-type in a function block.

Example

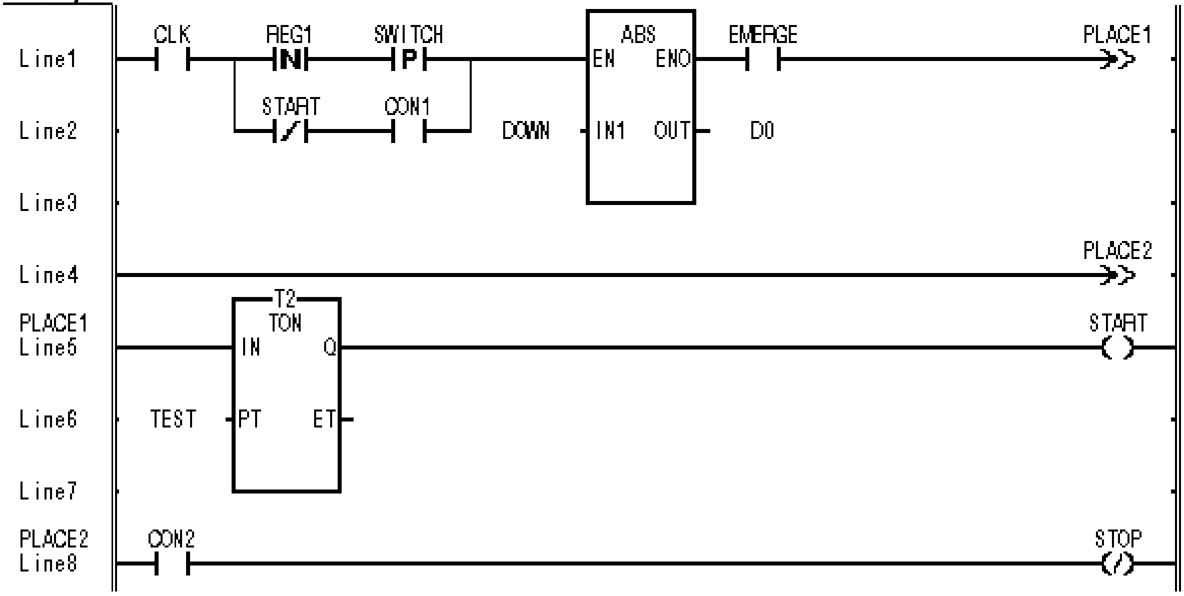

Bool type input/output of Function

Bool type input/output of Function Block

••

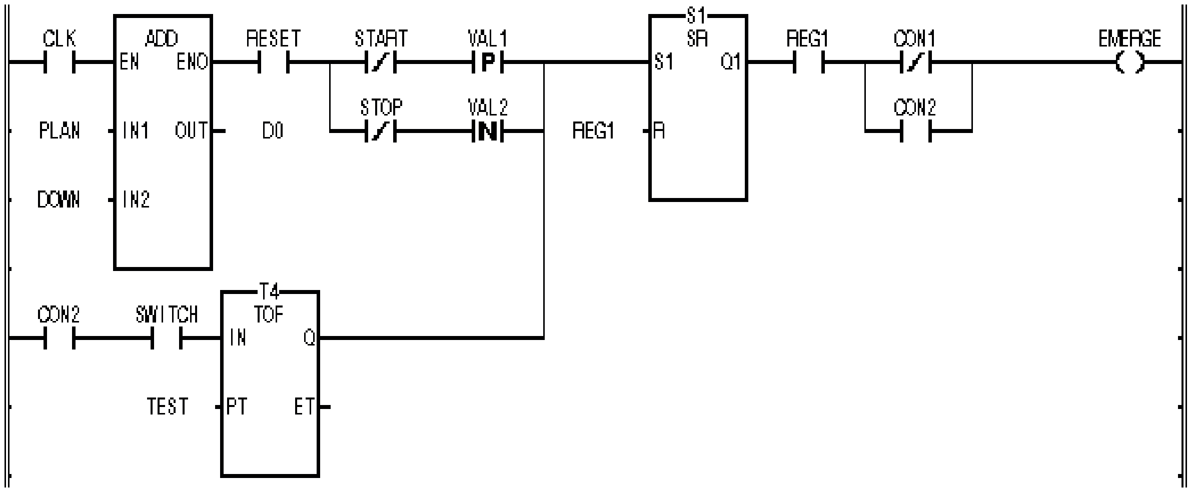
- ••Function in LD is different from that of IL. By convention the ladder logic connected Boolean input to a function is called EN and the corresponding output Boolean is called ENO, or enable out. If the value of EN is 1, then the function is executed, otherwise it is not executed. In all cases, the default is for the value of EN to be copied to the output ENO. If, for whatever reason, an error occurs in the execution of a function, the function is responsible to set ENO to FALSE (BOOL 0). EN is connected to the power flow line but ENO doesn't have to be connected to it. However, when connecting the power flow line to the function output instead of ENO, output data type should be a BOOL type. Note that only one power flow line can be connected to a function (when connecting the power flow line to the function output not ENO, do not connect anything to ENO output). All the inputs of a function are assigned by entering its data. The output of a function is stored at the output variable in the right side of it.
- ••You can use a function block in LD as you do in IL. Inputs of a function block are assigned much the same as a function. A function block is called when the left link is TRUE and not called when the left link is FALSE. The value of the left link IN is copied to the right link Q for further processing. The name of the function block is the "instance" name, which can be user-defined and must be unique to LD in which the function block appears. You don't have to assign output variables because they are in the instance. If a function block is connected to the power flow line, it is always executed because there is neither EN nor ENO in it. Therefore, it is required to use Jump (-->>) to determine whether or not to execute a function block according to the logic result. When connecting the power flow line to the function block, it is required to connect it to the input/output of which data type is BOOL.

Example



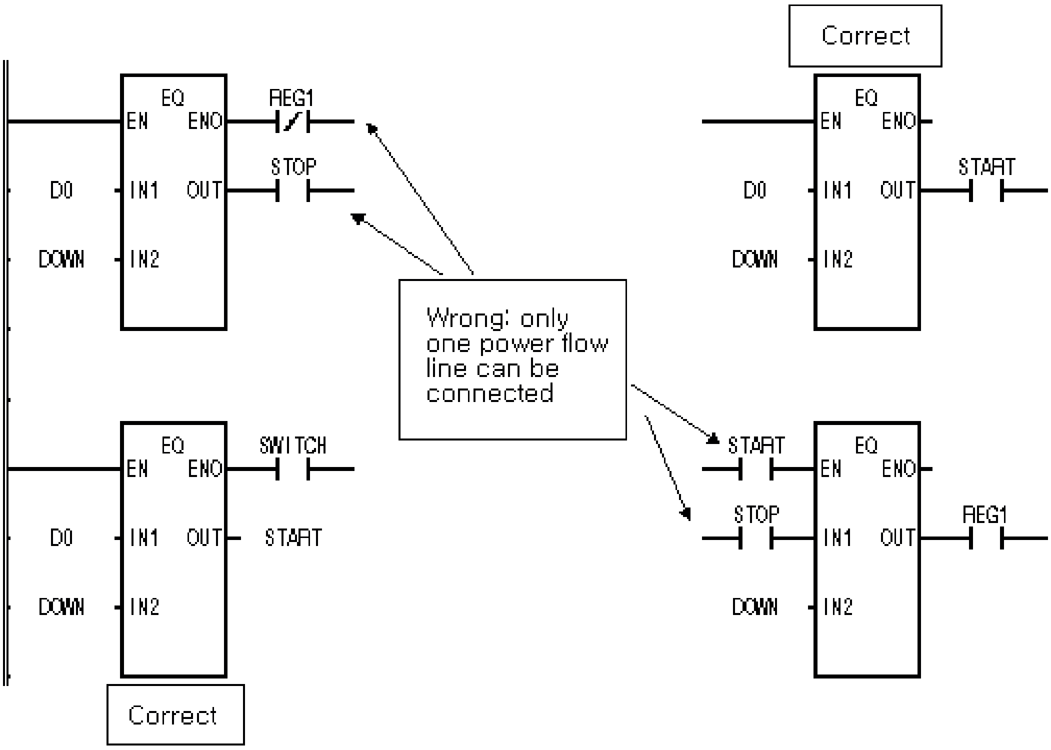
- ••You can place a function and function block in any place of LD. It is available to make a program by connecting the power flow line to their output and then putting the contact to that.

Example



- ••Only one power flow line can be connected to a function or function block.

Example



Correct

Wrong: only one power flow line can be connected

Correct

MEMO

••

7. Function and Function Block

It’s a list of function and function block. For each function and function block, please refer to the next chapter.

7.1. Function

7.1.1. Type Conversion Function

It converts each input data type into an output data type.••

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		
				[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

••

••

7.1.2. Arithmetic Function

7.1.2.1. Numerical Operation Function with One Input

•• It supports GMR, GM1, GM2 (Note: ABS function supports GM3, GM4, GM6, GM7).

•••••• •••••••••		••••••••••••••••••••••••••••••••
•••••••••••••••••••••••••••••••••		
•••	••••	••
•••	•••••	•••
•••••••••••••••••••		
•••	•••	••
•••	••••	•••
•••	••••	•••
•••		
•••	••••	•••••••••••••••••••••••••••••••••
•••	••••	•••••••••••••••••••••••••••••••••••••
•••	••••	•••••••••••••••••••••••••••••••••••••••
•••	•••••	••
••••	•••••	••
••••	•••••	••
•••••••••••••••••••••••••••••		
••••	•••••••••••••••••	•• ••
••••	•••••••••••••••••••••	
••••	•••••••••••••••••	•• ••
••••	•••••••••••••••••••••	

••

7.1.2.2. Basic Arithmetic Function

• ••

•••••• •••••••••		••••••••••••••••••••••••••••••••
••		
•••	••••	••
•••	••••	••
••		
•••	••••	•••
•••	••••	•••
•••	••••	••
•••	•••••	••
•••	•••••	••
•••••••••••••••••••••••••••••••••••••••		
•••	•••••••••••••••••	Exchanges two input data••

7.1.3. Bit Array Function

7.1.3.1. Bit-shift Function

••••••	•••••••••	••••••••••••••••••••••••••••••••
•••	••••	•••••••••••••••••••••••••
•••	••••	•••••••••••••••••••••••••••
•••	••••••••••••	••
•••	••••	•••••••••••••••••••••••••••
•••	••••	•••••••••••••••••••••••••••••
•••	••••••••••••••••••••••	Rotates a designated direction••

7.1.3.2. Bit Operation Function

••••••	•••••••••	••
•••	••••	••
•••	•••	••
•••	••••	••
•••	••••	••

7.1.4. Selection Function

••••••	•••••••••	••
•••	••••	•••
•••	••••	••
•••	••••	••
•••	••••••	•••
•••	••••	••

7.1.5. Data Exchange Function

••••••	•••••••••	••••••••••••••••••••••••••••••••
•••	••••••••••••••••••••	Swaps upper nibble for lower nibble data.••
	••••••••••••••••••••	Swaps upper byte for lower byte data.••
	••••••••••••••••••••	Swaps upper word for lower word data.••
	••••••••••••••••••••	Swaps upper double word for lower double word data.••
•••	••••••••••••••••••••	Swaps upper/lower nibble of byte elements.••
	••••••••••••••••••••	Swaps upper/lower byte of WORD elements.••
	••••••••••••••••••••	Swaps upper/lower WORD of DWORD elements.••
	••••••••••••••••••••	Swaps upper/lower DWORD of LWORD elements.••

••

7.1.6. Comparison Function

•••••	•••••••••	••
•••	••••••	'••••••••••••••••••••••••••••••••••••• •••
•••	••••••	'••• •••
•••	••••••	'••••••••••••••••••••••••••••••••• •••
•••	•••	••• •••
•••	••••••	'•••••••••••••••••••••••••••••••••• •••
•••	••••••	'••••••••••••••••••••••••••••••••••••• •••

7.1.7. Character String Function

•••••	••••••••••••••••••••	Description
•••	••••••••	Find a length of a character string
•••	••••••••••	Take a left side of a string
•••	••••••••••••	Take a right side of a string
•••	••••••••	Take a middle side of a string
•••	••••••••••••••	Concatenate the input character string in order
•••	••••••••••••••	Insert a string
•••	••••••••••••••	Delete a string
•••	••••••••••••••••	Replace a string
•••	••••••••••	Find a string

••
••

• •

7.1.8. Time/Time of Day/Date and Time of Day Function

••••••	•••••••••	•••••••••••••••••••••••••••••••••
•••	•••••••••	•••
•••	•••••••••	••••••••••••••••••••••••••••••••••••
	•••••••••	••••••••••••••••••••••••••••••••••••
	••••••••	•••••••••••••••••••••••••••••••••
	•••••••	••••••••••••••••••••••••••••••
•••	•••••••••	••••••••••••••••••••••••••••••••••••
•••	•••••••••	••••••••••••••••••••••••••••••••
•••	•••••••••••••••••••••••	Concatenate date with TOD• •

7.1.9. System Control Function

••••••	•••••••••	•••••••••••••••••••••••••••••••••
•••	••••••	Invalidates interrupt (Not to permit task program starting)• •
•••	••••••	Permits running for a task program• •
•••	•••••	••
•••	••••••••••••	Emergency running stop by a program• •
•••	•••••••••	•••
•••	••••••••	•••
•••	••••••••	•••
•••	••••	•••
•••	•••••••	•••

• •

••

••

7.1.10. Data Manipulation Function

••••••	•••••••••	•••••••••••••••••••••••••••••••••
•••	••••••••••••••••	Compare whether two inputs are equal after masking••
•••	••••••••	••
•••	••••••••	•••••••••••••••••••••••••••••••
•••	••••••••••••••••••	••
•••	••••••••••••••••••	••
•••	••••••••••••••••••••	•••
•••	••••••••••••••••••••	••
•••	••••••••••••••••••••	••
•••	••••••••••••••••••••	•••
••••	••••••••••••••••••••	•••
••••	••••••••••••••••••••	••
••••	••••••••••••••••••	•••
••••	••••••••••••••••••	Puts a character in a string••
••••	•••••••••••••••	•••
••••	••••••••••••••••••••	•••

••

7.1.11. Stack Operation Function

••••••	•••••••••	•••••••••••••••••••••••••••••••••
•••	•••••••••	••
•••	••••••••••••••••••	••••••••••••••••••••••••••••••••••

••

7.2. MK (MASTER-K) Function

•••••	••••••••••	••
•••	•••••••••	••
•••	•••••••••	••
•••	•••••••••	••
•••	••••	•••
•••	•••••••••	••
•••	••••••••	••
•••	••••••••	••

••

7.3. Array Operation Function

•••••	••••••••••	••••••••••••••••••••••••••••••••••
•••	•••••••••	••
•••	••••••••••••	••
•••	••••••••••••	•••
•••	•••••••••••••••••••	•••
•••	••••••••••••	••
•••	••••••••••••••••••	••
•••	••••••••••••••••••	•••
•••	••••••••••••	••
•••	•••••••••••••	••

••

7.4. Basic Function Block

7.4.1. Bistable Function Block

•••••	•••••••••••••••••	••
•••	•••	••
•••	•••	••
•••	•••••	••••••••••••••••••••••••••

7.4.2. Edge Detection Function Block

•••••	•••••••••••••••••	••
•••	•••••••	••
•••	•••••••	••

••

7.4.3. Counter

••••	••••••••••••••••••	•••••••••••••••••••••••••••••••••••••••
•••	••••	••••••••••••••••••••••••••
•••	••••	••••••••••••••••••••••••••••••
•••	•••••	•••••••••••••••••••••••••••••••••••••
•••	••••	••••••••••••••••••••••••••••••

7.4.4. Timer

••••	••••••••••••••••••	•••••••••••••••••••••••••••••••••••••••
•••	•••	•••••••••••••••••••••••••••
•••	••••	••••••••••••••••••••••••••••••••••
•••	••••	••••••••••••••••••••••••••••••••••••
•••	••••	••
•••	•••••••	•••••••••••••••••••••••••••••••••••
•••	•••••	•••
•••	••••••••	••••••••••••••••••••••••••••••••••••••
•••	•••••••••	•••
•••	••••••••••••••••••	••
••••	••••••••	•••
••••	•••••••••	•••

7.4.5. Other Function Block

••••	••••••••••••••••••	•••••••••••••••••••••••••••••••••••••••
•••	•••••	•••••••••••••••••••••••••••••••••••••
•••	••••••••••	••••••••••••••••••••••••••••••••••••••

••

8. Function/Function Block Library

8.1 Basic Function Library

This chapter describes the basic function library respectively.

POINT When a function error occurs, please refer to the following instruction.

- • • •Function error

  If an error occurs when a function is run, ENO will be 0 and, the error flag (_ERR, _LER) will be 1. Unless an error occurs, ENO will be equal to EN (EN and ENO are used in LD only).

- • • •Error flag

  _ERR (Error)

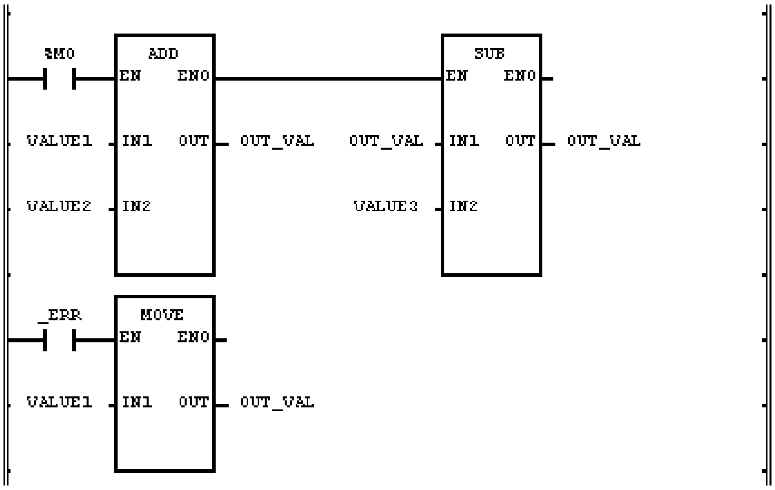
  - After function execution of which error is described, _ERR value will be changed as follows: (There's no change in _ERR value as long as there's no function error.)
  - In case of an operation error, it will be 1.
  - In other cases, it will be 0.

  _LER (Latched Error)

  - In case of an error after execution, _LER will be 1 and maintained until the end of the program.
  - It is possible to write 0 in the program.

■ **Program Example**

This is a program that moves VALUE1 data to OUT_VAL without executing SUB function if an ADD function error occurs.



(1) An error occurs in ADD function when its two inputs are as follows:

Input (IN1): VALUE1 (SINT) = 100 (16#64)

(IN2): VALUE2 (SINT) = 50 (16#32)

Output (OUT): OUT_VAL (SINT) = -106 (16#96)

(2) As an output value is out of range of its data type, the abnormal value will be stored in the OUT_VAL (SINT). At this time, ENO of ADD function will be 0 and SUB function will not be executed, and the error flag (_ERR and _LER) will be on.

(3) _ERR will be on and MOVE function will be executed.

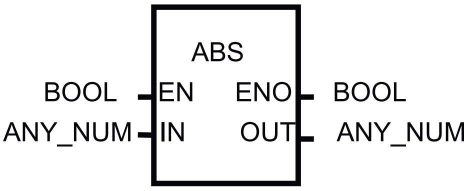
Input (IN1): VALUE1 (SINT) = 100 (16#64)

Output (OUT): OUT_VAL (SINT) = 100 (16#64)

ABS

Absolute value operation

·······	·····	······	······	·····	·····	·····	·····
···········	·······	····	· ·	·· ·	· ·	· ·	· ·

Function	Description
	**Input** EN: executes the function in case of 1 IN: input value of absolute value operation **Output** ENO: without an error, it will be 1 OUT: absolute value IN, OUT should be the same data type.

■ Function

It converts input IN into its absolute value and produces output OUT.

|X|, an absolute value of X is,

If X>=0, |X| = X,

If X<0, |X| = -X.

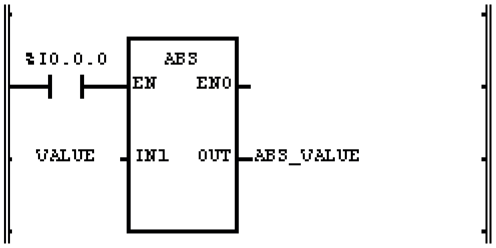
OUT = │IN│

■ Error

_ERR, _LER flags are set when input IN is a minimum value.

Ex) If IN value is –128 and its data type is SINT, an error occurs.

■ Program Example

LD	IL
	LD %I0.0.0 JMPN AL LD VALUE ABS ST ABS_VALUE AL :

(1) If the transition condition (%I0.0.0) is on, ABS function will be executed.

(2) If VALUE = -7, ABS_VALUE = │-7│ = 7.

If VALUE = 200, ABS_VALUE = │200│ = 200.

Input (IN): VALUE (INT) = -7

1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1

(16#FFF9)

(ABS) ↓

Output (OUT): ABS_VALUE (INT) = 7

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

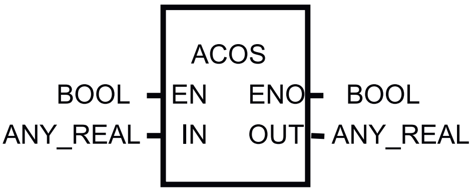
(16#0007)

The negative number of INT type is represented as the 2's compliment form (refer to 3.2.4. Data Type Structure)

ACOS• •

Arc Cosine operation

•••••••	••••	•••••••	•••••••	••••	••••	••••	••••	••	
•••••••••••	•••••	••••	• •	• ••	••	••	••	•	•

Function	Description
	**Input** EN: executes the function in case of 1 IN: input value of Arc Cosine operation **Output** ENO: without an error, it will be 1 OUT: Arc Cosine (radian) IN, OUT should be the same data type.

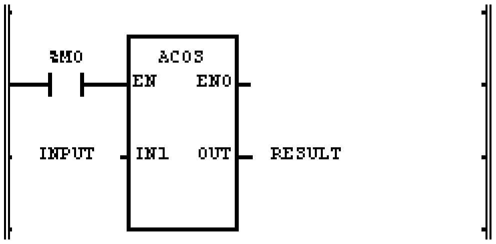
■ Function

It converts input IN into its Arc Cosine value and produces output OUT. The output range is between 0 and π.
OUT = ACOS (IN).

■ Error

Unless an IN value is between -1.0 and 1.0, _ERR, _LER flags are set.

■ Program Example

LD	IL
	LD %M0 JMPN LL LD INPUT ACOS ST RESULT LL :

(1) If the transition condition (%M0) is on, ACOS function will be executed.
(2) If INPUT is 0.8660... (√ 3 / 2), RESULT will be 0.5235... (π/6 rad = 30°).

ACOS ($\sqrt{3}$/ 2) = π/6
(COS π/6 = $\sqrt{3}$ / 2)

Input (IN1): INPUT (REAL) = 0.866

↓ (ACOS)

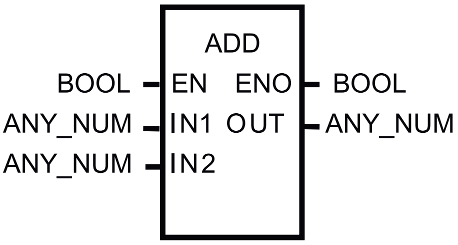
Output (OUT): RESULT (REAL) = 5.23499966E-01

REAL type representation is based on IEEE Standard 754-1984 (refer to 3.2.4. Data Type Structure).

ADD

Addition

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 ADD BOOL EN ENO BOOL ANY_NUM IN1 OUT ANY_NUM ANY_NUM IN2	**Input** EN: executes the function in case of 1 IN1: value to be added IN2: value to add Input variable number can be extended up to 8 **Output** ENO: without an error, it will be 1 OUT: added value IN1, IN2, ..., OUT should be the same data type.

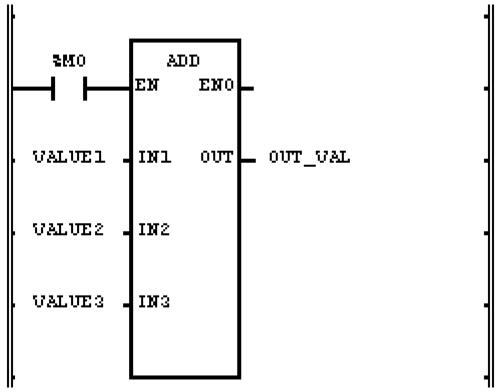
■ **Function**

It adds input variables up (IN1, IN2, ..., and INn, n: input number) and produces output OUT.
OUT = IN1 + IN2 + ... + INn

■ **Error**

When the output value is out of its data type, _ERR, _LER flags are set.

■ **Program Example**

LD	IL
 %M0 ADD EN ENO VALUE1 IN1 OUT OUT_VAL VALUE2 IN2 VALUE3 IN3	LD %M0 JMPN CA LD VALUE1 ADD IN1:= CURRENT RESULT IN2:= VALUE2 IN3:= VALUE3 ST OUT_VAL CA :

(1) If the transition condition (%M0) is on, ADD function will be executed.
(2) If input variable VALUE1 = 300, VALUE2 = 200, and VALUE3 = 100,
output variable OUT_VAL = 300 + 200 + 100 = 600.

Input (IN1): VALUE1 (INT) = 300 (16#012C)

0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0

+ (ADD)

(IN2): VALUE2 (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

+ (ADD)

(IN2): VALUE3 (INT) = 100 (16#0064)

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0

↓

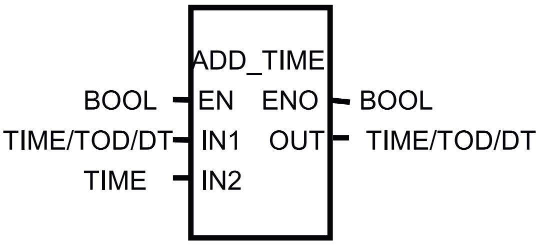
(OUT): OUT_VAL (INT) = 600 (16#0258)

0	0	0	0	0	0	1	0	0	1	0	1	1	0	0	0

ADD_TIME

Time Addition

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 ADD_TIME BOOL EN ENO BOOL TIME/TOD/DT IN1 OUT TIME/TOD/DT TIME IN2	**Input** EN: executes the function in case of 1 IN1: reference time, time of date IN2: time to add **Output** ENO: without an error, it will be 1 OUT: added result of TOD or time IN1, IN2, and OUT should be the same data type: If IN1 type is TIME_OF_DAY, OUT type will be also TIME_OF_DAY.

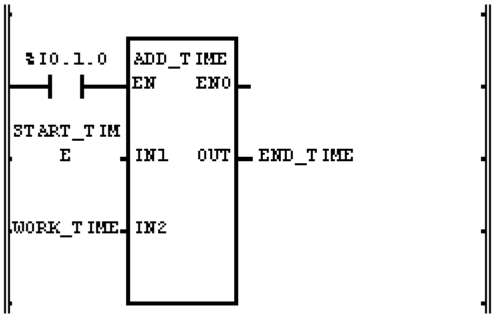
■ Function

- • If IN1 is TIME, added TIME will be an output.
- • If IN1 is TIME_OF_DAY, it adds TIME to reference TIME_OF_DAY and produces output TIME_OF_DAY.
- • If IN1 is DATE_AND_TIME, the output data type will be DT (Date and Time of Day) adding the time to the standard date and time of day.

■ Error

- • If an output value is out of range of related data type, _ERR, _LER flag will be set.
- • An error occurs: 1) when the result of adding the time and the time is out of range of TIME data type T#49D17H2M47S295MS; 2) the result of adding TOD (Time of Day) and the time exceeds 24hrs; 3) the result of adding the date and DT (Date and the Time of Day) exceeds the year, 2083.

■ Program Example

LD	IL
 %I0.1.0 ADD_TIME EN ENO START_TIM E IN1 OUT END_TIME WORK_TIME IN2	LD %I0.1.0 JMPN ABC LD START_TIME ADD_TIME IN1:= CURRENT RESULT IN2:= WORK_TIME ST END_TIME ABC :

(1) If the transition condition (%I0.1.0) is on, ADD_TIME function will be executed.

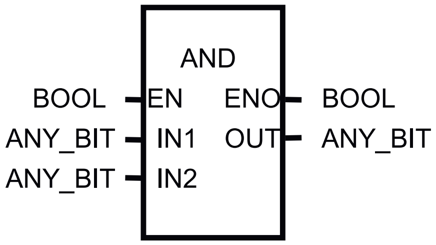
(2) If START_TIME is TOD#08:30:00 and WORK_TIME is T#2H10M20S500MS, END_TIME will be TOD#10:40:20.5.

Input (IN1): START_TIME (TOD) = TOD#08:30:00
+ (ADD_TIME)
(IN2): WORK_TIME (TIME) = T#2H10M20S500MS
↓
Output (OUT): END_TIME (TOD) = TOD#10:40:20.5

AND

Logical AND (Logical multiplication)

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

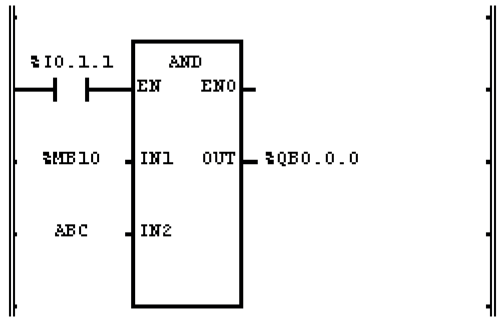
Function	Description
	**Input** EN: executes the function in case of 1 IN1: input 1 IN2: input 2 Input variables can be extended up to 8. **Output** ENO: without an error, it will be 1 OUT: AND result IN1, IN2, and OUT should be all the same data type.

■ **Function**

It performs logical AND operation on the input variables by bit and produces output OUT.

IN1 1111 0000
&
IN2 1010 1010
OUT 1010 0000

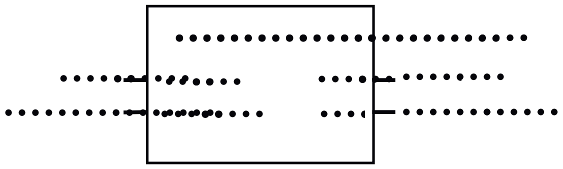
■ **Program Example**

LD	IL
	LD %I0.1.1 JMPN AA LD %MB10 AND IN1:= CURRENT RESULT IN2:= ABC ST %QB0.0.0 AA :

(1) If the transition condition (%I0.1.1) is on, AND function will be executed.
(2) If INI = %MB10 and IN2 = ABC, the result of AND will be shown in OUT (%QB0.0.0).

Input (IN1): %MB10 (BYTE) = 16#CC	1	1	0	0	1	1	0	0
	& (AND)							
(IN2): ABC (BYTE) = 16#F0	1	1	1	1	0	0	0	0
	↓							
Output (OUT): %QB0.0.0 (BYTE) = 16#C0	1	1	0	0	0	0	0	0

ARY_TO_STRING

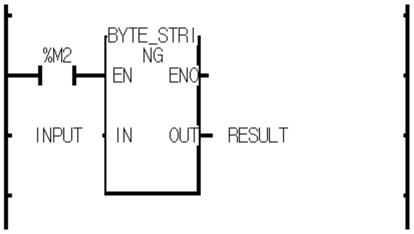
[illegible]	[illegible]
	**Input** EN: executes the function in case of 1 IN: byte array input **Output** ENO: without an error, it will be 1 OUT: string output

■ **Function**

It converts a byte array input into a string.

■ **Program Example**

LD



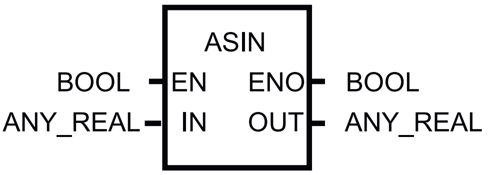
(1) If the transition condition (%M2) is on, BYTE_STRING function will be executed.

(2) Input variable INPUT is converted into string-type variable OUTPUT.

For example, if INPUT is 16#{22("), 47(G), 4D(M), 34(4), 2D(-), 43(C), 50(P), 55(U), 41(A), 22(")}, the RESULT will be "GM4-CPUA".

ASIN

Arc Sine operation

Function	Description
 ASIN BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: input value of Arc Sine operation **Output** ENO: without an error, it will be 1 OUT: radian output value after operation IN and OUT should be the same data type.

■ **Function**

It produces an output (Arc Sine value) of IN. The output value is between -π/2 and π/2.

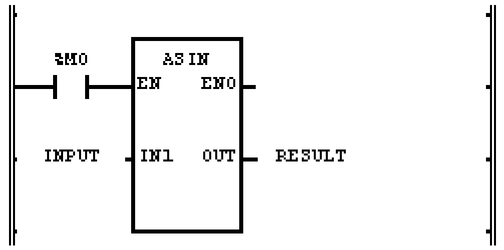
OUT = ASIN (IN)

■ **Error**

If an input value exceeds the range from -1.0 to 1.0, _ERR and _LER flags are set.

■ **Program Example**

LD



IL

```
LD          %M0
JMPN         AAA
LD          INPUT
ASIN
ST          RESULT
AAA :
```

(1) If the transition condition (%M0) is on, ASIN function will be executed.

(2) If INPUT variable is 0.8660.... ($\sqrt{3}$/2), the RESULT will be 1.0471.... (π/3 radian = 60°).

ASIN ($\sqrt{3}$ / 2) = π/3

Therefore, SIN (π/3) = $\sqrt{3}$ /2

Input (IN1): INPUT (REAL) = 0.866

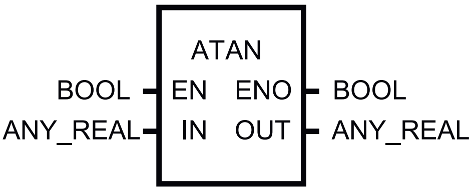
↓ (ASIN)

Output (OUT): RESULT (REAL) = 1.04714680E+00

ATAN

Arc Tangent operation

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

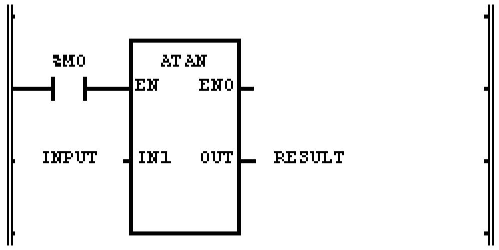
Function	Description
 ATAN BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: Input value of Arc Tangent operation **Output** ENO: without an error, it will be 1 OUT: radian output value after operation IN, OUT should be the same data type.

■ Function

It produces an output (Arc Tangent value) of IN value. The output value is between -π/2 and π/2.

OUT = ATAN (IN)

■ Program Example

LD	IL
	LD %M0 JMPN AA LD INPUT ATAN ST RESULT AA :

(1) If the transition condition (%M0) is on, ATAN function will be executed.

(2) If INPUT = 1.0, then output RESULT will be:

RESULT = π/4 = 0.7853...

ATAN (1) = π/4

(TAN (π/4) = 1)

Input (IN1): INPUT (REAL) = 1.0

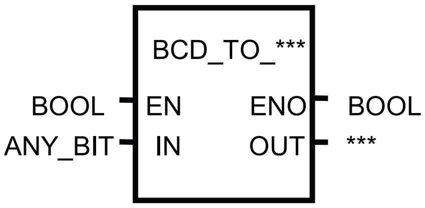
↓ (ATAN)

Output (OUT): RESULT (REAL) = 7.85398185E-01

BCD_TO_***

Converts BCD data into an integer number

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN: ANY_BIT (BCD) **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

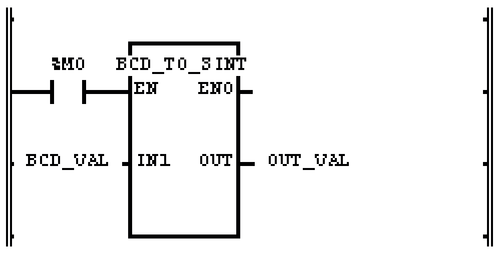
It converts input IN type and produces output OUT.

Function	Input type	Output type	Description
BCD_TO_SINT	BYTE	SINT	It converts BCD data into an output data type. It coverts only when the input date type is a BCD value. If an input data type is WORD, only the part of its data (0 • •16#9999) will be normally converted.
BCD_TO_INT	WORD	INT	
BCD_TO_DINT	DWORD	DINT	
BCD_TO_LINT	LWORD	LINT	
BCD_TO_USINT	BYTE	USINT	
BCD_TO_UINT	WORD	UINT	
BCD_TO_UDINT	DWORD	UDINT	
BCD_TO_ULINT	LWORD	ULINT	

■ Error

If IN is not a BCD data type, then the output will be 0 and _ERR, _LER flags will be set.

■ Program Example

LD	IL
	LD %M0 JMPN ABC LD BCD_VAL BCD_TO_SINT ST OUT_VAL ABC :

(1) If the transition condition (%M0) is on, BCD_TO_*** function will be executed.

(2) If BCD_VAL (BYTE) = 16#22 (2#0010_ 0010),
then the output variable OUT_VAL (SINT) = 22 (2#0001_ 0110).

Input (IN1): BCD_VAL (BYTE) = 16#22

0	0	1	0	0	0	1	0

(BCD_TO_SINT)

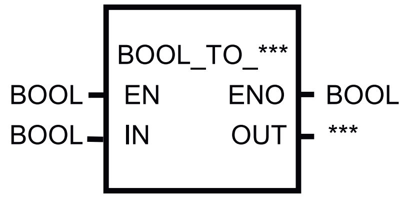
Output (OUT): OUT_VAL (SINT) = 22

0	0	0	1	0	1	1	0

BOOL_TO_***

BOOL type conversion

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

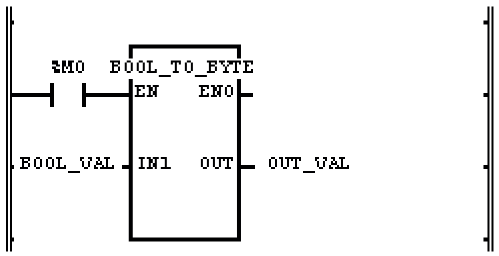
Function	Description
 BOOL_TO_*** BOOL BOOL EN IN ENO OUT BOOL ***	**Input** EN: executes the function in case of 1 IN: bit to convert (1 bit) **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ **Function**

It converts input IN type and produces output OUT.

Function	Output type	Description
BOOL_TO_SINT	SINT	If the input value (BOOL) is 2#0, it produces the integer number '0' and if it is 2#1, it does the integer number '1' according to the output data type.
BOOL_TO_INT	INT	
BOOL_TO_DINT	DINT	
BOOL_TO_LINT	LINT	
BOOL_TO_USINT	USINT	
BOOL_TO_UINT	UINT	
BOOL_TO_UDINT	UDINT	
BOOL_TO_ULINT	ULINT	
BOOL_TO_BYTE	BYTE	It converts BOOL into the output data type of which upper bits are filled with 0.
BOOL_TO_WORD	WORD	
BOOL_TO_DWORD	DWORD	
BOOL_TO_LWORD	LWORD	
BOOL_TO_STRING	STRING	It converts BOOL into a STRING type, which will be '0' or '1'.

■ **Program Example**

LD	IL
 %M0 BOOL_TO_BYTE EN ENO BOOL_VAL IN1 OUT OUT_VAL	LD %M0 JMPN ABC LD BOOL_VAL BOOL_TO_BYTE ST OUT_VAL ABC :

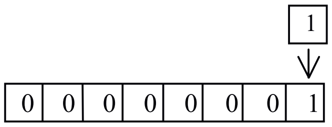
(1) If the transition condition (%M0) is on, BOOL_TO_*** function will be executed.

(2) If input BOOL_VAL (BOOL) = 2#1, then output OUT_VAL (BYTE) = 2#0000_ 0001.

Input (IN1): BOOL_VAL (BOOL) = 2#1

(BOOL_TO_SINT)

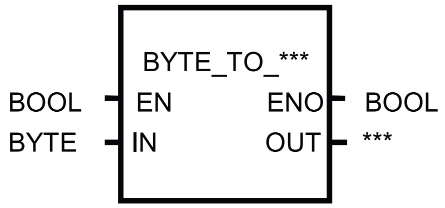
Output (OUT): OUT_VAL (BYTE) = 16#1



BYTE_TO_***

BYTE type conversion

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

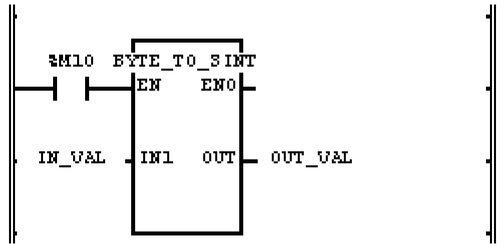
Function	Description
 BYTE_TO_*** BOOL EN ENO BOOL BYTE IN OUT ***	**Input** EN: executes the function in case of 1 IN: bit string to convert (8 bits) **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

It converts input IN type and produces output OUT.

Function	Output type	Description
BYTE _TO_SINT	SINT	Converts into SINT type without changing its internal bit array.
BYTE _TO_INT	INT	Converts into INT type filling the upper bits with 0.
BYTE _TO_DINT	DINT	Converts into DINT type filling the upper bits with 0.
BYTE _TO_LINT	LINT	Converts into LINT type filling the upper bits with 0.
BYTE _TO_USINT	USINT	Converts into USINT type without changing its internal bit array.
BYTE _TO_UINT	UINT	Converts into UINT type filling the upper bits with 0.
BYTE _TO_UDNT	UDINT	Converts into UDINT type filling the upper bits with 0.
BYTE _TO_ULINT	ULINT	Converts into ULINT type filling the upper bits with 0.
BYTE _TO_BOOL	BOOL	Takes the lower 1 bit and converts it into BOOL type.
BYTE _TO_WORD	WORD	Converts into WORD type filling the upper bits with 0.
BYTE _TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
BYTE _TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
BYTE _TO_STRING	STRING	Converts the input value into STRING type.

■ Program Example

LD	IL
 %M10 BYTE_TO_SINT EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M10 JMPN LLL LD IN_VAL BYTE_TO_SINT ST OUT_VAL LLL :

(1) If the transition condition (%M10) is on, BYTE_TO_SINT function will be executed.

(2) If IN_VAL (BYTE) = 2#0001_1000, OUT_VAL (SINT) = 24 (2#0001_1000).

Input (IN1): IN_VAL (BYTE) = 16#18

0	0	0	1	1	0	0	0

↓ (BYTE_TO_SINT)

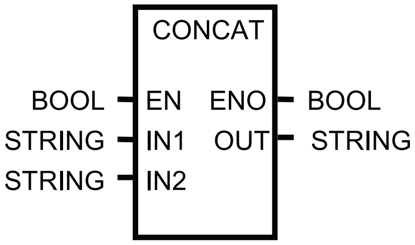
Output (OUT): OUT_VAL (SINT) = 24

0	0	0	1	1	0	0	0

CONCAT

Concatenates a character string

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN1: input character string IN2: input character string Input variable number can be extended up to 8. **Output** ENO: without an error, it will be 1. OUT: output character string

■ Function

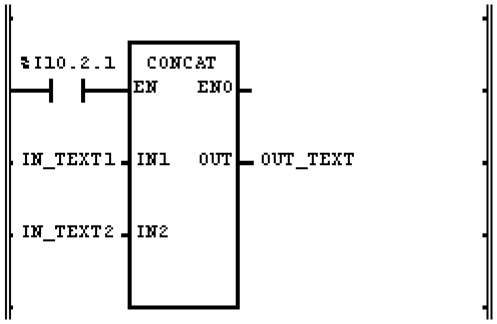
It concatenates the input character string IN1, IN2, IN3, …, INn (n: input number) in order and produces output character string OUT.

■ Error

If the sum of character number of each input character string is greater than 30, then the output CONCAT is the concatenate string of each input character string (up to 30 letters), and _ERR, _LER flags will be set.

■ Program Example

LD



IL

```
LD              %0.2.1
JMPN             THERE
LD              IN_TEXT1
CONCAT    IN1:=  CURRENT RESULT
          IN2:=  IN_TEXT2
ST               OUT_TEXT
THERE :
```

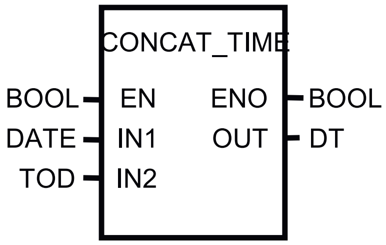
(1) If the transition condition (%I0.2.1) is on, CONCAT function will be executed.

(2) If input variable IN_TEXT1 = ‘ABCD’ and IN_TEXT2 = ‘DEF’, then OUT_TEXT = ‘ABCDDEF’.

Input (IN1): IN_TEXT1 (STRING) = ‘ABCD’
(IN2): IN_TEXT2 (STRING) = ‘DEF’
↓ (CONCAT)
Output (OUT): OUT_TEXT (STRING) = ‘ABCDDEF’

CONCAT_TIME

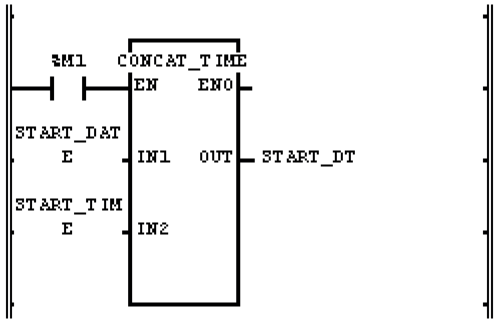
Concatenates date and time of day

Function	Description
	**Input** EN: executes the function in case of 1 IN1: date data input IN2: Time of day data input **Output** ENO: without an error, it will be 1. OUT: DT (Date and Time of Day) output

■ Function

It concatenates IN1 (date) and IN2 (time of day) and produces output OUT (DT).

■ Program Example

LD	IL
	LD %M1 JMPN AA LD START_DATE CONCAT_TIME IN1:= CURRENT RESULT IN2:= START_TIME ST START_DT AA :

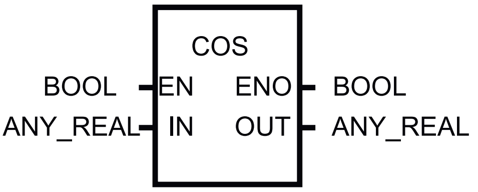
(1) If the transition condition (%M1) is on, CONCAT_TIME function will be executed.

(2) If START_DATE = D#1995-12-06 and START_TIME = TOD#08:30:00,
then, output START_DT = DT#1995-12-06-08:30:00.

Input (IN1): START_DATE1 (DATE) = D#1995-12-06
(CONCAT_TIME)
(IN2): START_TIME (TOD) = TOD#08:30:00
↓
Output (OUT): START_DT (DT) = DT#1995-12-06-08:30:00

COS

Cosine operation

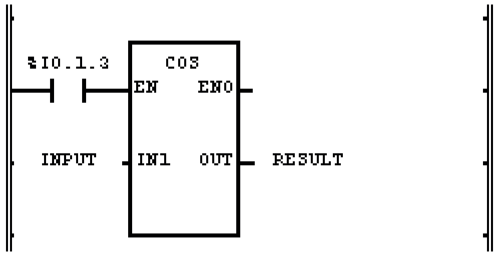
Function	Description
 COS BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: radian input value of Cosine operation **Output** ENO: without an error, it will be 1. OUT: result value of Cosine operation IN and OUT should be the same data type.

■ Function

It produces IN's Cosine operation value.

OUT = COS (IN)

■ Program Example

LD	IL
 %I0.1.3 COS EN ENO INPUT IN1 OUT RESULT	LD %I0.1.3 JMPN CCC LD INPUT COS ST RESULT CCC :

(1) If the transition condition (%I0.1.3) is on, COS function will be executed.

(2) If input INPUT = 0.5235 (π/6 rad = 30°), output RESULT = 0.8660 ... ($\sqrt{3}/2$).

COS (π/6) = $\sqrt{3}/2$ = 0.866

Input (IN1): INPUT (REAL) = 0.5235

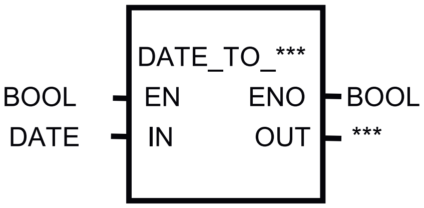
↓ (COS)

Output (OUT): RESULT (REAL) = 8.66074800E-01

DATE_TO_***

Date type conversion

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

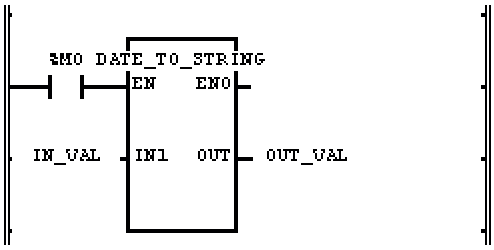
Function	Description
	**Input** EN: executes the function in case of 1 IN: date data to convert **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

It converts an input IN type and produces output OUT.

Function	Output type	Description
DATE_TO_UINT	UINT	Converts DATE into UINT type.
DATE_TO_WORD	WORD	Converts DATE into WORD type.
DATE_TO_STRING	STRING	Converts DATE into STRING type.

■ Program Example

LD	IL
	LD %M0 JMPN LL LD IN_VAL DATE_TO_STRING ST OUT_VAL LL :

(1) If the transition condition (%M0) is on, DATE_TO_STRING function will be executed.

(2) If IN_VAL (DATE) = D#1995-12-01, OUT_VAL (STRING) = D#1995-12-01.

Input (IN1): IN_VAL (DATE) = D#1995-12-01

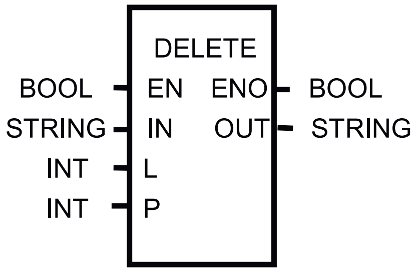
↓ (DATE_TO_STRING)

Output (OUT): OUT_VAL (STRING) = ‘D#1995-12-01’

DELETE

Deletes a character string

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 DELETE BOOL EN ENO BOOL STRING IN OUT STRING INT L INT P	**Input** EN: executes the function in case of 1 IN: input character string L: length of character string to delete P: position of character string to delete **Output** ENO: without an error, it will be 1. OUT: output character string

■ Function

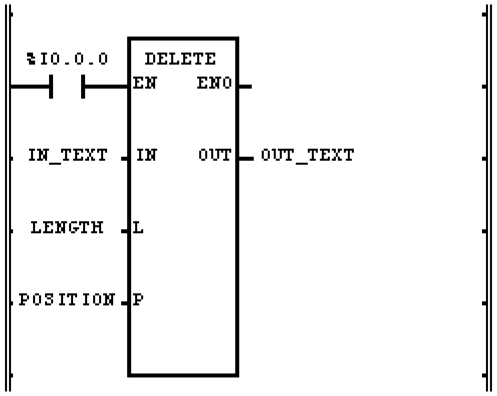
After deleting a character string (L) from the P character of IN, produces output OUT.

■ Error

If P≤ 0 or L< 0, or

If P > character number of IN, _ERR and _LER flags will be set.

■ Program Example

LD	IL
 %I0.0.0 DELETE EN ENO IN_TEXT IN OUT OUT_TEXT LENGTH L POSITION P	LD %I0.0.0 JMPN KKK LD IN_TEXT DELETE IN:= CURRENT RESULT L:= LENGTH P:= POSITION ST OUT_TEXT KKK :

(1) If the transition condition (%I0.0.0) is ON, DELETE function will be executed.

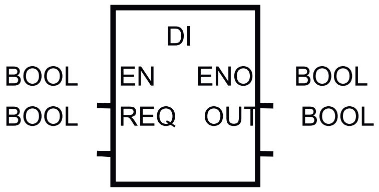
(2) If input variable IN_TEXT = ‘ABCDEF’, LENGTH = 3, and POSITION = 3, then OUT_TEXT (STRING) will be ‘ABF’.

Input (IN): IN_TEXT (STRING) = ‘ABCDEF’
(L): LENGTH (INT) = 3
(P): POSITION (INT) = 3
↓ (DELETE)
Output (OUT): OUT_VAL (STRING) = ‘ABF’

DI

Invalidates task program (Not to permit task program starting)

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

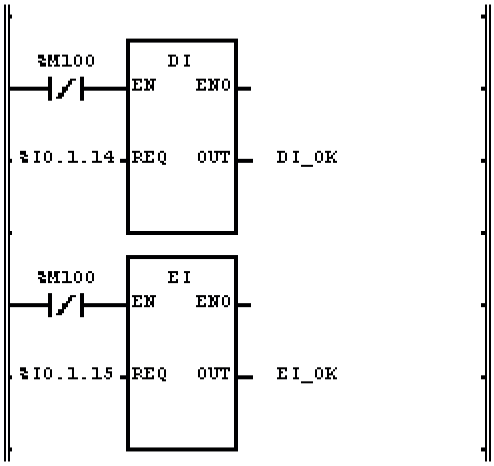
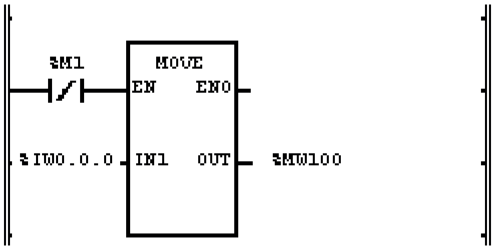
Function	Description
 DI BOOL BOOL EN REQ ENO OUT BOOL BOOL	**Input** EN: executes the function in case of 1 REQ: requires to invalidate task program starting **Output** ENO: without an error, it will be 1. OUT: If DI is executed, it will be 1.

■ **Function**

- •If EN = 1 and REQ = 1, it stops a task program (single, interval, interrupt).
- •Once DI function is executed, a task program does not start even if REQ input is 0.
- •In order to start a task program normally, please use 'EI' function.
- •If you want to partially stop the task program for the troubled part, (otherwise, miss the continuity of operation process due to the execution of other task program), it is available to use this function.
- •The task programs created while its execution is not invalidated will be executed according to task program types as follows:
  - Single task: it will be executed after 'EI' function or current-running task program execution. In his case, it repeats a task program as many as the state of single variable changes.
  - Interval task, interrupt: Interval task, interrupt: the task occurred when it is not permitted to execute will be executed after 'EI' function or the current-running task program execution. But, if it occurs more than 2 times, TASK_ERR is ON and TC_CNT (the number of task collision) is counted.

■ **Program Example**

This is the program that controls the task program increasing the value per second by using DI (Invalidates task program) and EI (permits running for task program).

LD	IL
(1) Scan program (TASK program control)  %M100 DI EN ENO %I0.1.14 REQ OUT DI_OK %M100 EI EN ENO %I0.1.15 REQ OUT EI_OK	(1) Scan program (TASK program control) LDN %M100 JMPN KK LD %I0.1.14 DI ST DI_OK KK : LDN %M100 JMPN LL LD %I0.1.15 EI ST EI_OK LL :
(2) Task program increasing by executing per second.  %M1 MOVE EN ENO %IW0.0.0 IN1 OUT %MW100	(2) Task program increasing by executing per second LDN %M1 JMPN MM LD %IW0.0.0 MOVE ST %MW100 MM :

(1) If REQ (assigned as direct variable %I0.1.14) of DI is on, DI function will be executed and output DI_OK will be 1.

(2) If DI function is executed, the task program to be executed per second stops.

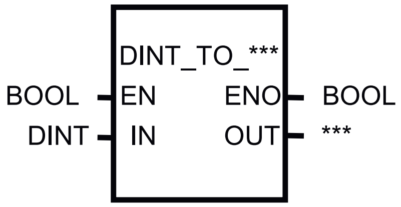
(3) If REQ (assigned as direct variable %I0.1.15) of EI is on, EI function will be executed and output EI_OK will be 1.

(4) If EI function is executed, the task program stopped due to function DI will restart.

DINT_TO_***

Invalidates task program (Not to permit task program starting)

•••••••	•••••	••••••••	••••••••	••••••	•••••	••••••	•••••••
••••••••••••	••••••	•••••	••	••	••	••	••

Function	Description
 DINT_TO_*** BOOL EN ENO BOOL DINT IN OUT ***	**Input** EN: executes the function in case of 1 IN: double integer value to convert **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

It converts Input IN type and produces output OUT.

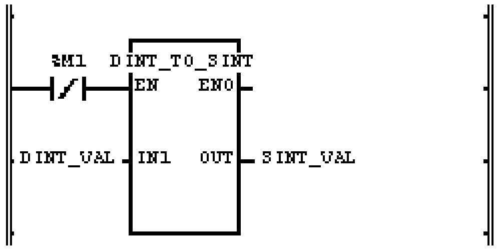
Function	Output type	Description
DINT_TO_SINT	SINT	If input is -128 • •127, normal conversion. Except this, an error occurs.
DINT_TO_INT	INT	If input is -32768 • •32767, normal conversion. Except this, an error occurs.
DINT_TO_LINT	LINT	Converts normally into LINT type.
DINT_TO_USINT	USINT	If input is 0 • •255, normal conversion. Except this, an error occurs.
DINT_TO_UINT	UINT	If input is 0 • •65535, normal conversion. Except this, an error occurs.
DINT_TO_UDINT	UDINT	If input is 0 • •2147483647, normal conversion. Except this, an error occurs.
DINT_TO_ULINT	ULINT	If input is 0 • •2147483647, normal conversion. Except this, an error occurs.
DINT_TO_BOOL	BOOL	Takes the low 1 bit and converts into BOOL type.
DINT_TO_BYTE	BYTE	Takes the low 8 bit and converts into BYTE type.
DINT_TO_WORD	WORD	Takes the low 18 bit and converts into WORD type.
DINT_TO_DWORD	DWORD	Converts into DWORD type without changing the internal bit array.
DINT_TO_LWORD	LWORD	Converts into LWORD type filling the upper bytes with 0.
DINT_TO_BCD	DWORD	If input is 0 • •99,999,999, normal conversion. Except this, an error occurs.
DINT_TO_REAL	REAL	Converts DINT into REAL type. During conversion, an error caused by the precision may occur.
DINT_TO_LREAL	LREAL	Converts DINT into LREAL type. During conversion, an error caused by the precision may occur.

■ Error

If a conversion error occurs, _ERR, _LER flags will be set.

When an error occurs, it takes as many lower bits as the bit number of the output type and produces an output without changing the internal bit array.

■ **Program Example**

LD	IL
 %M1 DINT_TO_SINT EN ENO DINT_VAL IN1 OUT SINT_VAL	LD %M1 JMPN LSB LD DINT_VAL DINT_TO_SINT ST SINT_VAL LSB :

(1) If the transition condition (%M1) is on, DINT_TO_SINT function will be executed.

(2) If INI = DINT_VAL (DINT) = -77, SINT_VAL (SINT) = -77.

Input (IN1): DINT_VAL (DINT) = -77

upper	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
lower	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1

(DINT_TO_SINT)

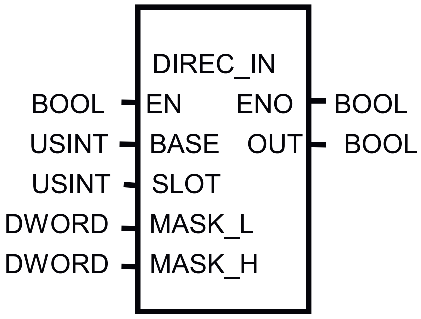
Output (OUT): OUT_VAL (SINT) = -77

1	0	1	1	0	0	1	1

DIREC_IN

Update input data

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

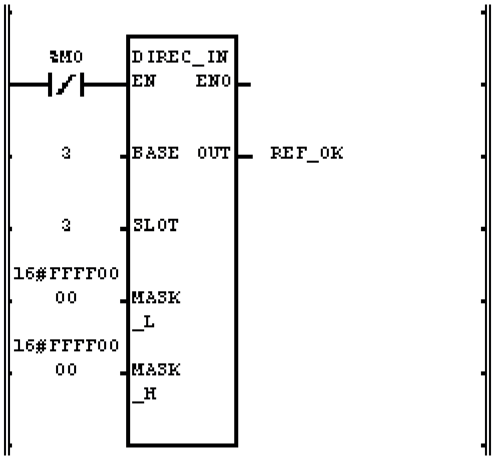
Function	Description
 DIREC_IN BOOL EN ENO BOOL USINT BASE OUT BOOL USINT SLOT DWORD MASK_L DWORD MASK_H	**Input** EN: executes the function in case of 1 BASE: base number of an input module installed SLOT: slot number of an input module installed MASK_L: designates bits not to be updated among lower 32-bit data of input MASK_H: designates bits not to be updated among upper 32-bit data of input **Output** ENO: without an error, it will be 1. OUT: if update is completed, output will be 1.

■ **Function**

- • If EN is 1 during the scan, DIREC_IN function reads 64-bit data of an input module from the designated position of BASE and SLOT and updates them.
- • At this time, only the actual contacts of an input module will be updated in the image scope.
- • DIREC_IN function is available to use when you want to change the ON/OFF state of input (%I) during the scan.
- • Generally, it's impossible to update input data during 1 scan (executing a scan program) because a scan-synchronized batch processing mode executes the batch processing to read input data and produce output data after a scan program. It's available to update related input data, if you use DIREC_IN function during program execution.

■ **Program Example**

1. This is the program that updates a 16-contact module installed in the 4th slot (slot number is 3) of the 3rd extension base of which input data are 2# 1010_1010_1110_1011.

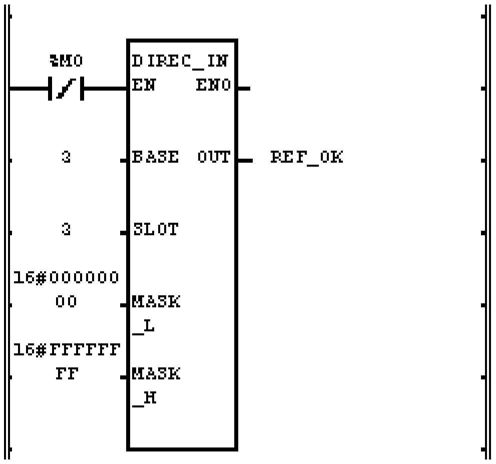
LD	IL
	LD %M0 JMPN ABC LD 3 DIREC_IN BASE:= CURRENT RESULT SLOT:= 3 MASK_L:= 16#FFFF0000 MASK_H:= 16#FFFF0000 ST REF_OK ABC :

(1) If the input condition (%M0) is on, function DIREC_IN will be executed.

(2) The image scope to update will be %IW3.3.0 and %IW3.3.0 will be updated with 2#1010_1010_1110_1011 during the scan because a 16-contact module is installed and the lower 16-bit data update is allowed (MASK_L = 16#FFFF0000).

(3) It doesn't matter what data are set in MASK_H because a 16-contact module is installed.

2. This is the program that updates the lower 16-bit data of the 32-contact module installed in the 4th slot (slot number is 3) of the 3rd extension base of which input data are 2#0000_0000_1111_1111_1100_1100_0011_0011.

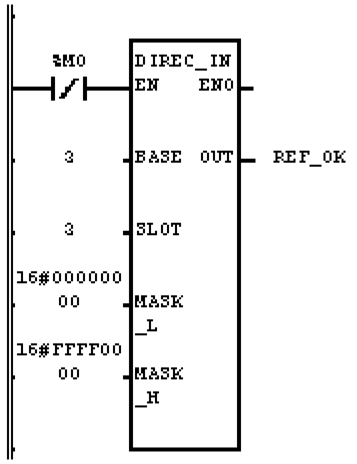
LD	IL
	LD %M0 JMPN ABC LD 3 DIREC_IN BASE:= CURRENT RESULT SLOT:= 3 MASK_L:= 16#FFFF0000 MASK_H:= 16#FFFFFFFF ST REF_OK ABC :

(1) If input condition (%M0) is on, function DIREC_IN will be executed.

(2) The image scope to update will be %ID3.3.0 but only %IW3.3.0 will be updated with 2#1100_1100_0011_0011 during the scan because a 16-contact module is installed and the lower 16-bit data update is allowed (MASK_L = 16#FFFF0000).

3. This is the program that updates the lower 48-bit data of the 64-contact module installed in the 4th slot (slot number is 3) of the 3rd extension base of which input data are 16#0000_FFFF_AAAA_7777 (2#0000_0000_0000_0000_1111_1111_1111_1111_1010 _1010_1010_1010_0111_0111_0111_0111).

LD



IL

```
LD                      %M0
JMPN                     ABC
LD                      3
DIREC_IN   BASE:=        CURRENT RESULT
           SLOT:=       3
           MASK_L:=     16#00000000
           MASK_H:=     16#FFFF0000
ST                       REF_OK
ABC :
```

(1) If the input condition (%M0) is on, function DIREC_IN will be executed.

(2) The installed module is a 64-contact module and the image scope to update will be %IL3.3.0 (%ID3.3.0 and ID3.3.1).

%ID3.3.0 will be updated because the lower 32-bit data update is allowed (MASK_L = 16#00000000).

%IW3.3.2 of %ID3.3.1 will be updated because only the lower 16-bit data update (among upper 32 bits) is allowed (MASK_H = 16#FFFF0000).

Accordingly, the data update of the image scope is as follows:

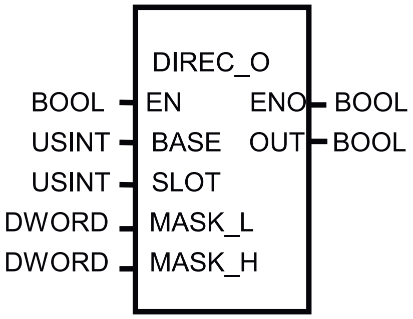
%IL3.3.0
- %ID3.3.0
  - %IW.3.3.0: 2#0111_0111_0111_0111
  - %IW.3.3.1: 2#1010_1010_1010_1010
- %ID3.3.1
  - %IW3.3.2: 2#1111_1111_1111_1111
  - %IW3.3.3: maintains the previous value

(3) If the input update is completed, output REF_OK will be 1.

DIREC_O

Update output data

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

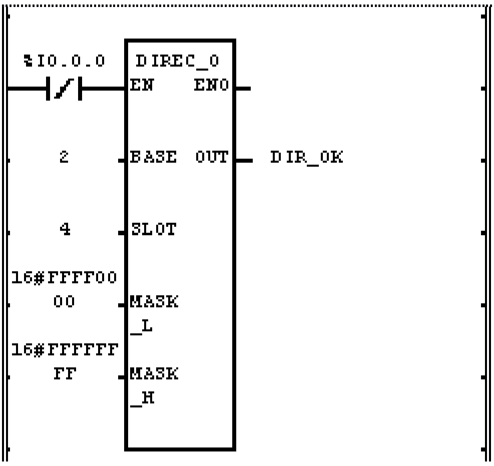
Function	Description
	**Input** EN: executes the function in case of 1 BASE: base number of an input module installed SLOT: slot number of an input module installed MASK_L: designates bits not to be updated among lower 32-bit data of output MASK_H: designates a bit not to update among upper 32-bit data of output **Output** ENO: without an error, it will be 1. OUT: If update is completed, output will be 1.

■ Function

- • If EN is 1 during the scan, DIREC_O function reads 64-bit data of an output module from the designated position of BASE and SLOT and updates the unmasked (MASK (0)) data.
- • DIREC_O is available to use when you want to change the ON/OFF state of output (%Q) during the scan.
- • Generally, it's impossible to update input data during 1 scan (executing a scan program) because a scan-synchronized batch processing mode executes the batch processing to read input data and produce output data after a scan program.
- • It's available to update related output data, if you use DIREC_O function during program execution.
- • If the base/slot number is wrong or it is not available to write data normally in an output module, ENO and OUT are '1' (without an error, it will be 1).

■ Program Example

1. This is the program that produces output data 2#0111_0111_0111_0111 in a 16-contact relay output module installed in the 5th slot (slot number is 4) of the 2nd extension base.

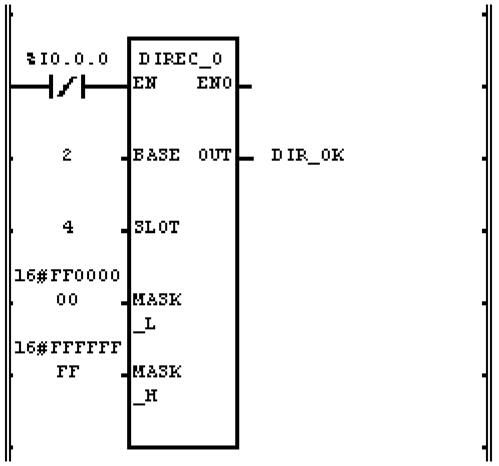
LD	IL
	LD %I0.0.0 JMPN AAA LD 2 DIREC_O BASE: = CURRENT RESULT SLOT: = 4 MASK_L: = 16#FFFF0000 MASK_H: = 16#FFFFFFFF ST REF_OK AAA :

(1) Input the slot and base number in which an output module installed.

(2) Set MASK_L as 16#FFFF0000 because the output data to produce are the lower 16 bits among the output contacts.

(3) If the transition condition (%I0.0.0) is on, DIREC_O will be executed and the data of the output module will be updated as 2#0111_0111_0111_0111 during the scan.

2. This is the program that updates the lower 24 bits of the 32-contact transistor output module, installed in the 5th slot (slot number is 4) of the 2nd extension base, with 2#1111_0000_1111_0000_1111_0000 during the scan.

LD	IL
 %I0.0.0 DIREC_O EN ENO 2 BASE OUT DIR_OK 4 SLOT 16#FF0000 00 MASK _L 16#FFFFFF FF MASK _H	LD %I0.0.0 JMPN AAA LD 2 DIREC_O BASE:= CURRENT RESULT SLOT:= 4 MASK_L:= 16#FF000000 MASK_H:= 16#FFFFFFFF ST REF_OK AAA:

(1) Input the slot and base number in which an output module installed.

(2) Set MASK_L as 16#FF000000 because the output data to produce are the lower 24 bits among the output contacts.

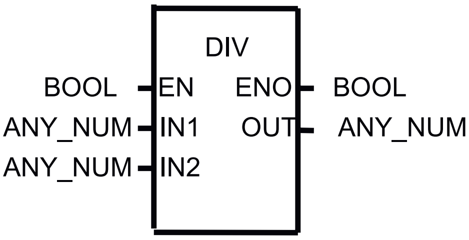
(3) If the transition condition (%I0.0.0) is off, function DIREC_O will be executed and the data of the output module will be updated as 2#• • • • _•••••••• •_1•1 1•1_•0000_1111_0000_1111_0000 during the scan.

Maintains the previous value.

DIV

Division

•••••••	•••••••••••	•••••••••	••••••••	••••••	••••••	•••••	••••••••
•••••••••••••	••••••	•••••	••	••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1 IN1: the value to be divided (dividend) IN2: the value to divide (divisor) **Output** ENO: without an error, it will be 1. OUT: the divided result (quotient) The variable connected to IN1, IN2 and OUT should be all the same data type.

■ Function

It divides IN1by IN2 and produces an output omitting decimal fraction from the quotient.

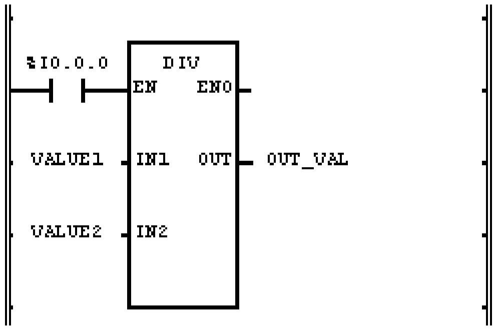
OUT = IN1/IN2

IN1	IN2	OUT	Remarks
7	2	3	Decimal fraction omitted.
7	-2	-3	
-7	2	-3	
-7	-2	3	
7	0	• •	Error

■ Error

If the value to divide (divisor) is '0', _ERR, _LER flags will be set.

■ Program Example

LD	IL
	LD %I0.0.0 JMPN LL LD VALUE1 DIV IN1:= CURRENT RESULT IN2:= VALUE2 ST OUT_VAL LL :

(1) If the transition condition (%I0.0.0) is on, DIV function will be executed.

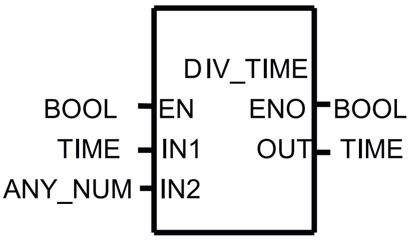
(2) If input VALUE1 = 300 and VALUE2 = 100, then output OUT_VAL = 300/100 = 3.

Input (IN1): VALUE1 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
/ (DIV)																
(IN2): VALUE2 (INT) = 100 (16#0064)	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
Output (OUT): OUT_VAL (INT) = 3 (16#3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

DIV_TIME

Time division

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN1: Time to divide IN2: The value to divide **Output** ENO: without an error, it will be 1. OUT: divided result time

■ Function

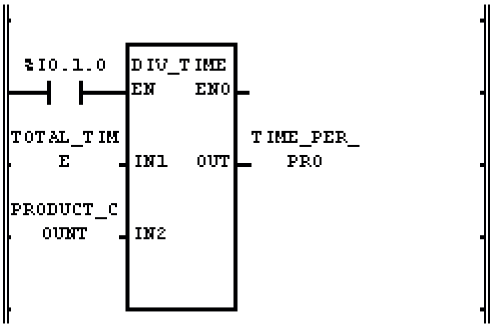
It divides IN1 (time) by IN2 (number) and produces output OUT (divided time).

■ Error

If a divisor (IN2) is 0, _ERR and _LER flags will be set.

■ Program Example

This is the program that calculates the time required to produce one product in some product line if the working time of day is 12hr 24min 24sec and product quantity of a day is 12 in a product line.

LD	IL
	LD %I0.1.0 JMPN SS LD TOTAL_TIME DIV_TIME IN1:= CURRENT RESULT IN2:= PRODUCT_COUNT ST TIME_PER_PRO SS :

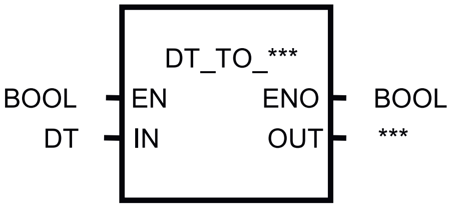
(1) If the transition condition (%I0.1.0) is on, DIV_TIME function will be executed.

(2) If it divides TOTAL_TIME (T#12H24M24S) by PRODUCT_COUNT (12), the time required to produce one product TIME_PER_PRO (T#1H2M2S) will be an output. That is, it takes 1hr 2min 2sec to produce one product.

Input (IN1): TOTAL_TIME (TIME) = T#12H24M24S
/ (DIV_TIME)
(IN2): PRODUCT_COUNT (INT) = 12
↓
Output (OUT): TIME_PER_PRO (TIME) = T#1H2M2S

DT_TO_***

DT type conversion

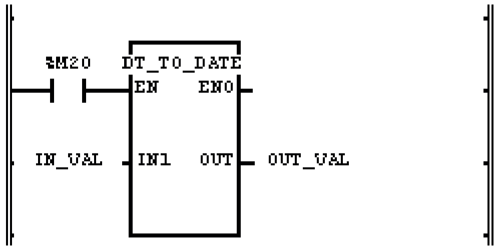
Function	Description
 DT_TO_*** BOOL DT EN IN ENO OUT BOOL ***	**Input** EN: executes the function in case of 1 IN: date and time of day data to convert **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ **Function**

It converts Input IN type and produces output OUT.

Function	Output type	Description
DT_TO_LWORD	LWORD	Converts DT into LWORD type. (The inverse conversion is available as there is no internal data change).
DT_TO_DATE	DATE	Converts DT into DATE type.
DT_TO_TOD	TOD	Converts DT into TOD type.
DT_TO_STRING	STRING	Converts DT into STRING type.

■ **Program Example**

LD	IL
 %M20 DT_TO_DATE EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M20 JMPN L LD IN_VAL DT_TO_DATE ST OUT_VAL L :

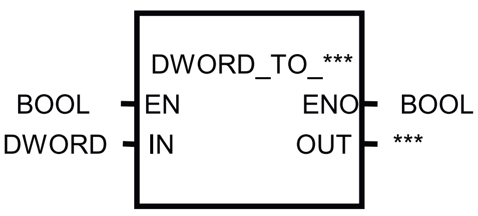
(1) If the transition condition (%M20) is on, DT_TO_DATE function will be executed.

(2) If input IN_VAL (DT) = DT#1995-12-01-12:00:00, output OUT_VAL (DATE) = D#1995-12-01.

Input (IN1): IN_VAL (DT) = DT#1995-12-01-12:00:00
↓ (DT_TO_DATE)
Output (OUT): OUT_VAL (DATE) = D#1995-12-01

DWORD_TO_***

DWORD type conversion

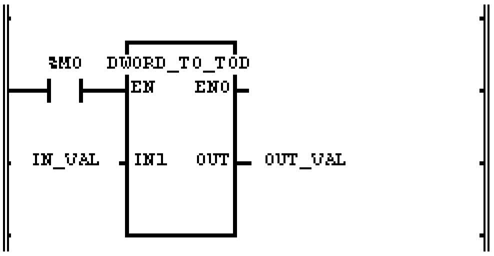
Function	Description
 DWORD_TO_*** BOOL EN ENO BOOL DWORD IN OUT ***	**Input** EN: executes the function in case of 1 IN: bit string to convert (32bit) **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

It converts Input IN type and produces output OUT.

Function	Output type	Description
DWORD _TO_SINT	SINT	Takes the lower 8 bits and converts into SINT type.
DWORD _TO_INT	INT	Takes the lower 16 bits and converts into INT type.
DWORD _TO_DINT	DINT	Converts into DINT type without changing the internal bit array.
DWORD _TO_LINT	LINT	Converts into LINT type filling the upper bits with 0
DWORD _TO_USINT	USINT	Takes the lower 8 bits and converts into USINT type.
DWORD _TO_UINT	UINT	Takes the lower 16 bits and converts into UINT type.
DWORD _TO_UDINT	UDINT	Converts into UDINT type without changing the internal bit array.
DWORD _TO_ULINT	ULINT	Converts into ULINT type filling the upper bits with 0.
DWORD _TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
DWORD _TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
DWORD _TO_WORD	WORD	Takes the lower 16 bits and converts into WORD type.
DWORD _TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
DWORD _TO_REAL	REAL	Converts into REAL type without changing the internal bit array.
DWORD _TO_TIME	TIME	Converts into TIME type without changing the internal bit array.
DWORD _TO_TOD	TOD	Converts into TOD type without changing the internal bit array.
DWORD _TO_STRING	STRING	Changes input value into decimal and converts into STRING type.

■ Program Example

LD	IL
 %M0 DWORD_TO_TOD EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN AA LD IN_VAL DWORD_TO_WORD ST OUT_VAL AA :

(1) If the transition condition (%M0) is on, DWIRD_TO_TOD function will be executed.
(2) If output IN_VAL (DWORD) = 16#3E8 (1000), output OUT_VAL (TOD) = TOD#1S.

Input (IN1): IN_VAL (DWORD) = 16#3E8(1000)

High	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0

↓ Converts a data type only without changing a data (internal bit array state)

Output (OUT): OUT_VAL(TOD) = TOD#1S

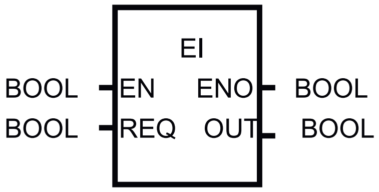
High	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0

Calculates TIME, TOD by converting decimal into MS unit. That is, 1000 is 1000ms = 1s.
Refer to 3.2.4. Data Type Structure.

EI

Permits running for task program

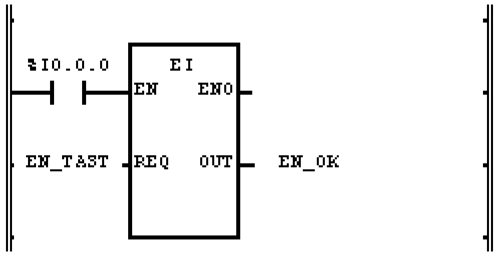
•••••••	•••••	•••••••	•••••••	••••••	•••••	••••••	•••••••
••••••••••••	••••••	•••••	••	••	••	••	••

Function	Description
 EI BOOL EN ENO BOOL BOOL REQ OUT BOOL	**Input** EN: executes the function in case of 1 REQ: requires to permit running for task program **Output** ENO: without an error, it will be 1. OUT: If EI is executed, an output will be 1.

■ Function

- • If EN is 1 and REQ input is 1, task program blocked by 'DI' function starts normally.
- • Once 'EI' command is executed, task program starts normally even if REQ input is 0.
- • Task programs created when they are not permitted to operate will be executed after 'EI' function or the current-running task program execution.

■ Program Example (refer to DI)

LD	IL
 %I0.0.0 EI EN ENO EN_TAST REQ OUT EN_OK	LD %I0.0.0 JMPN LSB LD EN_TASK EI ST EN_OK LSB :

If EN_TASK is 1, a task program starts normally.

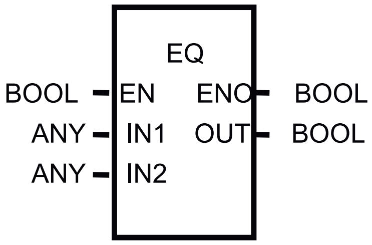
If EI function permits running for a task program, output EN_OK will be 1.

.

EQ

'Equal to' comparison

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

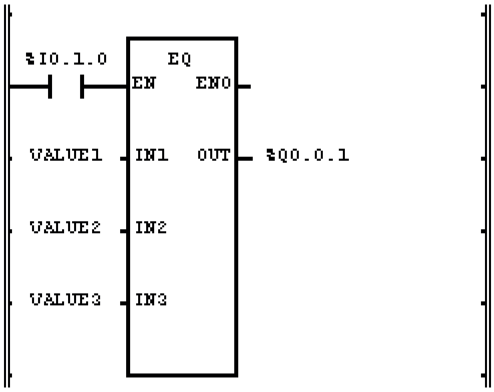
Function	Description
	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: The value to compare Input variable number can be extended up to 8. IN1, IN2, ... should be the same type. **Output** ENO: without an error, it will be 1. OUT: comparison result value

■ Function

If IN1 = IN2 = IN3 ... = INn (n : input number), output OUT will be 1.

In other cases, OUT will be 0.

■ Program Example

LD	IL
	LD %I0.1.0 JMPN AA LD VALUE1 EQ IN1:= CURRENT RESULT IN2:= VALUE2 IN3:= VALUE3 ST %Q0.0.1 AA :

(1) If the transition condition (%I0.1.0) is on, EQ function will be executed.

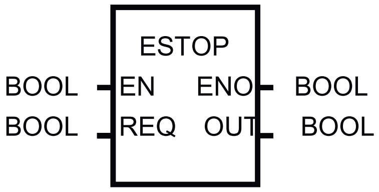
(2) If VALUE1 = 300, VALUE2 = 300, VALUE3 = 300 (comparison result VALUE1 = VALUE2 = VALUE3), output %Q0.0.1 = 1.

Input (IN1): VALUE1 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
	= (EQ)															
(IN2): VALUE2 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
	= (EQ)															
(IN3): VALUE1 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
																↓
Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1)																1

ESTOP

Emergency running stop by program

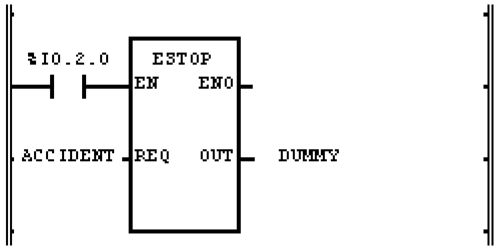
•••••••	••••••••••••	••••••••	•••••	••••	••••	•••	•••••••
••••••••••••	•••••••	••••••	• ••	•• ••	• ••	• ••	• ••

Function	Description
 ESTOP BOOL EN ENO BOOL BOOL REQ OUT BOOL	**Input** EN: executes the function in case of 1 REQ: requires the emergency running stop **Output** ENO: without an error, it will be 1. OUT: If ESTOP is executed, an output will be 1.

■ Function

- • If transition condition EN is 1 and the signal to require the emergency running stop by program REQ is 1, program operation stops immediately and returns to STOP mode.
- • In case that a program stops by 'ESTOP' function, it does not start despite of power re-supply.
- • If operation mode moves from STOP to RUN, it restarts.
- • If 'ESTOP' function is executed, the running program stops during operation; if it is not a cold restart mode, an error may occur when restarts.

■ Program Example

LD	IL
 %I0.2.0 ESTOP EN ENO ACCIDENT REQ OUT DUMMY	LD %I0.2.0 JMPN SSS LD ACCIDENT ESTOP (ST DUMMY) SSS :

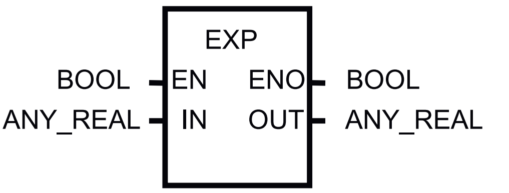
(1) If the transition condition (%I0.2.0) is on, ESTOP function will be executed.

(2) If ACCIDENT = 1, the running program stops immediately and returns to STOP mode.
In case of emergency, it is available to use it as a double safety device with mechanical interrupt.

EXP

Natural exponential operation

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

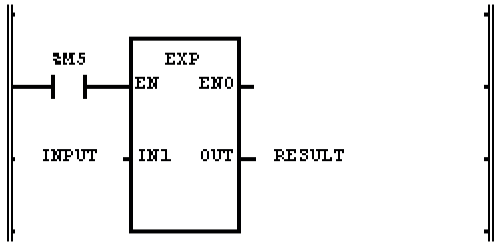
Function	Description
	**Input** EN: executes the function in case of 1 IN: input value of exponent operation **Output** ENO: without an error, it will be 1. OUT: result value IN, OUT should be the same data type.

■ Function

It calculates the natural exponent with exponent IN and produces output OUT.

OUT = e^{IN}

■ Program Example

LD	IL
	LD %M5 JMPN JJ LD INPUT EXP ST RESULT JJ :

(1) If the transition condition (%M5) is on, EXP function will be executed.

(2) If INPUT is 2.0, RESULT will be 7.3890…

$e^{2.0}$ = 7.3890.....

Input (IN1): INPUT (REAL) = 2.0

High	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0

(16#40000000)

↓ (EXP)

Output (OUT): RESULT (REAL) = 7.38905621E+00

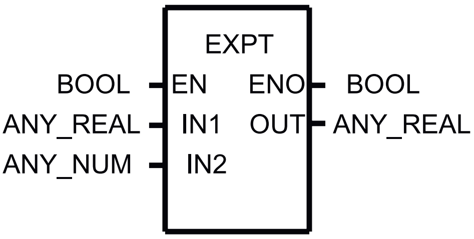
High	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0

(16#40EC7326)

EXPT

Exponential operation

•••••••	•••••	•••••••	••••••	••••••	•••••	•••••	••••••••
•••••••••••••	••••••	•••••	••	•••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1 IN1: real number IN2: exponent **Output** ENO: without an error, it will be 1. OUT: result value IN1 and OUT should be the same data type.

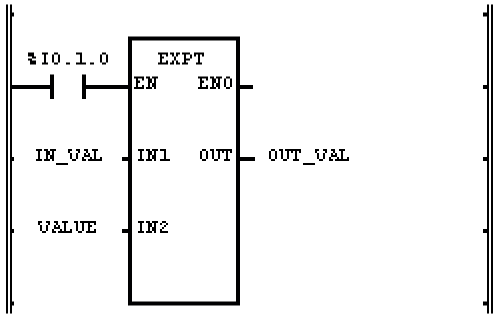
■ **Function**

It calculates IN1 with exponent IN2 and produces output OUT.
OUT = $IN1^{IN2}$

■ **Error**

If an output is out of range of related data type, _ERR and _LER flags will be set.

■ **Program Example**

LD	IL
	LD %I0.1.0 JMPN LSB LD IN_VAL EXPT IN1:= CURRENT RESULT IN2:= VALUE ST OUT_VAL LSB :

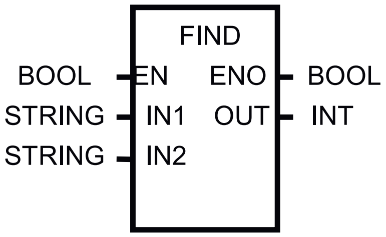
(1) If the transition condition (%I0.1.0) is on, ‘EXPT’ exponential function will be executed.
(2) If input IN_VAL = 1.5, VALUE = 3, output OUT_VAL = 1.5^3 = 1.5 • 1.5 • 1.5 = 3.375.

Input (IN1): IN_VAL (REAL) = 1.5
(IN2): VALUE (INT) = 3
↓ (EXPT)
Output (OUT): OUT_VAL (REAL) = 3.37500000E+00

FIND

Finds a character string

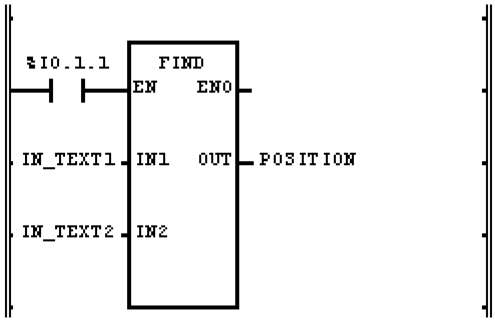
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN1: input character string IN2: character string to find **Output** ENO: without an error, it will be 1. OUT: location of character string to be found

■ Function

It finds the location of character string IN2 from input character string IN1. If the location is found, it shows a position of a first character of character string IN2 from character string IN1. Otherwise, output will be 0.

■ Program Example

LD	IL
	LD %I0.1.1 JMPM XYZ LD IN_TEXT1 FIND IN1:= CURRENT RESULT IN2:= IN_TEXT2 ST POSITION XYZ :

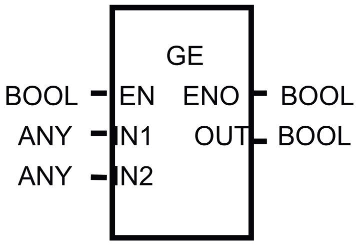
(1) If the transition condition (%I0.1.1) is on, FIND function will be executed.
(2) If input character string IN_TEXT1=‘ABCEF’ and IN_TEXT2=‘BC’, then output variable POSITION = 2.
(3) The first location of IN_TEXT2 (‘BC’) from input character string IN_TEXT1 (‘ABCEF’) is 2nd.

Input (IN1): IN_TEXT1 (STRING) = ‘ABCEF’
(FIND)
(IN2): IN_TEXT2 (STRING) = ‘BC’
↓
Output (OUT): POSITION (INT) = 2

GE

'Greater than or equal to' comparison

•••••••	•••••	••••••••	••••••••	•••••	•••••	•••••	••••••••
•••••••••••••	••••••	•••••	••	••	••	••	••

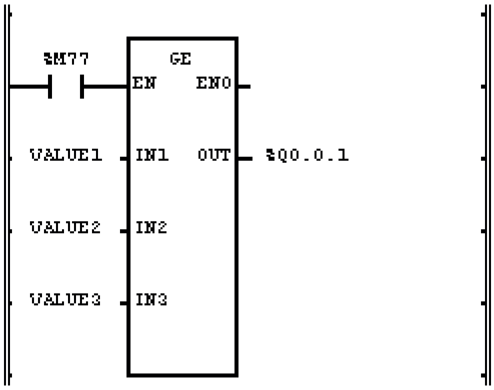
Function	Description
	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: the value to compare Input variable number can be extended up to 8. IN1, IN2, ... should be the same data type. **Output** ENO: without an error, it will be 1. OUT: comparison result value

■ Function

If IN1 ≥ IN2 ≥ IN3... ≥ INn (n: input number), an output will be 1.

Otherwise it will be 0.

■ Program Example

LD	IL
	LD %M77 JMPN YY LD VALUE1 GE IN1= CURRENT RESULT IN2= VALUE2 IN3= VALUE3 ST %Q0.0.1 YY:

(1) If the transition condition (%M77) is on, GE function will be executed.

(2) If input variable VALUE1 = 300, VALUE3 = 200, comparison result will be VALUE1 ≥ VALUE2 ≥ VALUE3. The output %Q0.01 = 1.

Input (IN1): VALUE1 (INT) = 300 (16#012C)

0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0

• •(GE)

(IN2): VALUE2 (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

• •(GE)

(IN3): VALUE3 (INT) = 100 (16#0064)

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0

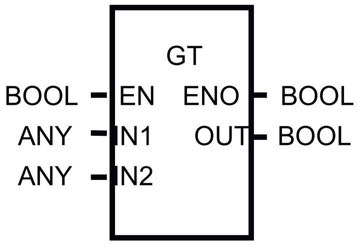
↓

Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1) 1

GT

'Greater than' comparison

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

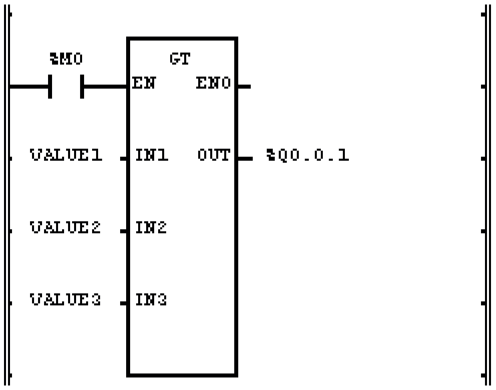
Function	Description
	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: the value to compare Input variable number can be extended up to 8. IN1, IN2, ... should be the same data type. **Output** ENO: without an error, it will be 1. OUT: comparison result value

■ Function

If IN1 > IN2 > IN3... > INn (n: input number), an output will be 1.
Otherwise it will be 0.

■ Program Example

LD



IL

```
LD                  %M0
JMPN                 AAA
LD                   VALUE1
GT          IN1:=    CURRENT RESULT
            IN2:=    VALUE2
            IN3:=    VALUE3
ST                    %Q0.0.1
AAA :
```

(1) If the transition condition (%M0) is on, GT function will be executed.

(2) If input variable VALUE1 = 300, VALUE2 = 200, and VALUE3 = 100, comparison result will be VALUE1 > VALUE2 > VALUE3. The output %Q0.0.1 = 1.

Input (IN1): VALUE1 (INT) = 300 (16#012C)

0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0

> (GT)

(IN2): VALUE2 (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

> (GT)

(IN3): VALUE3 (INT) = 100 (16#0064)

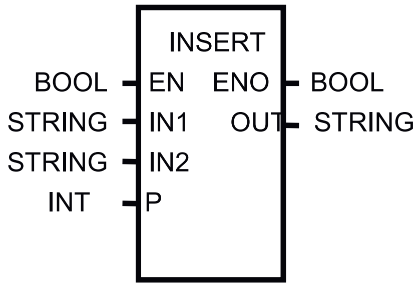
0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0

Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1) ↓ 1

INSERT

Inserts a character string

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 INSERT BOOL EN ENO BOOL STRING IN1 OUT STRING STRING IN2 INT P	**Input** EN: executes the function in case of 1 IN1: character string to be inserted IN2: character string to insert P: position to insert a character string **Output** ENO: without an error, it will be 1. OUT: output character string

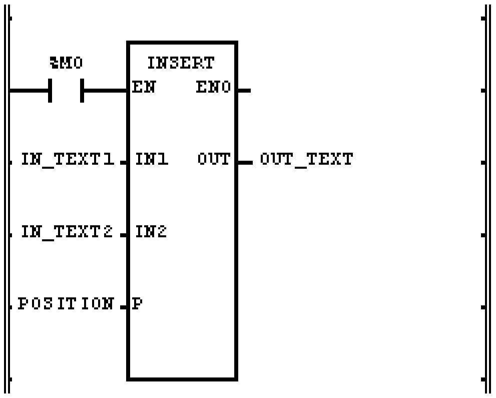
■ Function

It inserts character string IN2 after the P character of IN1 and produces output OUT.

■ Error

If P ≤ 0, 'character number of variable IN1' < P, or if the character number of result exceeds 30 (just 30 characters are produced), then _ERR, _LER flags will be set.

■ Program Example

LD	IL
 %M0 INSERT EN ENO IN_TEXT1 IN1 OUT OUT_TEXT IN_TEXT2 IN2 POSITION P	LD %M0 JMPN AA LD IN_TEXT1 INSERT IN1:= CURRENT RESULT IN2:= IN_TEXT2 P:= POSITION ST OUT_TEXT AA:

(1) If the transition condition (%M0) is on, INSERT function will be executed.

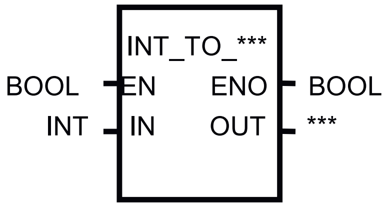
(2) If input variable IN_TEXT1 = 'ABCD', IN_TEXT2 = 'XY', and POSITON = 2,
output variable OUT_TEXT = 'ABXYCD'.

Input (IN1): IN_TEXT1 (STRING) = 'ABCD'
(IN2): IN_TEXT2 (STRING) = 'XY'
(P): POSITION (INT) = 2
↓ (FIND)
Output (OUT): OUT_TEXT = 'ABXYCD'

INT_TO_***

INT type conversion

•••••••	••••	••••	••••	•••	••••	••••	••••
•••••••••••	••••••	•••••	• •	• •	• •	• •	• •

Function	Description
 INT_TO_*** BOOL EN ENO BOOL INT IN OUT ***	**Input** EN: executes the function in case of 1 IN: integer value to convert **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ **Function**

It converts input IN type and produces output OUT.

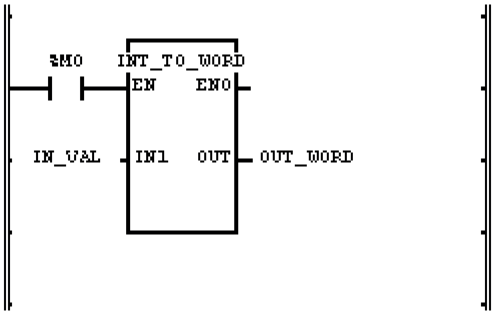
Function	Output type	Description
INT_TO_SINT	SINT	If input is -128 • • 127, normal conversion. Except this, an error occurs.
INT_TO_DINT	DINT	Converts into DINT type normally.
INT_TO_LINT	LINT	Converts into LINT type normally.
INT_TO_USINT	USINT	If input is 0 • • 255, normal conversion. Except this, an error occurs.
INT_TO_UINT	UINT	If input is 0 • • 32767, normal conversion. Except this, an error occurs.
INT_TO_UDINT	UDINT	If input is 0 • • 32767, normal conversion. Except this, an error occurs.
INT_TO_ULINT	ULINT	If input is 0 • • 32767, normal conversion. Except this, an error occurs.
INT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
INT_TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
INT_TO_WORD	WORD	Converts into WORD type without changing the internal bit array.
INT_TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
INT_TO_LWORD	LWORD	Converts into LWORD type filling the high bit with 0.
INT_TO_BCD	WORD	If input is 0~9,999, normal conversion. Except this, an error occurs.
INT_TO_REAL	REAL	Converts INT into REAL type normally.
INT_TO_LREAL	LREAL	Converts INT into LREAL type normally.

■ **Error**

If a conversion error occurs, _ERR _LER flags will be set.

If an error occurs, take as many lower bits as the bit number of the output type and produces an output without changing the internal bit array.

■ **Program Example**

LD	IL
 %M0 INT_TO_WORD EN ENO IN_VAL IN1 OUT OUT_WORD	LD %M0 JMPN AAA LD IN_VAL INT_TO_WORD ST OUT_WORD AAA:

(1) If the input condition (%M0) is on, INT_TO_WORD function will be executed.

(2) If input variable IN_VAL (INT) = 512 (16#200), output variable OUT_WORD (WORD) = 16#200.

Input (IN1): IN_VAL (INT) = 512 (16#200)	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

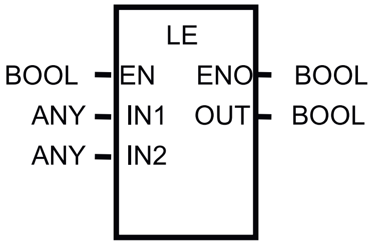
↓ (INT_TO_WORD)

Output (OUT): OUT_WORD (WORD) = 16#200	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

LE

'Less than or equal to' comparison

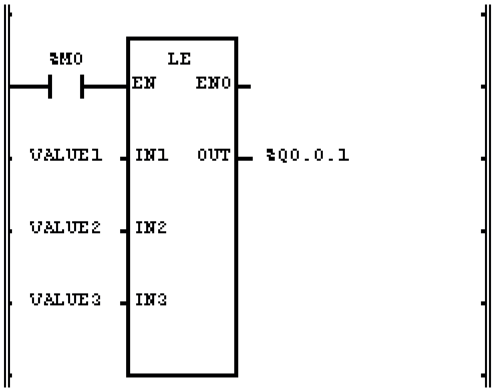
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 LE BOOL EN ENO BOOL ANY IN1 OUT BOOL ANY IN2	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: the value to compare Input variable number can be extended up to 8. IN1, IN2, ...should be the same data type. **Output** ENO: without an error, it will be 1. OUT: comparison result value

■ Function

If IN1 ≤ IN2 ≤ IN3... ≤ INn (n: input number), output OUT will be 1.
Otherwise it will be 0.

■ Program Example

LD	IL
 %M0 LE EN ENO VALUE1 IN1 OUT %Q0.0.1 VALUE2 IN2 VALUE3 IN3	LD %M0 JMPN BBB LD VALUE1 LE IN1:= CURRENT RESULT IN2:= VALUE2 IN3:= VALUE3 ST %Q0.0.1 BBB:

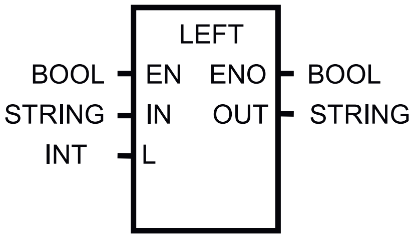
(1) If the transition condition (%M0) is on, LE function will be executed.
(2) If input variable VALUE1 = 150, VALUE2 = 200, and VALUE3 = 250, output %Q0.0.1 = 1
(VALUE1 ≤ VALUE2 ≤ VALUE3).

Input (IN1): VALUE1 (INT) = 150 (16#0096)	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0
									≤ (LE)							
(IN2): VALUE2 (INT) = 200 (16#00C8)	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0
									≤ (LE)							
(IN3): VALUE1 (INT) = 250 (16#0064)	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1)																1

LEFT

Takes the left side of a character string

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN: input character string L: length of character string **Output** ENO: without an error, it will be 1. OUT: output character string

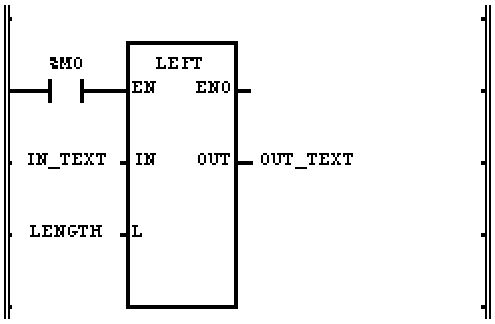
■ Function

It takes a left character string (L) of IN and produces output OUT.

■ Error

If L < 0, _ERR and _LER flags will be set.

■ Program Example

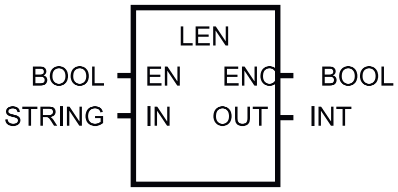
LD	IL
	LD %M0 JMPN FF LD IN_TEXT LEFT IN:= CURRENT RESULT L:= LENGTH ST OUT_TEXT FF:

(1) If the transition condition (%M0) is on, function LEFT function will be executed.

(2) If input variable IN_TEXT = 'ABCDEFG' and LENGTH = 3, output character string OUT_TEXT = 'ABC'.

```
Input (IN1): IN_TEXT (STRING) = 'ABCDEFG'
      (IN2): LENGTH (INT) = 3
                                ↓  (LEFT)
Output (OUT): OUT_TEXT (STRING) = 'ABC'
```

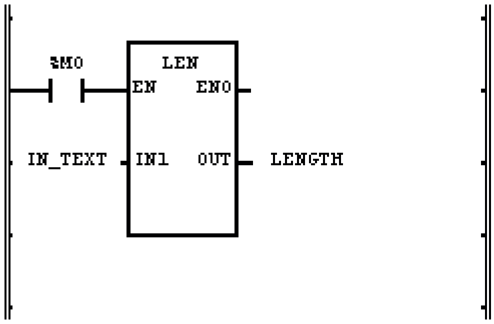
LEN

Function	Description
 LEN BOOL EN ENO BOOL STRING IN OUT INT	**Input** EN: executes the function in case of 1 IN: input character string **Output** ENO: without an error, it will be 1. OUT: the length of a character string

■ Function

It produces a length (character number) of the input character string (IN).

■ Program Example

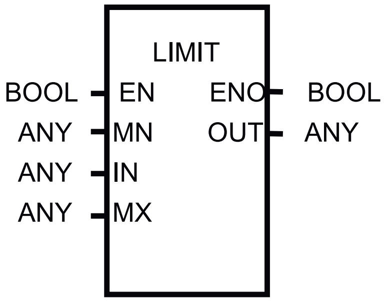
LD	IL
	LD %M0 JMPN II LD IN_TEXT LEN IN:= CURRENT RESULT ST LENGTH II:

(1) If the transition condition (%M0) is on, LEN function will be executed.

(2) If input variable IN_TEXT = ‘ABCD’, output variable LENGTH = 4.

Input (IN1): IN_TEXT (STRING) = ‘ABCD’
↓ (LEN)
Output (OUT): LENGTH (INT) = 4

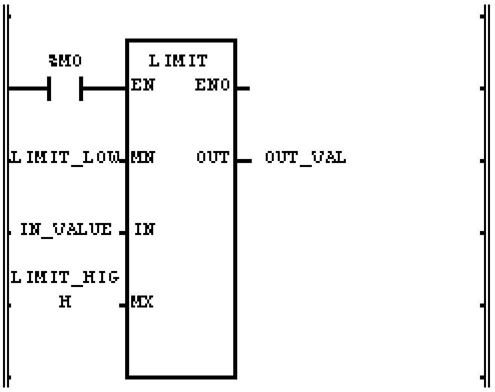
LIMIT

Function	Description
	**Input** EN: executes the function in case of 1 MN: minimum value IN: the value to be limited MX: maximum value **Output** ENO: without an error, it will be 1. OUT: value in the range MN, IN, MX, OUT should be the same data type.

■ Function

- • If input IN value is between MN and MX, the IN will be an output.
  That is, if MN ≤ IN ≤ MX, OUT = IN
- • If input IN value is less than MN, MN will be an output. That is, if IN < MN, OUT = MN.
- • If input IN value is greater than MX, MX will be an output. That is, if IN > MX, OUT = MX

■ Program Example

LD	IL
	LD %M0 JMPN MM LD LIMIT_LOW LIMIT MN:= CURRENT RESULT IN := IN_VALUE MX:= LIMIT_HIGH ST OUT_VAL MM:

```
LD              %M0
JMPN            MM
LD              LIMIT_LOW
LIMIT   MN:=    CURRENT RESULT
        IN :=   IN_VALUE
        MX:=    LIMIT_HIGH
ST              OUT_VAL
MM:
```

(1) If the transition condition (%M0) is on, LIMIT function will be executed.

(2) Output variable OUT_VAL for lower limit input LIMIT_LOW, upper limit input (LIMIT_HIGH) and limited value input IN_VALUE will be as follows:

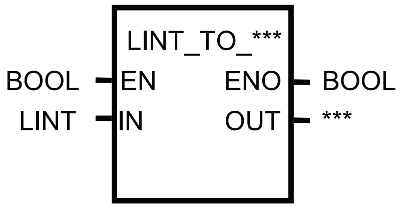
LIMIT_LOW	IN_VALUE	LIMIT_HIGH	OUT_VAL
1000	2000	3000	2000
1000	500	3000	1000
1000	4000	3000	3000

Input (MN): LIMIT_LOW (INT) = 1000
(IN): IN_VALUE (INT) = 4000
(MX): IN_VALUE (INT) = 3000
↓ (LIMIT)
Output (OUT): OUT_VAL (INT) = 3000

LINT_TO_***

•••••••••••••••••••••••••••••••••••••

•••••••	••••	•••••	•••••	••••	••••	••••	••••
•••••••••••	••••••	••••	• •	• ••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1 IN: long integer value to convert **Output** ENO: without an error, it will be 1. OUT: type converted data

■ **Function**

It converts input IN type and produces output OUT.

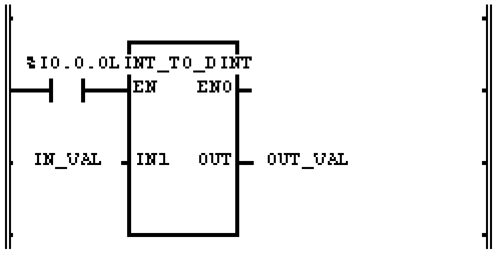
Function	Output type	Description
LINT_TO_SINT	SINT	If input is -128 • •127, normal conversion. Otherwise an error occurs.
LINT_TO_INT	INT	If input is –32,768• •32,767, normal conversion. Otherwise an error occurs.
LINT_TO_DINT	DINT	If input is -2^{31} • •2^{31}-1, normal conversion. Otherwise an error occurs.
LINT_TO_USINT	USINT	If input is 0• •255, normal conversion. Otherwise an error occurs.
LINT_TO_UINT	UINT	If input is 0• •65,535, normal conversion. Otherwise an error occurs.
LINT_TO_UDINT	UDINT	If input is 0 • •2^{32}-1, normal conversion. Otherwise an error occurs.
LINT_TO_ULINT	ULINT	If input is 0 • •2^{63}-1, normal conversion. Otherwise an error occurs.
LINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
LINT_TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
LINT_TO_WORD	WORD	Takes the lower 16 bits and converts into WORD type.
LINT_TO_DWORD	DWORD	Takes the lower 32 bits and converts into DWORD type.
LINT_TO_LWORD	LWORD	Converts into LWORD type without changing the internal bit array.
LINT_TO_BCD	LWORD	If input is 0~9,999,999,999,999,999, normal conversion. Otherwise an error occurs.
LINT_TO_REAL	REAL	Converts LINT into REAL type. During the conversion, an error caused by the precision may occur.
LINT_TO_LREAL	LREAL	Converts LINT into LREAL type. During the conversion, an error caused by the precision may occur.

■ **Error**

If a conversion error occurs, _ERR and _LER flags will be set.

If an error occurs, take as many lower bits as the bit number of the output type and produces an output without changing the Internal bit array.

■ **Program Example**

LD	IL
 %I0.0.0 LINT_TO_DINT EN ENO IN_VAL IN1 OUT OUT_VAL	LD %I0.0.0 JMPN AAA LD IN_VAL LINT_TO_DINT ST OUT_VAL AAA:

(1) If the input condition (%I0.0.0) is on, LINT_TO_DINT function will be executed.

(2) If input variable IN_VAL (LINT) = 123_456_789, output variable OUT_VAL (DINT) = 123_456_789.

Input (IN1): IN_VAL (LINT) = 123,456,789
(16#75BCD15)

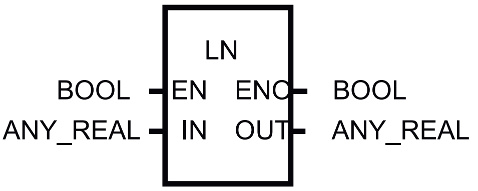
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	1	0	1	1	0	1	1
1	1	0	0	1	1	0	1	0	0	0	1	0	1	0	1

↓ (LINT_TO_DINT)

Output (OUT): OUT_VAL (DINT) = 123,456,789
(16#75BCD15)

0	0	0	0	0	1	1	1	0	1	0	1	1	0	1	1
1	1	0	0	1	1	0	1	0	0	0	1	0	1	0	1

LN

Function	Description
	**Input** EN: executes the function in case of 1 IN: input value of natural logarithm operation **Output** ENO: without an error, it will be 1. OUT: natural logarithm value IN, OUT should be the same data type.

■ Function

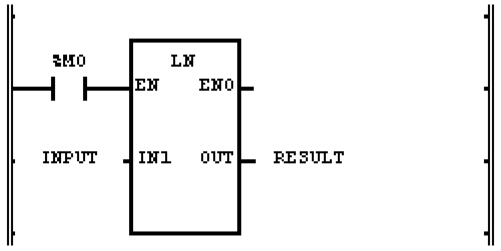
It finds a natural logarithm value of IN and produces output OUT.

OUT = ln IN

■ Error

If an input is 0 or a negative number, _ERR and _LER flags will be set.

■ Program Example

LD	IL
	LD %M0 JMPN AE LD INPUT LN ST RESULT AE:

(1) If the transition condition (%M0) is on, LN function will be executed.

(2) If input variable INPUT is 2.0, output variable RESULT will be 0.6931 ....

ln (2.0) = 0.6931...

Input (IN1): INPUT (REAL) = 2.0

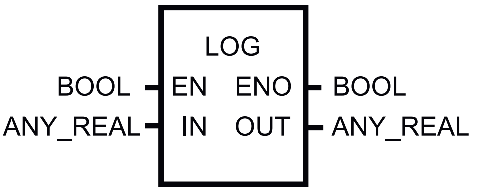
↓ (LN)

Output (OUT): RESULT (REAL) = 6.93147182E-01

LOG

Base 10 Logarithm operation

•••••••	••••	••••••	••••••	•••••	•••••	••••••	•••••••	•	•
•••••••••••	•••••••	•••••	• ••	• ••	••	••	••		

Function	Description
 LOG BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: input value of common logarithm operation **Output** END: without an error, it will be 1. OUT: the value of common logarithm operation IN, OUT should be the same data type.

■ Function

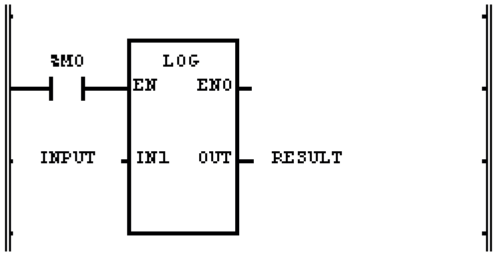
It finds the value of Base 10 Logarithm of IN and produces output OUT.

OUT = log10 IN = log IN

■ Error

If input value IN is 0 or a negative number, _ ERR and _LER flags will be set.

■ Program Example

LD	IL
	LD %M0 JMPN BB LD INPUT LOG ST RESULT BB:

(1) If the transition condition (%M0) is on, LOG function will be executed.

(2) If input variable INPUT is 2.0, output variable RESULT will be 0.3010 .....

•••••••••••••••••••••••••••••••••••••

Input (IN1): INPUT (REAL) = 2.0

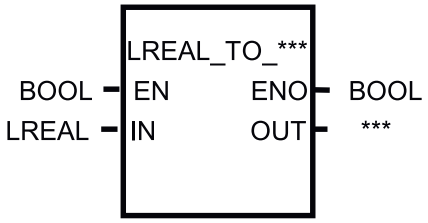
↓ (LOG)

Output (OUT): RESULT (REAL) = 3.01030010E-01

LREAL_TO_***

LREAL type conversion

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN: LREAL value to convert **Output** ENO: without an error, it will be 1. OUT: type converted data

■ Function

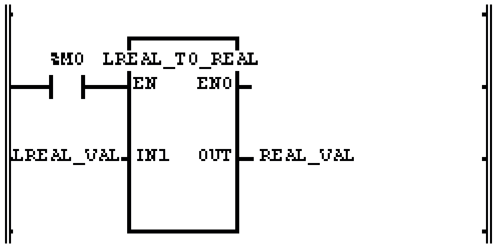
It converts input IN type and produces output OUT.

Function	Output type	Description
LREAL_TO_SINT	SINT	If integer number of input is -128 • •127, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_INT	INT	If integer number of input is -32768 • •32767, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_DINT	DINT	If integer number of input is -2^{31} • •2^{31}-1, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_LINT	LINT	If integer number of input is -2^{63} • •2^{63}-1, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_USINT	USINT	If integer number of input is 0 • •255, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_UINT	UINT	If integer number of input is 0 • •65,535, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_UDINT	UDINT	If integer number of input is 0 • •2^{32}-1, normal conversion. Otherwise an error occurs (decimal round off).
LREAL_TO_ULINT	ULINT	If integer number of input is 0 • •2^{64}-1, normal conversion. Otherwise an error occurs (decimal round-off).
LREAL_TO_LWORD	LWORD	Converts into LWORD type without changing the internal bit array.
LREAL_TO_REAL	REAL	Converts LREAL into REAL type normally. During the conversion, an error caused by the precision may occur.

■ Error

If an overflow occurs because an input value is greater than the value available for the output type, _ERR and _LER flags will be set. If an error occurs, an output will be 0.

■ **Program Example**

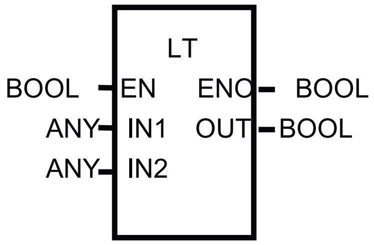
LD	**IL**
 %M0 LREAL_TO_REAL EN ENO LREAL_VAL IN1 OUT REAL_VAL	LD LREAL_VAL LREAL_TO_REAL ST REAL_VAL

(1) If the input condition (%M0) is on, LREAL_TO_REAL function will be executed.
(2) If input variable LREAL_VAL (LREAL) = -1.34E-12, output variable REAL_VAL (REAL)= -1.34E-12.

Input (IN1): LREAL_VAL (LREAL) = -1.34E-12
↓ (LREAL_TO_REAL)
Output (OUT): REAL_VAL (REAL) = -1.34E-12

LT

'Less than' comparison

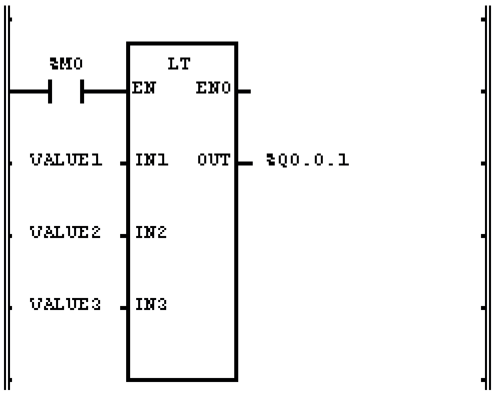
Function	Description
 LT BOOL EN ENO BOOL ANY IN1 OUT BOOL ANY IN2	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: the value to compare Input variable number can be extended up to 8. IN1, IN2, ...should be the same data type. **Output** ENO: without an error, it will be 1. OUT: comparison result value

■ Function

If IN1 < IN2 < IN3... < INn (n: input number), output value OUT will be 1.
Otherwise output OUT will be 0.

■ Program Example

LD



IL

```
LD              %M0
JMPN            AA
LD               VALUE1
LT        IN1:=   CURRENT RESULT
          IN2:=   VALUE2
          IN3:=   VALUE3
ST               %Q0.0.1
AA:
```

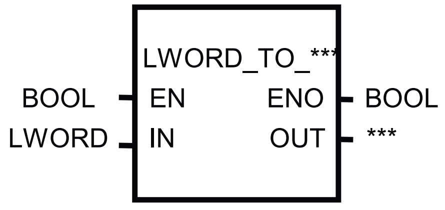
(1) If the transition condition (%M0) is on, LT function will be executed.
(2) If input variable VALUE1 = 100, VALUE2 = 200, and VALUE3 = 300, output %Q0.0.1 = 1.

Input (IN1): VALUE1 (INT) = 100 (16#0064)	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
< (LT)																
(IN2): VALUE2 (INT) = 200 (16#00C8)	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0
< (LT)																
(IN3): VALUE3 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	0
Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1)																1

LWORD_TO_***

LWORD type conversion

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

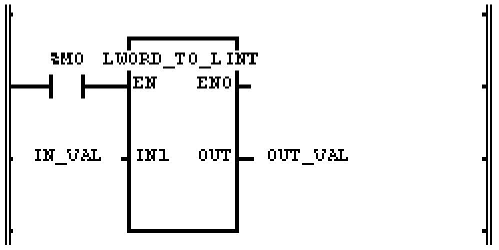
Function	Description
 LWORD_TO_*** BOOL EN ENO BOOL LWORD IN OUT ***	**Input** EN: executes the function in case of 1 IN: bit string to convert (64bit) **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

It converts input IN type and produces output OUT.

Function	Output type	Description
LWORD _TO_SINT	SINT	Takes the lower 8 bits and converts into SINT type.
LWORD _TO_INT	INT	Takes the lower 16bits and converts into INT type.
LWORD _TO_DINT	DINT	Takes the lower 32bits and converts into DINT type.
LWORD _TO_LINT	LINT	Converts into LINT type without changing the internal bit array.
LWORD _TO_USINT	USINT	Takes the lower 8 bits and converts into USINT type.
LWORD _TO_UINT	UINT	Takes the lower 16 bits and converts into UINT type.
LWORD _TO_UDINT	UDINT	Takes the lower 32bits and converts into UDINT type.
LWORD _TO_ULINT	ULINT	Converts into ULINT type without changing the internal bit array.
LWORD _TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
LWORD _TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
LWORD _TO_WORD	WORD	Takes the lower 16 bits and converts into WORD type.
LWORD _TO_DWORD	DWORD	Takes the lower 32 bits and converts into DWORD type.
LWORD _TO_LREAL	LREAL	Converts LWORD into LREAL type.
LWORD _TO_DT	DT	Converts into DT type without changing the internal bit array.
LWORD _TO_STRING	STRING	Converts input value into STRING type.

■ **Program Example**

LD	IL
 %M0 LWORD_TO_LINT EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN PPP LD IN_VAL LWORD_TO_LINT ST OUT_VAL PPP:

(1) If the input condition (%M0) is on, LWORD_TO_LINT function will be executed.

(2) If input variable IN_VAL (LWORD) = 16#FFFFFFFFFFFFFFFF, output variable OUT_VAL (LINT) will be -1 (16#FFFFFFFFFFFFFFFF).

Input (IN1): IN_VAL (LWORD) = 16#FFFFFFFFFFFFFFFF

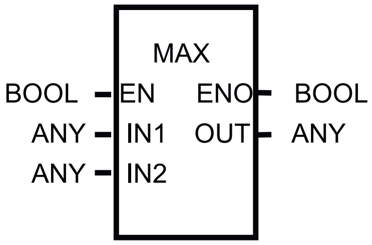
⇓ (LWORD_TO_LINT)

Output (OUT): OUT_VAL (LINT) = -1

MAX

Maximum value

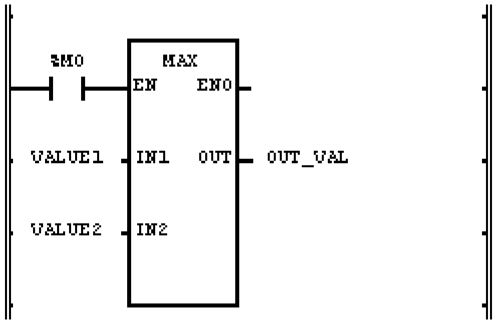
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1 IN1: the value to be compared IN2: the value to compare Input variable number can be extended up to 8. **Output** ENO: without an error, it will be 1. OUT: maximum value among input IN1, IN2,…, OUT should be the same data type.

■ Function

It produces the maximum value among input IN1, IN2,..., INn (n: input number).

■ Program Example

LD	IL
	LD %M0 JMPN GG LD VALUE1 MAX IN1:= CURRENT RESULT IN2:= VALUE2 ST OUT_VALUE GG:

(1) If the transition condition (%M0) is on, MAX function will be executed.

(2) As the result of comparing input variable (VALUE1 = 100 and VALUE2 = 200), maximum value is 200. Output OUT_VAL will be 200.

Input (IN1): VALUE1 (INT) = 100 (16#0064)

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0

(MAX)

(IN2): VALUE2 (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

↓

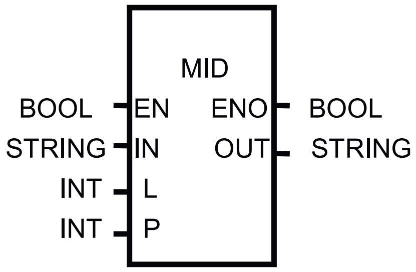
Output (OUT): OUT_VAL (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

MID

Takes the middle part of a character string

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
 MID BOOL EN ENO BOOL STRING IN OUT STRING INT L INT P	**Input** EN: executes the function in case of 1 IN: input character string L: the length of character string to output P: starting location of character string to output **Output** ENO: without an error, it will be 1. OUT: output character string

■ **Function**

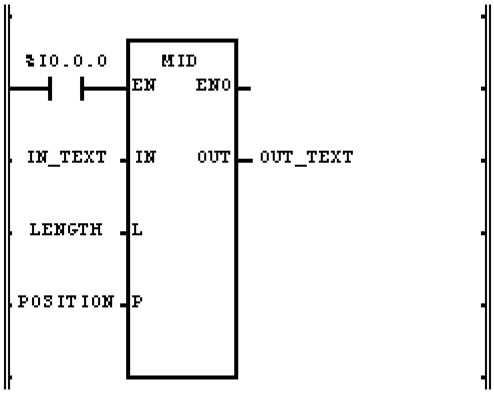
It produces a character string (L) of IN from the P character.

■ **Error**

If (character number of variable IN) < P, P <= 0 or L < 0, then _ERR and _LER flags will be set.

■ **Program Example**

LD	IL

 %I0.0.0
MID
EN
ENO
IN_TEXT
IN
OUT
OUT_TEXT
LENGTH
L
POSITION
P
	```
	LD          %I0.0.0
	JMPN        MM
	LD          IN_TEXT
	MID   IN:=  CURRENT RESULT
	      L: =  LENGTH
	      P: =  POSITION
	ST          OUT_TEXT
	MM:
	```

(1) If the transition condition (%I0.0.0) is on, MID function will be executed.

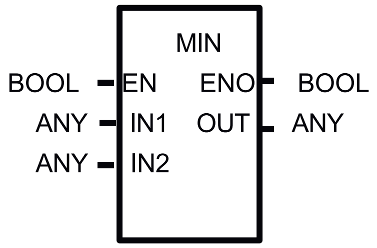
(2) If input character string IN_TEXT = ‘ABCDEFG’, the length of character string LENGTH = 3, and starting location of character starting POSITION = 2, output variable OUT_TEXT = ‘BCD’.

Input (IN): IN_TEXT1 (STRING) = ‘ABCDEFG’
(L): LENGTH (INT) = 3
(P): POSITION (INT) = 2
$\Downarrow$ (MID)
Output (OUT): OUT_TEXT = ‘BCD’

MIN

Minimum value

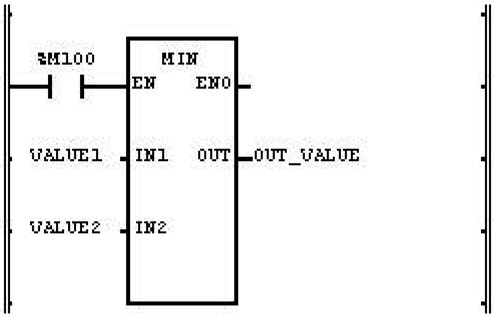
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	• ••	• ••	• ••	• ••

Function	Description
	**Input** EN: executes the function in case of 1 IN1: value to be compared IN2: value to compare Input variable number can be extended up to 8 **Output** ENO: without an error, it will be 1 OUT: minimum value among input values IN1, IN2, ..., OUT should be all the same data type.

■ **Function**

Produces the minimum value among input IN1, IN2, ... , INn (n: input number).

■ **Program Example**

LD	IL
	LD %M100 JMPN BBB LD VALUE1 MIN IN1:= CURRENT RESULT IN2:= VALUE2 ST OUT_VALUE BBB:

(1) If the transition condition (%M100) is ON, MIN function is executed.

(2) The output is OUT_VALUE = 100 because its minimum value is 100 as the result of comparing VALUE1 = 100 to VALUE2 = 200.

Input (IN1): VALUE1 (INT) = 100 (16#0064)

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	(

(MIN)

(IN2): VALUE2 (INT) = 200 (16#00C8)

0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	(

↓

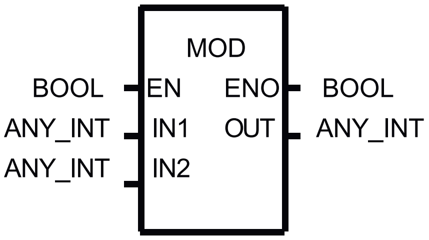
Output (OUT): OUT_VAL (INT) = 100 (16#0064)

0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	(

MOD

Dividing result (remainder)

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 MOD BOOL EN ENO BOOL ANY_INT IN1 OUT ANY_INT ANY_INT IN2	**Input** EN: executes the function in case of 1 IN1: dividend IN2: divisor **Output** ENO: without an error, it will be 1 OUT: dividing result (remainder) IN1, IN2, ..., OUT should be all the same data type.

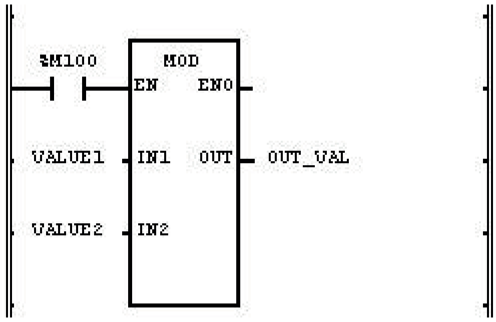
■ **Function**

Divides IN1 by IN2 and outputs its remainder as OUT.

OUT = IN1 - (IN1/IN2) • IN2 (if IN2 = 0, OUT = 0)

IN1	IN2	OUT
7	2	1
7	-2	1
-7	2	-1
-7	-2	-1
7	0	0

■ **Program Example**

LD	IL
 %M100 MOD EN ENO VALUE1 IN1 OUT OUT_VAL VALUE2 IN2	LD %M100 JMPN BB LD VALUE1 MOD IN1:= CURRENT RESULT IN2:= VALUE2 ST OUT_VAL BB:

(1) If the transition condition (%M100) is ON, MOD function is executed.

(2) If the dividend VALUE1 = 37 and the divisor VALUE2 = 10, the remainder value OUT_VAL is 7 as a result of dividing 37 by 10.

Input (IN1): VALUE1 (INT) = 37 (16#0025)

0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1

(MOD)

(IN2): VALUE2 (INT) = 10 (16#000A)

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	(

∨

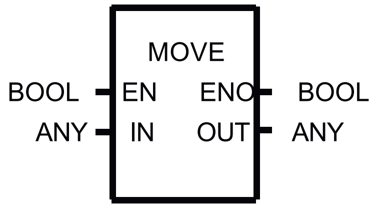
Output (OUT): OUT_VAL (INT) = 7 (16#0007)

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

MOVE

Data movement (Copy data)

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

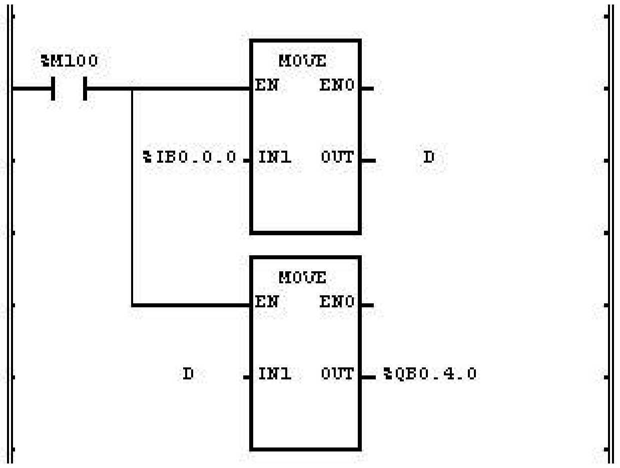
Function	Description
 MOVE BOOL EN ENO BOOL ANY IN OUT ANY	**Input** EN: executes the function in case of 1 IN: value to be moved **Output** ENO: without an error, it will be 1 OUT: moved value Variables connected to IN and OUT are the same type.

■ **Function**

Moves an IN value to OUT.

■ **Program Example**

This is a program that transfers the 8-contact inputs %I0.0.0• %I0.0.7 to the variable D and then moves them to output %Q0.4.0• %Q0.4.7.

LD	IL
 %M100 MOVE EN ENO %IB0.0.0 IN1 OUT D MOVE EN ENO D IN1 OUT %QB0.4.0	LD %M100 JMPN AAA LD %IB0.0.0 MOVE ST D LD D MOVE ST %QB0.4.0 AAA:

(1) If the transition condition (%M100) is ON, MOVE function is executed.

(2) It moves 8-contact input module data to the variable D by the first MOVE function and moves them to %Q0.4.0• %Q0.4.7.

Input (IN1): %IB0.0.0 (BYTE) = 16#18 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

(MOVE)

D (BYTE) = 16#18 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

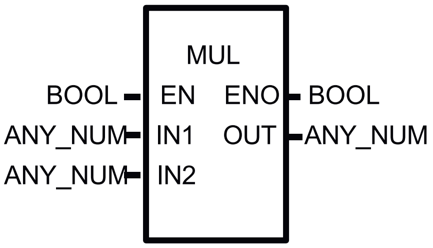
(MOVE)

Output (OUT): %QB0.4.0 (BYTE) = 16#18 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

MUL

Multiplication

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 MUL BOOL EN ENO BOOL ANY_NUM IN1 OUT ANY_NUM ANY_NUM IN2	**Input** EN: executes the function in case of 1 IN1: multiplicand IN2: multiplier Input is available to extend up to 8. **Output** ENO: without an error, it will be 1 OUT: multiplied value Variables connected to IN1, IN2, ..., OUT are all the same data type.

■ Function

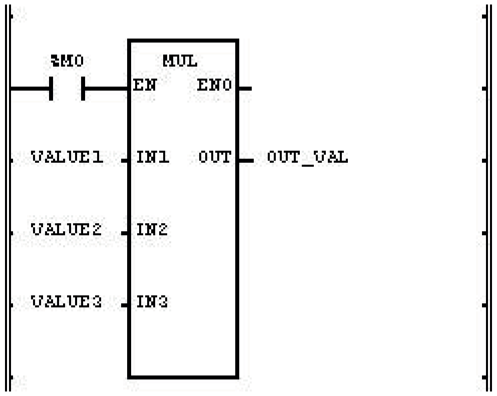
Multiplies an IN1, IN2,..., INn (n: input number) and outputs the result as OUT.

OUT = IN1 • IN2 • ... • INn

■ Error

If an output value is out of its data-type range, _ERR and _LER flags are set.

■ Program Example

LD	IL
 %M0 MUL EN ENO VALUE1 IN1 OUT OUT_VAL VALUE2 IN2 VALUE3 IN3	LD %M0 JMPN ABC LD VALUE1 MUL IN1:= CURRENT RESULT IN2:= VALUE2 IN3:= VALUE3 ST OUT_VAL ABC:

(1) If the transition condition (%M0) is ON, MUL function is executed.

(2) If input variables of MUL function, VALUE1 = 30, VALUE2 = 20, VALUE3 = 10, then the output variable OUT_VAL = 30 • 20 • 10 = 6000.

Input (IN1): VALUE1 (INT) = 30 (16#001E)

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	(

+ (MUL)

(IN2): VALUE2 (INT) = 20 (16#0014)

0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	(

+ (MUL)

(IN3): VALUE3 (INT) = 10 (16#000A)

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	(

↓

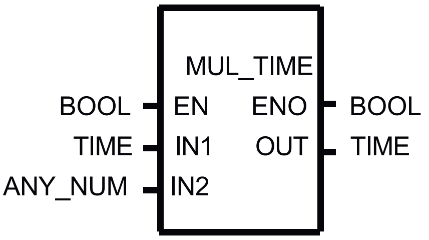
Output (OUT): OUT_VAL (INT) = 6000 (16#1770)

0	0	0	1	0	1	1	1	0	1	1	1	0	0	0	(

MUL_TIME

Time multiplication

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: time to be multiplied IN2: multiplying value **Output** ENO: without an error, it will be 1 OUT: multiplied result

■ Function

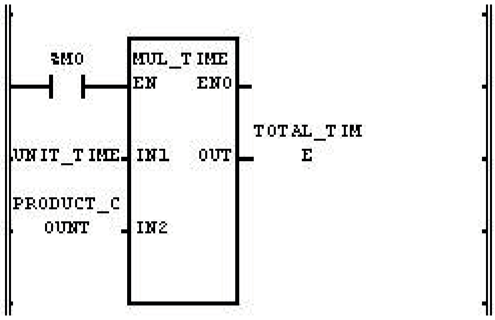
Multiplies the IN1 (time) by IN2 (number) and outputs the result time as OUT.

■ Error

If an output value is out of its TIME-data range, _ERR and _LER flags are set.

■ Program Example

This is the program that sets the required working time: the average estimated time per unit product is 20min 2sec and the number of product to produce a day is 20 in one product line.

LD	IL
	LD %M0 JMPN ABC LD UNIT_TIME MUL_TIME IN1:= CURRENT RESULT IN2:= PRODUCT_COUNT ST TOTAL_TIME ABC:

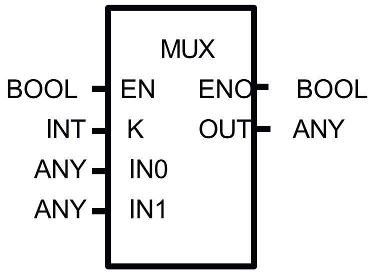
(1) Write input variable (IN1: the estimated time per unit product) UNIT_TIME: T#20M2S.
(2) Write input variable (IN2: quantity of production) PRODUCT_COUNT: 20.
(3) Write TOTAL_TIME to the output variable (OUT: total required working time).
(4) If the transition condition (%M0) is on, T#6H40M40S will be produced in output TOTAL_TIME.

Input (IN1): UNIT_TIME (TIME) = T#20MS2S
(MUL_TIME)
(IN2): PRODUCT_COUNT (INT) = 16#18
↓
Output (OUT): TOTAL_TIME (TIME) = T#6H40M40S

MUX

Selection from multiple inputs

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 MUX BOOL EN ENO BOOL INT K OUT ANY ANY IN0 ANY IN1	**Input** EN: executes the function in case of 1 K: selection IN0: the value to be selected IN1: the value to be selected Input variable number can be extended up to 8 **Output** ENO: without an error, it will be 1. OUT: the selected value IN0, IN1, ..., OUT should be the same time.

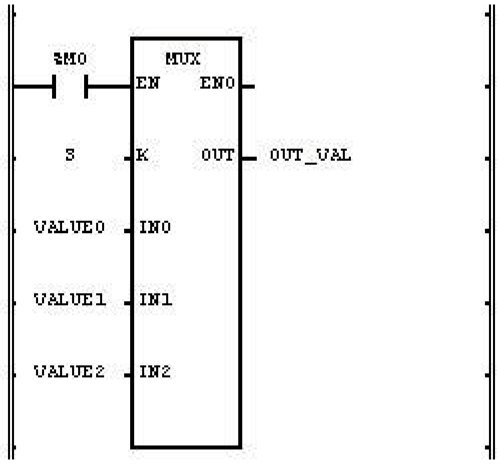
■ **Function**

Selects one among several inputs (IN0, IN1, .., INn) with K value and produces it.
If K = 0, IN0 will be an output; if K = 1, IN1 will be an output; if K = n, INn will be an output.

■ **Error**

If K is greater than or equal to the number of input variable INn, then IN0 will be an output and _ERR, _LER flags will be set.

■ **Program Example**

LD	IL
 %M0 MUX EN ENO S K OUT OUT_VAL VALUE0 IN0 VALUE1 IN1 VALUE2 IN2	LD %M0 JMPN ABC LD S MUX K:= CURRENT RESULT IN0:= VALUE0 IN1:= VALUE1 IN2:= VALUE2 ST OUT_VAL ABC:

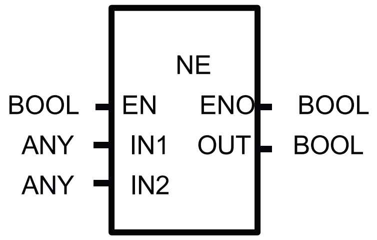
(1) If the transition condition (%M0) is on, MUX function will be executed.
(2) Input variable is selected by selection variable S and is moved to OUT.

Input (K): S (INT) = 2
(IN0): VALUE0 (WORD) = 16#11
(IN1): VALUE1 (WORD) = 16#22
(IN2): VALUE2 (WORD) = 16#33
↓ (MUX)
Output (OUT): OUT_VAL (WORD) = 16#33

NE

‘ Not equal to’ comparison

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

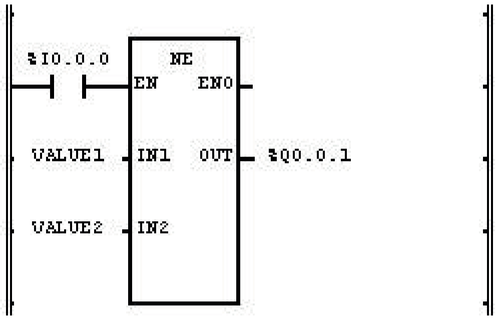
Function	Description
 NE BOOL EN ENO BOOL ANY IN1 OUT BOOL ANY IN2	**Input** EN: executes the function in case of 1 IN1: The value to be compared IN2: The value to be compared IN1, IN2 should be the same data type. **Output** ENO: without an error, it will be 1. OUT: the compared result value

■ **Function**

If IN1 is not equal to IN2, output OUT will be 1.

If INI is equal to IN2, output OUT will be 0.

■ **Program Example**

LD	IL
 %I0.0.0 NE EN ENO VALUE1 IN1 OUT %Q0.0.1 VALUE2 IN2	LD %I0.0.0 JMPN PP LD VALUE1 NE IN1:= CURRENT RESULT IN2:= VALUE2 ST %Q0.0.1 PP:

(1) If the transition condition (%I0.0.0) is on, NE function will be executed.

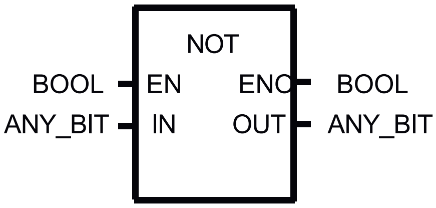
(2) If input variable VALUE1 = 300, VALUE2 = 200 (the compared result VALUE1 and VALUE2 are different), output result value will be %Q0.0.1 = 1.

Input (IN1): VALUE1 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	(
								(NE)								
(IN2): VALUE2 (INT) = 200 (16#0C8)	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	(
Output (OUT): %Q0.0.1 (BOOL) = 1 (16#1)																1

NOT

Reverse Logic (Logic inversion)

·······	····	····	····	····	····	····	····
············	· ·	· ·	· ·	· ·	· ·	· ·	· ·

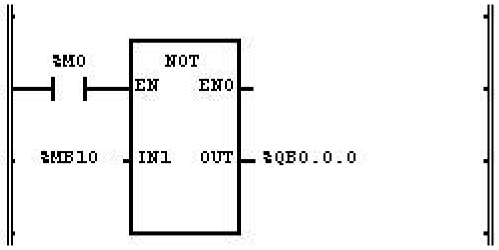
Function	Description
	**Input** EN: executes the function in case of 1 IN: the value to be logically inverted **Output** ENO: without an error, it will be 1 OUT: the inversed (NOT) value IN, OUT should be the same data type.

■ Function

It inverts the IN (by bit) and produces output OUT.

IN 1100 1010
OUT 0011 0101

■ Program Example

LD	IL
	LD %M0 JMPN AAA LD %MB10 NOT IN:= CURRENT RESULT ST %QB0.0.0 AAA:

(1) If the transition condition (%M0) is on, NOT function will be executed.

(2) If NOT function is executed, input data value of %MB10 will be inversed and will be written in %QB0.0.0.

Input (IN1): %MB10 (BYTE) = 16#CC

1	1	0	0	1	1	0	0

↓ (NOT)

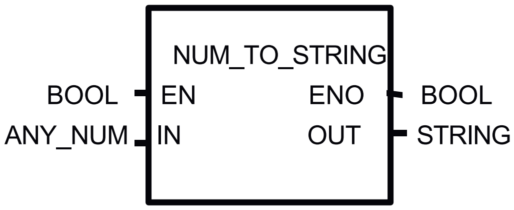
Output (OUT): %QB0.0.0 (BYTE) = 16#33

0	0	1	1	0	0	1	1

NUM_TO_STRING

Converts number to a character string

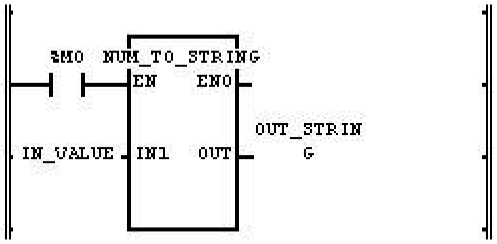
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 NUM_TO_STRING BOOL EN ENO BOOL ANY_NUM IN OUT STRING	**Input** EN: executes the function in case of 1 IN: input data to be converted to STRING **Output** ENO: without an error, it will be 1. OUT: converted data (character)

■ **Function**

It converts the numeric data of IN to the character data and produces output OUT.

■ **Program Example**

LD	IL
 %M0 NUM_TO_STRING EN ENO IN_VALUE IN1 OUT OUT_STRIN G	LD %M0 JMPN AAA LD IN_VALUE NUM_TO_STRING ST OUT_STRING

(1) If the transition condition (%M0) is ON, function NUM_TO_STRING will be executed.

(2) If IN_VALUE (INT) = 123, OUT_STRING will be ‘ 123’ ; if IN_VALUE (REAL) = 123.0, OUT_STRING will be ‘ 1.23E2’ .

Input (IN1): IN_VALUE (INT) = 123

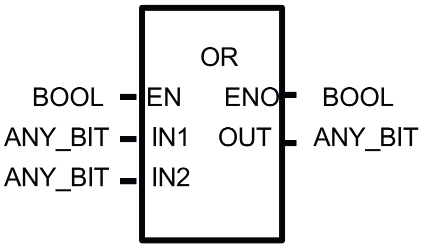
↓ (NUM_TO_STRING)

Output (OUT): OUT_STRING (STRING) = ‘ 123’

OR

Logical OR

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

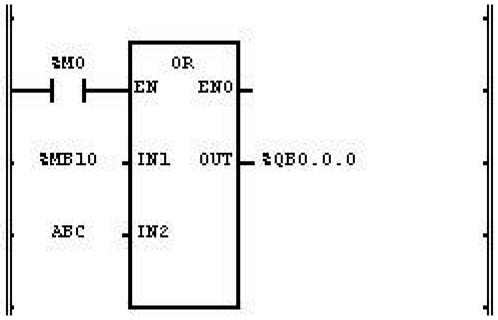
Function	Description
 OR BOOL EN ENO BOOL ANY_BIT IN1 OUT ANY_BIT ANY_BIT IN2	**Input** EN: executes the function in case of 1 IN1: input 1 IN2: input 2 Input variables can be extended up to 8. **Output** ENO: without an error, it will be 1. OUT: OR result IN1, IN2, OUT should be all the same data type.

■ Function

It performs a logical OR on the input variables by bit and produces output OUT.

IN1 1111 0000
OR
IN2 1010 1010
OUT 1111 1010

■ Program Example

LD	IL
 %M0 OR EN ENO %MB10 IN1 OUT %QB0.0.0 ABC IN2	LD %M0 JMPN AAA LD %MB10 OR IN1:= CURRENT RESULT IN2:= ABC ST %QB0.0.0

(1) If the transition condition (%M0) is on, function OR will be executed.

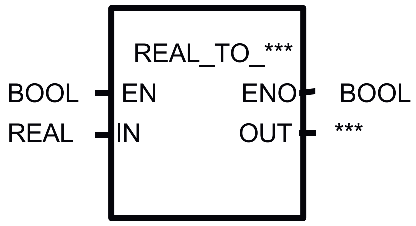
(2) The result of a logic sum (OR) for %MB10 = 11001100 and ABC = 11110000 will be produced in %QB0.0.0 = 11111100.

Input (IN1): %MB10 (BYTE) = 16#CC	1	1	0	0	1	1	0	0
Logical OR operation								
(IN2): ABC (BYTE) = 16#F0	1	1	1	1	0	0	0	0
↓								
Output (OUT): %QB0.0.0 (BYTE) = 16#FC	1	1	1	1	1	1	0	0

REAL_TO_***

REAL type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	•	•	•	•

Function	Description
	**Input** EN: executes the function in case of 1 IN: the REAL value to be converted **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ **Function**

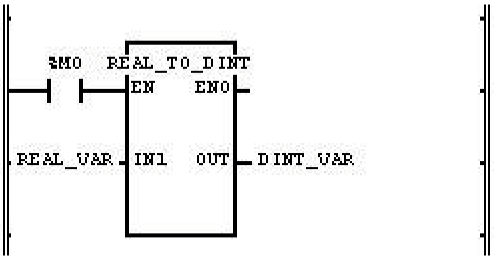
It converts the IN type and outputs it as OUT.

Function	Output type	Description
REAL_TO_SINT	SINT	If integer part of input is -128 • • 127, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_INT	INT	If integer part of input is -32768 • • 32767, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_DINT	DINT	If integer part of input is -2^{31} • • 2^{31}-1, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_LINT	LINT	If integer part of input is -2^{63} • • 2^{63}-1, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_USINT	USINT	If integer part of input is 0 • • 255, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_UINT	UINT	If integer part of input is 0 • • 65,535, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_UDINT	UDINT	If integer part of input is 0 • • 2^{32}-1, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_ULINT	ULINT	If integer part of input is 0 • • 2^{64}-1, normal conversion. Otherwise an error occurs. (Decimals round-off)
REAL_TO_DWORD	DWORD	Converts into DWORD type without changing the internal bit array.
REAL_TO_LREAL	LREAL	Converts REAL into LREAL type normally.

■ **Error**

If overflow occurs (an input value is greater than the value to be stored in output type), _ERR, _LER flags will be set. If an error occurs, the output will be 0.

■ **Program Example**

LD	IL
 %M0 REAL_TO_DINT EN ENO REAL_VAR IN1 OUT DINT_VAR	LD %M0 JMPN AAA LD REAL_VAL REAL_TO_DINT ST DINT_VAL

(1) If the transition condition (%M0) is ON, function REAL_TO_DINT will be executed.

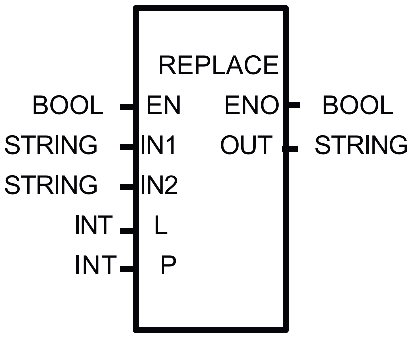
(2) If REAL_VAL (REAL type) = 1.234E4, DINT_VAL (DINT) = 12340.

Input (IN1): REAL_VAL(REAL) = 1.234E4
↓ (REAL_TO_DINT)
Output (OUT): DINT_VAL(DINT) = 12340

REPLACE

Replace a string (Character string replacement)

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: character string to be replaced IN2: character string to replace L: the length of character string to be replaced P: position of character string to be replaced **Output** ENO: without an error, it will be 1. OUT: output character string

■ Function

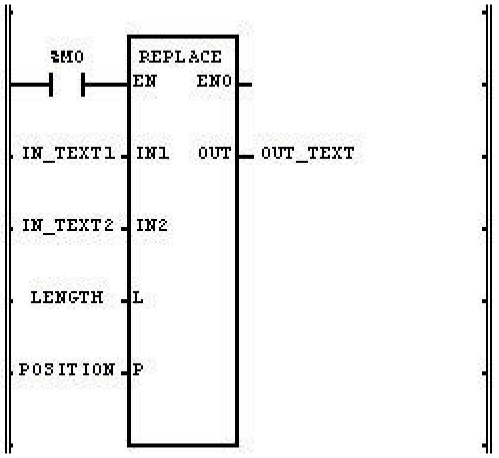
Its function is to remove the L-length charter from IN1 (starting from P) and put IN2 in the removed position as output OUT.

■ Error

_ERR, _LER flags will be set if:

- •P ≤ 0 or L < 0
- •P > (input character number of IN1)
- • character number of result > 30

■ Program Example

LD	IL
	LD %M0 JMPN MBC LD IN_TEXT1 REPLACE IN1:= CURRENT RESULT IN2: = IN_TEXT2 L: = LENGTH P: = POSITION ST OUT_TEXT ABC:

(1) If the transition condition (%M0) is ON, function REPLACE (character string replacement) will be executed.

(2) If input variable of character string to be replaced IN_TEXT1 = `ABCDEF`, input variable of character string to replace N_TEXT2 = `X`, input variable of character string length to be replaced LENGTH = 3 and input variable of character string position designation to be replaced POSITION = 2, then ‘ BCD’ of IN_TEXT will be replaced with ‘ X’ of IN_TEXT2 and output variable OUT_TEXT will be ‘ AXET’ .

Input (IN1): IN_TEXT1 (STRING) = `ABCDEF`
(IN2): IN_TEXT2 (STRING) = `X`
(L): LENGTH (INT) = 3
(P): POSITION (INT) = 2

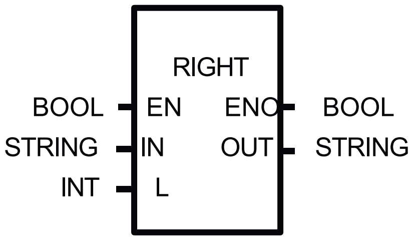
↓

Output (OUT): OUT_TEXT (STRING) = `AXET`

RIGHT

To take the right of character string

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪

Function	Description
 RIGHT BOOL EN ENO BOOL STRING IN OUT STRING INT L	**Input** EN: If EN is 1, function executes. IN: input character string L: length of character string **Output** ENO: without an error, it will be 1. OUT: output character string

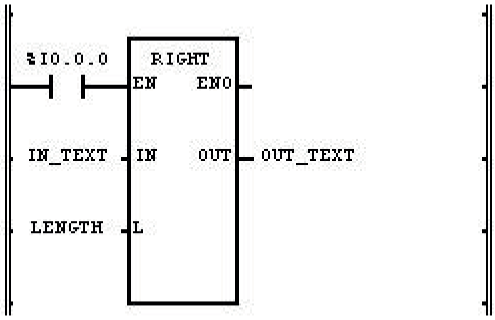
■ Function

It takes a right L-length character string of IN and produces output OUT.

■ Error

If L < 0, _ERR and _LER flags will be set.

■ Program Example

	IL
 %I0.0.0 RIGHT EN ENO IN_TEXT IN OUT OUT_TEXT LENGTH L	LD %I0.0.0 JMPN AAA LD IN_TEXT RIGHT IN:= CURRENT RESULT L: = LENGTH ST OUT_TEXT AAA:

(1) If the transition condition (%I0.0.0) is on, function RIGHT (to take the right of character string) will be executed.

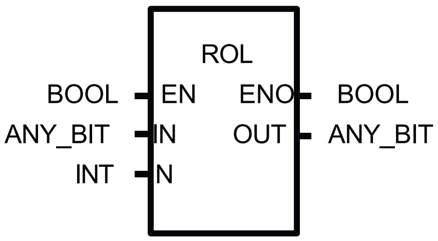
(2) If character string declared as input variable IN_TEXT = `ABCDEFG` and the length of character string to output LENGTH = 3, output character string variable OUT_TEXT = `EFG`.

Input (IN1): IN_TEXT (STRRING) = `ABCDEFG`
(L): LENGTH (INT) = 3
↓ (RIGHT)
Output (OUT): OUT_TEXT (STRRING) = `EFG`

ROL

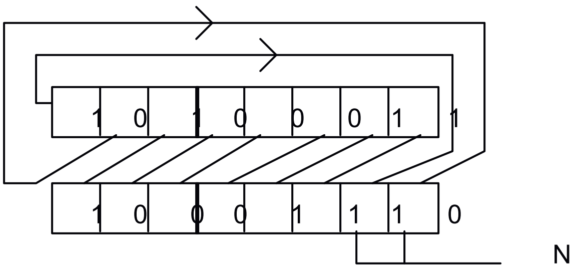
Rotate to left

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 ROL BOOL EN ENO BOOL ANY_BIT IN OUT ANY_BIT INT N	**Input** EN: executes the function in case of 1 IN: the value to be rotated N: bit number to rotate **Output** ENO: without an error, it will be 1 OUT: the rotated value

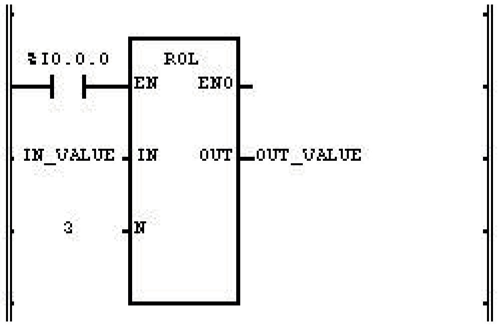
■ **Function**

It rotates input IN to the left as many as N bit number.



■ **Program Example**

This is the program that rotates the value of input data (1100_1100_1100_1100:16#CCCC) to the left by 3 bits if input %I0.0.0 is on.

LD	IL
 %I0.0.0 ROL EN ENO IN_VALUE IN OUT OUT_VALUE 3 N	LD %I0.0.0 JMPN PPP LD IN_VALUE ROL IN:= CURRENT RESULT N:= 3 ST OUT_VALUE PPP:

(1) Set input variable IN_VALUE to rotate.
(2) Set the value to be rotated (3).
(3) Set output variable to output the rotated data value as OUT_VALUE.
(4) If the transition condition (%I0.0.0) is ON, function ROL will be executed and a data bit set as input variable will be rotated to the left by 3 bits and produces output OUT_VALUE.

Input (IN1): IN_VALUE (WORD) = 16#CCCC

1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

(N): 3

↓ (ROL)

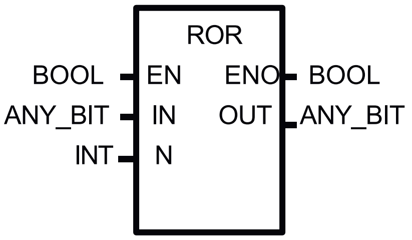
Output (OUT): OUT_VALUE (WORD) = 16#6666

0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0

ROR

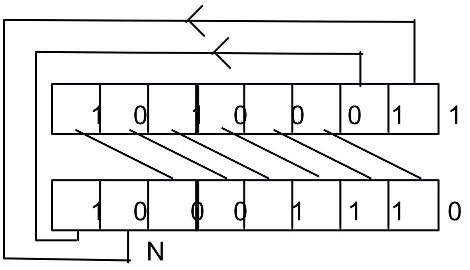
Rotate to right

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 ROR BOOL EN ENO BOOL ANY_BIT IN OUT ANY_BIT INT N	**Input** EN: executes the function in case of 1 IN: the value to be rotated N: bit number to rotate **Output** ENO: without an error, it will be 1. OUT: the rotated value

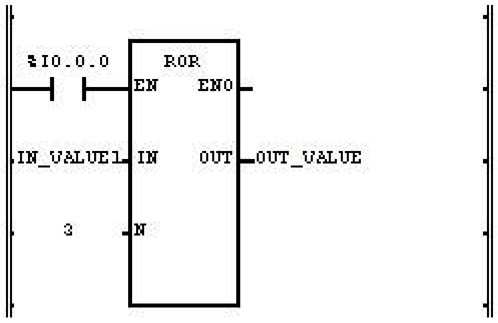
■ **Function**

It rotates input IN to the right as many as N bit number.



■ **Program Example**

This is the program that rotates input data value (1110001100110001: 16#E331) to the right by 3 bits if input %I0.0.0 is ON.

LD	IL
 %I0.0.0 ROR EN ENO IN_VALUE1 IN OUT OUT_VALUE 3 N	LD %I0.0.0 JMPN PO LD IN_VALUE1 ROR IN1:= CURRENT RESULT N:= 3 ST OUT_VALUE PO

(1) Set input variable of a data value to rotate as IN_VALUE1.

(2) Insert bit number 3 into bit number input N.

(4) If the transition condition (%I0.0.0) is ON, function ROR (rotate Right) will be executed and data bit set as input variable will be rotated to the right by 3 bits and produces output OUT_VALUE.

Input (IN1): IN_VALUE1 (WORD) = 16#E331

1	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1

(N): 3 ∨ (ROR)

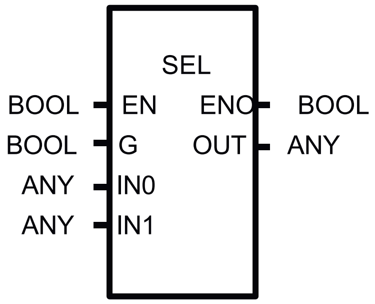
Output (OUT): OUT_VALUE(WORD) = 16#3C66

0	0	1	1	1	1	0	0	0	1	1	0	0	1	1	0

SEL

Selection from two inputs

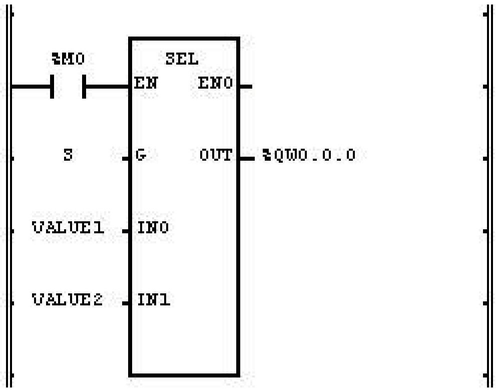
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 SEL BOOL EN ENO BOOL BOOL G OUT ANY ANY IN0 ANY IN1	**Input** EN: executes the function in case of 1 G: selection IN0: the value to be selected IN1: the value to be selected **Output** ENO: without an error, it will be 1 OUT: the selected value IN1, IN2, OUT should be all the same type.

■ Function

If G is 0, IN0 will be an output and if G is 1, IN1 will be an output.

■ Program Example

LD	IL
 %M0 SEL EN ENO S G OUT %QW0.0.0 VALUE1 IN0 VALUE2 IN1	LD %M0 JMPN PPP LD S SEL G:= CURRENT RESULT IN1:= VALUE1 IN2:= VALUE2 ST %QW0.0.0 PPP:

(1) If the transition condition (%M0) is ON, function SEL will be executed.

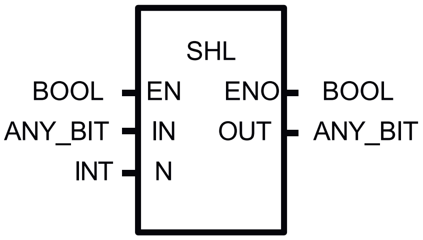
(2) If S = 1 and VALUE1 = 16#1110, VALUE2 = 16#FF00, then output variable %QW0.0.0 = 16#FF0.

Input (G): S = 1
(IN0): VALUE1 (WORD) = 16#1110
(IN1): VALUE2(WORD) = 16#FF00
⇓ (SEL)
Output (OUT): %QW0.0.0 (WORD) = 16#FF00

SHL

Shift Left

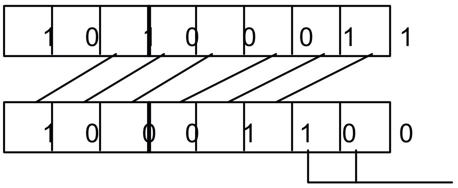
•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: If EN is 1, function is executed. IN: bit string to be shifted N: bit number to be shifted **Output** ENO: without an error, it will be 1 OUT: the shifted value

■ Function

It shifts input IN to the left as many as N bit number.

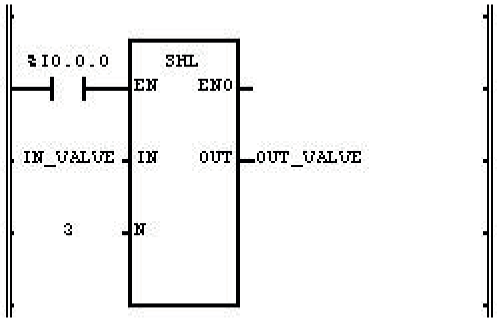
N number bit on the rightmost of input IN will be filled with 0.



N will be filled with 0.

■ Program Example

This is the program that shifts input data value (1100_1100_1100_1100:16#CCCC) to the left by 3 bits if input %I0.0.0 is ON.

LD	IL
	LD %I0.0.0 JMPN ABC LD IN_VALUE SHL IN:= CURRENT RESULT N:= 3 ST OUT_VALUE ABC:

(1) Set the input variable IN_VALUE (11001110:16#CE).

(2) Insert bit number 3 into N.

(3) If the transition condition (%Z0.0.0) is ON, function SHL (shift Left) will be executed and data bit set as input variable shifts to the left by 3 bits and produces output OUT_VALUE.

Input (IN1): IN_VALUE (WORD) = 16#CCCC

1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	(

(N): 3

(SHL)

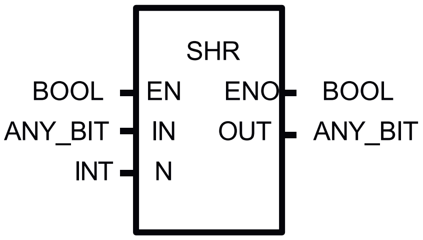
Output (OUT): OUT_VALUE (WORD) =16#6660

0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	(

SHR

Shift Right

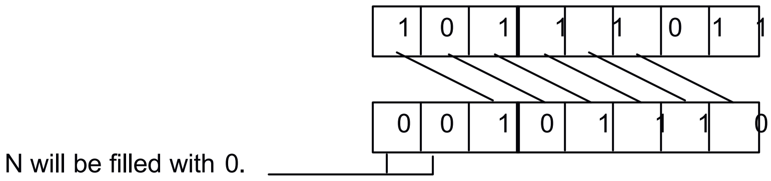
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN: bit string to be shifted N: bit number to be shifted **Output** ENO: without an error, it will be 1. OUT: the shifted value

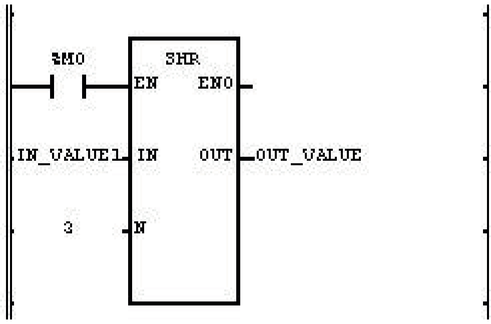
■ Function

It shifts input IN to the right as many as N bit number.

N number bit on the leftmost of input IN will be filled with 0.



■ Program Example

LD	IL
	LD %M0 JMPN AAA LD IN_VALUE SHR IN:= CURRENT RESULT N:= SHIFT_NUM ST OUT_VALUE

(1) If the transition condition (%M0) is on, function SHL (Shift Left) will be executed.

(2) Data bit set as input variable shift to the right by 3 bits and produces outputs OUT_VALUE.

Input (IN1): IN_VALUE (WORD) = 16#E331

(N): 3

Output (OUT): OUT_VALUE (WORD) = 16#1C66

1	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1

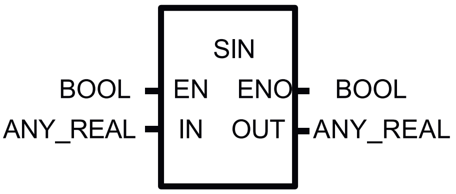
↓ (SHR)

0	0	0	1	1	1	0	0	0	1	1	0	0	1	1	(

SIN

Sine operation

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

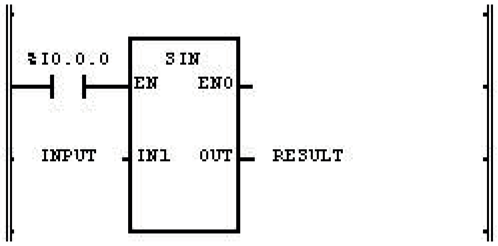
Function	Description
 SIN BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: input value of Sine operation (radian) **Output** ENO: without an error, it will be 1 OUT: Sine operation result value IN, OUT should be the same data type.

■ Function

Finds the Sine operation value of IN and produces output OUT.

OUT = SIN (IN)

■ Program Example

LD	IL
 %I0.0.0 SIN EN ENO INPUT IN1 OUT RESULT	LD %I0.0.0 JMPN PPP LD INPUT SIN ST RESULT PPP:

(1) If the transition condition (%I0.0.0) is ON, function SIN (Sine operation) will be executed.

(2) If the value of input variable INPUT is 1.0471 .... (π/3 rad = 60°), RESULT declared as output variable will be 0.8660 .... ($\sqrt{3}/2$).

SIN (π/3) = $\sqrt{3}/2$ = 0.8660

Input (IN1): INPUT (REAL) = 1.0471

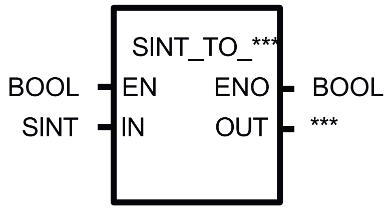
↓ (SIN)

Output (OUT): RESULT (REAL) = 8.65976572E-01

SINT_TO_***

SINT type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	• ••	• ••	• ••	• ••

Function	Description
 SINT_TO_*** BOOL EN ENO BOOL SINT IN OUT ***	**Input** EN: executes the function in case of 1 IN: short Integer value **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

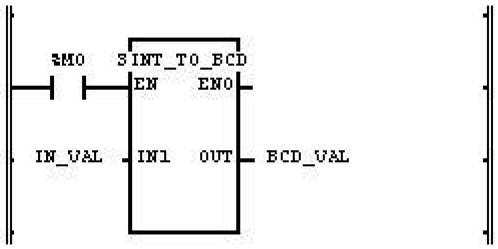
It converts the IN type and outputs it as OUT.

Function	Output type	Description
SINT_TO_INT	INT	Converts into INT type normally.
SINT_TO_DINT	DINT	Converts into DINT type normally.
SINT_TO_LINT	LINT	Converts into LINT type normally.
SINT_TO_USINT	USINT	If input is 0 • • 127, normal conversion. Otherwise an error occurs.
SINT_TO_UINT	UINT	If input is 0 • • 127, normal conversion. Otherwise an error occurs.
SINT_TO_UDINT	UDINT	If input is 0 • • 127, normal conversion. Otherwise an error occurs.
SINT_TO_ULINT	ULINT	If input is 0 • • 127, normal conversion. Otherwise an error occurs.
SINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
SINT_TO_BYTE	BYTE	Converts into BYTE type without changing the internal bit array.
SINT_TO_WORD	WORD	Converts into WORD type filling the upper bits with 0.
SINT_TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
SINT_TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
SINT_TO_BCD	BYTE	If input is 0 ~ 99, normal conversion. Otherwise an error occurs.
SINT_TO_REAL	REAL	Converts SINT into REAL type normally.
SINT_TO_LREAL	LREAL	Converts SINT into LREAL type normally.

■ Error

If a conversion error occurs, _ERR and _LER flags will be set. If an error occurs, take the lower bits as many as bit number of output type and output it without changing the internal bit array.

■ **Program Example**

LD	IL
 %M0 SINT_TO_BCD EN ENO IN_VAL IN1 OUT BCD_VAL	LD %M0 JMPN AAA LD IN_VAL SINT_TO_BCD ST BCD_VAL AAA:

(1) If the input condition (% M0) is ON, function SINT_TO_BCD will be executed.

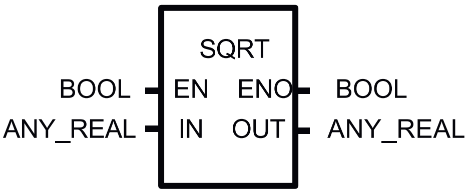
(2) If input variable IN_VAL (SINT) = 64 (2#0100_0000), output variable OUT_VAL (BCD type) = 16#64 (2#0110_0100).

Input (IN1): IN_VAL(SINT) = 64(16#40)	0	1	0	0	0	0	0	0
					↓ (SINT_TO_BCD)			
Output (OUT): OUT_VAL(BCD) = 16#64(16#64)	0	1	1	0	0	1	0	0

SQRT

Calculate SQRT (Square root operation)

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	•	•	•	•

Function	Description
 SQRT BOOL EN ENO BOOL ANY_REAL IN OUT ANY_REAL	**Input** EN: executes the function in case of 1 IN: input value of square root operation **Output** ENO: without an error, it will be 1. OUT: square root value IN, OUT should be the same data type.

■ Function

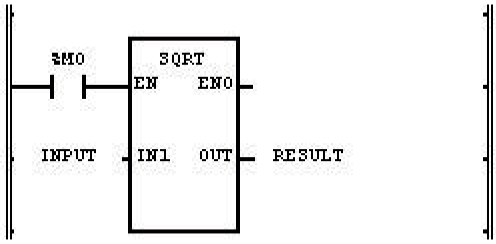
It finds the square root value of IN and output it as OUT.

OUT = $\sqrt{IN}$

■ Error

If the value of IN is a negative number, _ERR and _LER flag will be set.

■ Program Example

LD	IL
	LD %M0 JMPN AAA LD INPUT SQRT ST RESULT AAA:

(1) If the transition condition (%M0) is ON, function SQRT (square root operation) will be executed.

(2) If the value of input variable declared as INPUT is 9.0, RESULT declared as output variable will be 3.0.

$\sqrt{9.0} = 3.0$

Input (IN1): INPUT (REAL) = 9.0

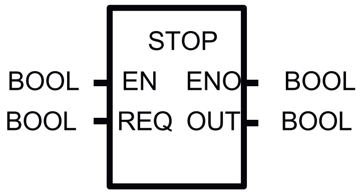
↓ (SQRT)

Output (OUT): RESULT (REAL) = 3.0

STOP

Stop running by program

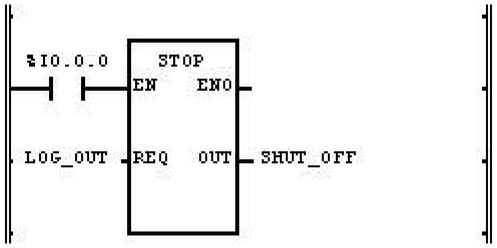
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 STOP BOOL EN ENO BOOL BOOL REQ OUT BOOL	**Input** EN: executes the function in case of 1 RE: requires the operation stop by program **Output** ENO: without an error, it will be 1. OUT: If STOP function is executes, it will be 1.

■ **Function**

- • If EN and REQ are 1, stop running and return to STOP mode.
- • If function 'STOP' is executed, the program stops after completing scan program in executing.
- • Program restarts in case of power re-supply or the change of operation mode from STOP to RUN.

■ **Program Example**

LD	IL
 %I0.0.0 STOP EN ENO LOG_OUT REQ OUT SHUT_OFF	LD %I0.0.0 JMPN PT LD LOG_OUT STOP ST SHUT_OFF PT:

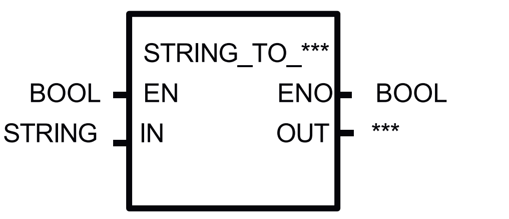
(1) If the transition condition (%I0.0.0) and LOG_OUT is 1, it becomes to STOP mode after completing the scan program in executing.

(2) It is recommended to turn off the power of PLC in the stable state after executing 'STOP' function declared as input variable.

STRING_TO_***

STRING type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 STRING_TO_*** BOOL EN ENO BOOL STRING IN OUT ***	**Input** EN: If EN is 1, function converts. IN: character string **Output** ENO: without an error, it will be 1. OUT: type-converted data

■ Function

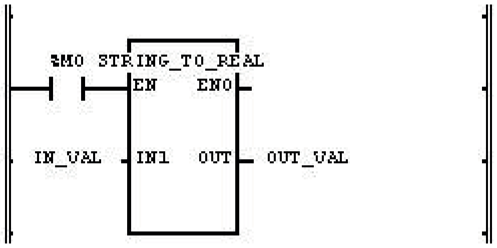
Converts the IN type and outputs it as OUT.

Function	Output type	Description
STRING _TO_SINT	SINT	Converts STRING into SINT type.
STRING _TO_INT	INT	Converts STRING into INT type.
STRING _TO_DINT	DINT	Converts STRING into DINT type.
STRING _TO_LINT	LINT	Converts STRING into LINT type.
STRING _TO_USINT	USINT	Converts STRING into USINT type.
STRING _TO_UINT	UINT	Converts STRING into UINT type.
STRING _TO_UDINT	UDINT	Converts STRING into UDINT type.
STRING _TO_ULINT	ULINT	Converts STRING into ULINT type.
STRING _TO_BOOL	BOOL	Converts STRING into BOOL type.
STRING _TO_BYTE	BYTE	Converts STRING into BYTE type.
STRING _TO_WORD	WORD	Converts STRING into WORD type.
STRING _TO_DWORD	DWORD	Converts STRING into DWORD type.
STRING _TO_LWORD	LWORD	Converts STRING into LWORD type.
STRING _TO_REAL	REAL	Converts STRING into REAL type.
STRING _TO_LREAL	LREAL	Converts STRING into LREAL type.
STRING _TO_DT	DT	Converts STRING into DT type.
STRING _TO_DATE	DATE	Converts STRING into DATE type.
STRING _TO_TOD	TOD	Converts STRING into TOD type.
STRING _TO_TIME	TIME	Converts STRING into TIME type.

■ Error

If input character type does not match with output data type, _ERR and _LER flags will be set.

■ **Program Example**

LD	IL
 %M0 STRING_TO_REAL EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN ZZ LD IN_VAL STRING_TO_REAL ST OUT_VAL ZZ:

(1) If the input condition (%M0) is ON, function STRING_TO_REAL will be executed.

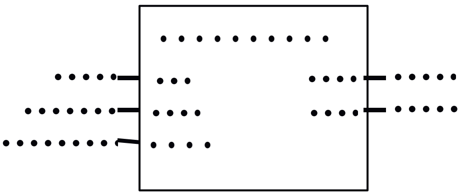
(2) If input variable IN_VAL (STRING) = ‘ -1.34E12’ , output variable OUT_VAL (REAL) = -1.34E12.

Input (IN1): IN_VAL (STRING) = ‘ -1.34E12’
↓ (STRING_TO_REAL)
Output (OUT): OUT_VAL (REAL) = -1.34E12

STRING_TO_ARY

....

....	....	....	....	....	....	....	....
....	. .	. .	. .	. .	. .	. .	. .

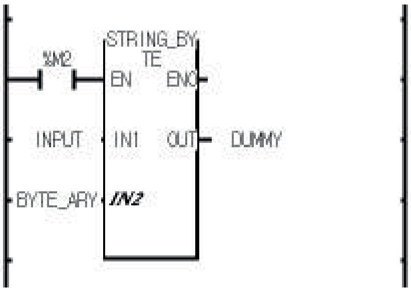
Function	Description
	**Input** EN: If EN is 1, function converts. IN: string input **Output** ENO: without an error, it will be 1. OUT: dummy output **In/Out** IN2: converted byte array output

■ Function

It converts a string into 30 byte arrays.

■ Program Example

LD



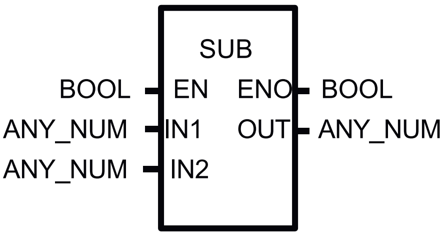
(1) If the transition condition (%M2) is on, STRING_BYTE function is executed.

(2) If input variable INPUT is “ GM4-CPUA” , In/Out variable BYTE_ARY is as follows:
16#{22(“), 47(G), 4D(M), 34(4), 2D(-), 43(C), 50(P), 55(U), 41(A), 22(“)}.

SUB

Subtraction

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 SUB BOOL EN ENO BOOL ANY_NUM IN1 OUT ANY_NUM ANY_NUM IN2	**Input** EN: executes the function in case of 1 IN1: the value to be subtracted IN2: the value to subtract **Output** ENO: without an error, it will be 1. OUT: the subtracted result value The variables connected to IN1, IN2 and OUT should be all the same data type.

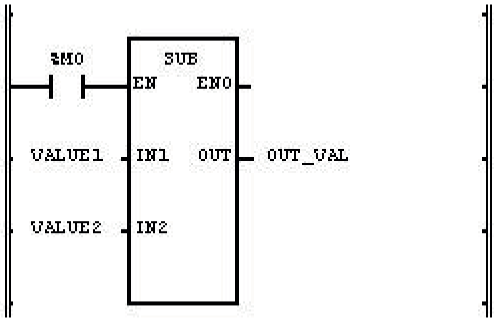
■ Function

It subtracts IN2 from IN1 and outputs it as OUT.
OUT = IN1 • IN2

■ Error

If output value is out of range of related data type, _ERR and _LER flags will be set.

■ Program Example

LD	IL
 %M0 SUB EN ENO VALUE1 IN1 OUT OUT_VAL VALUE2 IN2	LD %M0 JMPN AAA LD VALUE1 SUB IN1:= CURRENT RESULT IN2:= VALUE2 ST OUT_VAL AAA:

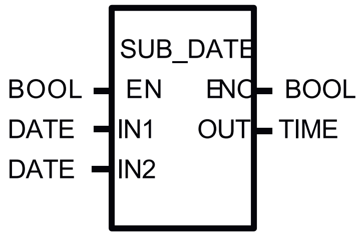
(1) If the transition condition (%M0) is ON, function SUB will be executed.
(2) If input variables VALUE1 = 300, VALUE2 = 200, OUT_VAL will be 100 after operation.

Input (IN1): VALUE1 (INT) = 300 (16#012C)	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	(
											- (SUB)					
(IN2): VALUE2 (INT) = 200 (16#00C8)	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	(
Output (OUT): OUT_VAL (INT) = 100 (16#0064)	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	(

SUB_DATE

Date subtraction

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 SUB_DATE BOOL EN ENO BOOL DATE IN1 OUT TIME DATE IN2	**Input** EN: executes the function in case of 1 IN1: standard date IN2: the date to subtract **Output** ENO: without an error, it will be 1. OUT: produces the difference between two dates as time data.

■ Function

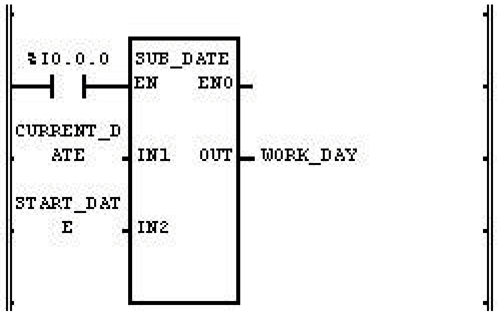
It subtracts IN2 (specific date) from IN1(standard date) and outputs the difference between two dates as OUT.

■ Error

If output value is out of range (TIME data type), _ERR and _LER flags will be set.

An error occurs: 1) when date difference exceeds the range of TIME data type (T#49D17H2M47S295MS); 2) the result of date operation is a negative number.

■ Program Example

LD	IL
 %I0.0.0 SUB_DATE EN ENO CURRENT_D ATE IN1 OUT WORK_DAY START_DAT E IN2	LD %I0.0.0 JMPN PPP LD CURRENT_DATE SUB_DATE IN1:= CURRENT RESULT IN2:= START_DATE ST WORK_DAY PPP:

(1) If the transition condition (%I0.0.0) is ON, function SUB_DATE will be executed.

(2) If input variable CURRENT_DATE is D#1995-12-15 and START_DATE is D#1995-11-1, the working days declared as output variable WORK_DAY will be T#44D.

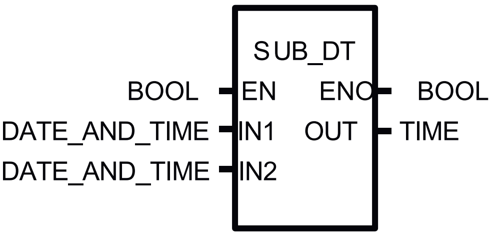
Input (IN1): CURRENT_DATE (DATE) = D#1995-12-15
(SUB_DATE)
(IN2): START_DATE (DATE) = D#1995-11-1

Output (OUT): WORK_DAY (TIME) = T#44D ↓

SUB_DT

Date and Time subtraction

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	• ••	• ••	• ••	• ••

Function	Description
 SUB_DT BOOL EN ENO BOOL DATE_AND_TIME IN1 OUT TIME DATE_AND_TIME IN2	**Input** EN: executes the function in case of 1 IN: standard date and time of day IN2: date and time of day to subtract **Output** ENO: without an error, it will be 1. OUT: the subtracted result time

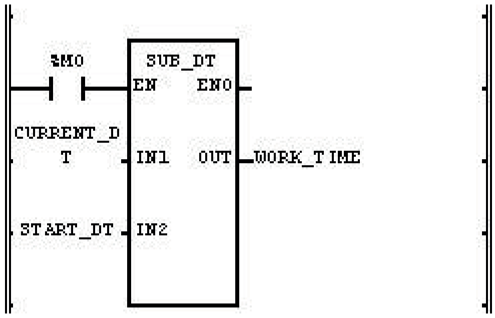
■ **Function**

It subtracts IN2 (specific date and time of day) from IN1 (standard date and time of day) and outputs the time difference as OUT.

■ **Error**

If output value is out of range of TIME data type, _ERR and _LER flags will be set.
If the result of date and time of day subtraction operation is a negative number, an error occurs.

■ **Program Example**

LD	IL
 %M0 SUB_DT EN ENO CURRENT_D T IN1 OUT WORK_TIME START_DT IN2	LD %M0 JMPN PPP LD CURRENT_DT SUB_DT IN1:= CURRENT RESULT IN2:= START_DT ST WORK_TIME PPP:

(1) If the transition condition (%M0) is ON, function SUB_DT (Time and Date subtraction) will be executed.

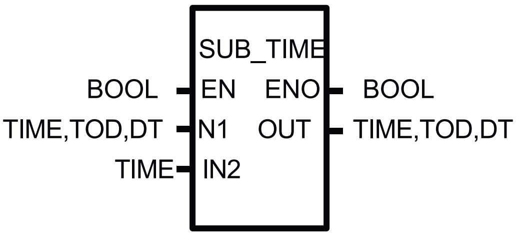
(2) If the current date and time of day CURRENT_DT is DT#1995-12-15-14:30:00 and the starting date and the time of day to work START_DT is DT#1995-12-13-12:00:00, the continuous working time declared as output variable WORK_TIME will be T#2D2H30M.

```
Input (IN1): CURRENT_DT (DT) = DT#1995-12-15-14:30:00
                                            (SUB_DATE)
      (IN2): START_DT (DT) = DT#1995-12-13-12:00:00
                                               ↓
Output (OUT): WORK_TIME (TIME) = T#2D2H30M
```

SUB_TIME

Time subtraction

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 SUB_TIME BOOL EN ENO BOOL TIME,TOD,DT IN1 OUT TIME,TOD,DT TIME IN2	**Input** EN: executes the function in case of 1 IN1: standard time of day IN2: the time to subtract **Output** ENO: without an error, it will be 1. OUT: the subtracted result time or time of day OUT data type is the same as the input IN1 type. That is, if IN1 type is TIME, OUT type should be TIME.

■ Function

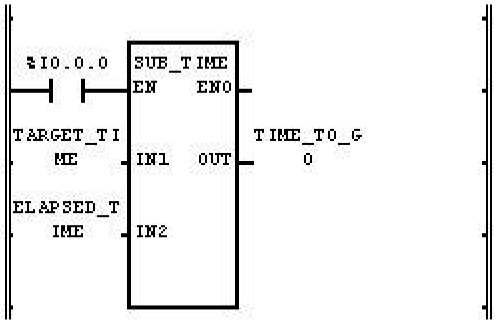
- • If IN1 is TIME, it subtracts the time from the standard time and produces OUT (time difference).
- • If IN1 is TIME_OF_DAY, it subtracts the time from the standard time of day and outputs the time of a day as OUT.
- • If IN1 is DATE_AND_TIME, it subtracts the time from the standard date and the time of day and produces the date and the time of day as OUT.

■ Error

If the output value is out of range of related data type, _ERR and _LER flags will be set.

If the result subtracting the time from the standard time is a negative number or the result subtracting the time from the time of day is a negative number, an error occurs.

■ Program Example

LD	IL
 %I0.0.0 SUB_TIME EN ENO TARGET_TIME IN1 OUT TIME_TO_GO ELAPSED_TIME IN2	LD %I0.0.0 JMPN AAA LD TARGET_TIME SUB_TIME IN1:= CURRENT RESULT IN2:= ELAPSED_TIME ST TIME_TO_GO AAA:

(1) If the transition condition (%I0.0.0) is ON, function SUB_TIME (time subtraction) will be executed.

(2) If total working time declared as input variable TARGET_TIME is T#2H30M, the elapsed time ELAPSED_TIME is T#1H10M30S300MS, the remaining working time declared as output variable TIME_TO_GO will be T#1H19M29S700MS.

Input (IN1): TARGET_TIME (TIME) = T#2H30M

(SUB_TIME)

(IN2): ELAPSED_TIME (TIME) = T#1H10M30S300MS

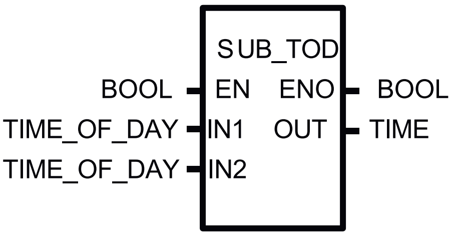
↓

Output (OUT): TIME_TO_GO (TIME) = T#1H19M29S700MS

SUB_TOD

TOD Subtraction

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 SUB_TOD BOOL EN ENO BOOL TIME_OF_DAY IN1 OUT TIME TIME_OF_DAY IN2	**Input** EN: executes the function in case of 1 IN1: standard time of day IN2: the time of day to subtract **Output** ENO: without an error, it will be 1. OUT: the subtracted result time

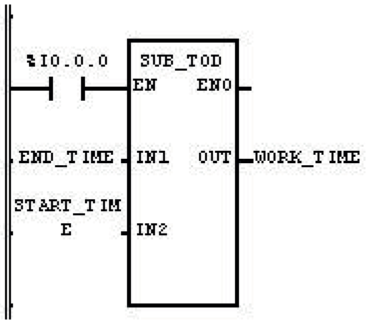
■ Function

It subtracts the IN2 (specific time of day) from IN1 (standard time of day) and outputs the time difference as OUT.

■ Error

If the result subtracting the time of day from the time of day is a negative number, an error occurs.

■ Program Example

LD	IL
 %I0.0.0 SUB_TOD EN ENO END_TIME IN1 OUT WORK_TIME START_TIM E IN2	LD %I0.0.0 JMPN AAA LD END_TIME SUB_TOD IN1:= CURRENT RESULT IN2:= START_TIME ST WORK_TIME AAA:

(1) If the transition condition (%I0.0.0) is ON, function SUB_TOD (time of day subtraction) will be executed.

(2) If END_TIME declared as input variable is TOD#14:20:30.5 and the starting time to work START_TIME is TOD#12:00:00, the required time to work WORK_TIME declared as output variable will be T#2H20M30S500MS.

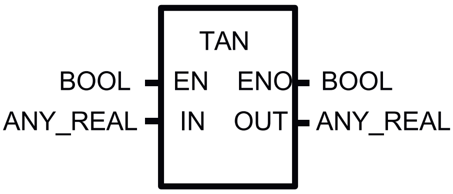
Input (IN1): END_TIME (TOD) = TOD#14:20:30.5
(SUB_TOD)
(IN2): START_TIME (TOD) = TOD#12:00:00
↓
Output (OUT): WORK_TIME (TIME) = T#2H20M30S500MS

TAN

Tangent Operation

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	•	• ••	•	•

* Applied only in GM4-CPUC among GM4 series

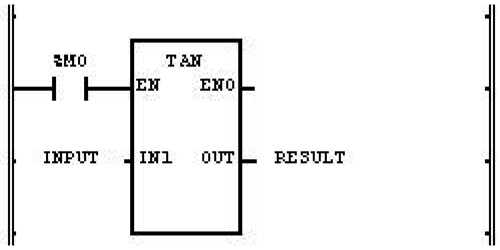
Function	Description
	**Input** EN: executes the function in case of 1 IN: tangent input value (radian) **Output** ENO: without an error, it will be 1 OUT: the result value of Tangent operation IN, OUT should be the same data type.

■ Function

It performs Tangent operation of IN and produces output OUT.

OUT = TAN (IN)

■ Program Example

LD	IL
	LD %M0 JMPN BBB LD INPUT TAN ST RESULT BBB:

(1) If the transition condition (%M0) is ON, function TAN (Tangent operation) will be executed.

(2) If the value of input variable declared as INPUT is 0.7853... (π/4 rad = 45°), RESULT declared as output variable will be 1.0000.

TAN (π/4) = 1

Input (IN1): INPUT (REAL) = 0.7853

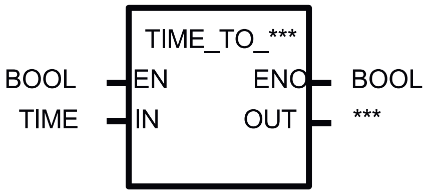
↓ (TAN)

Output (IN2): RESULT (REAL) = 9.99803722E-01

TIME_TO_***

TIME type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ••	• ••	• ••	• ••	• ••	• ••	• ••

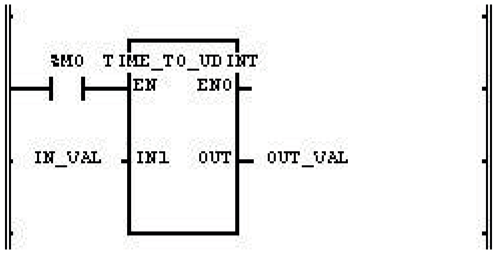
Function	Description
	**Input** EN: executes the function in case of 1 IN: time data to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ **Function**

It converts the IN type and produces OUT.

Function	Output type	Description
TIME_TO_UDINT	UDINT	Converts TIME into UDINT type. It converts only data type without changing the data (internal bit array state).
TIME_TO_DWORD	DWORD	Converts TIME into DWORD type. It converts only data type without changing the data (internal bit array state).
TIME_TO_STRING	STRING	Converts TIME into STRING type.

■ **Program Example**

LD	IL
	LD %M0 JMPN AA LD IN_VAL TIME_TO_UDINT ST OUT_VAL AA:

(1) If the transition condition (%M0) is ON, function TIME_TO_UDINT will be executed.

(2) If input variable IN_VAL (TIME) = T#120MS, output variable OUT_VAL (UDINT) = 120.

Input (IN1): IN_VAL (TIME) = T#120MS (16#78)

0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	(

↓ (TIME_TO_UDINT)

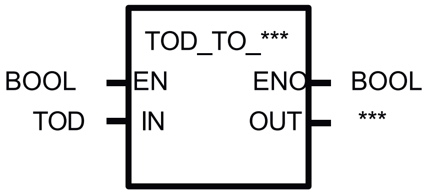
Output (OUT): OUT_VAL (UDINT) = 120 (16#78)

0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	(

TOD_TO_***

TOD type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪

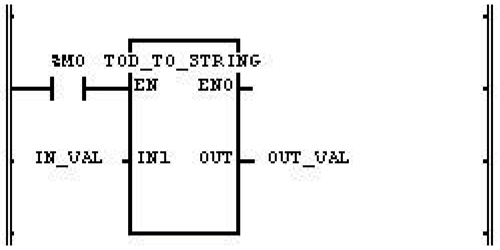
Function	Description
 TOD_TO_*** BOOL EN ENO BOOL TOD IN OUT ***	**Input** EN: executes the function in case of 1 IN: time of a day data to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

It converts the IN type and outputs it as OUT.

Function	Output type	Description
TOD_TO_UDINT	UDINT	Converts TOD into UDINT type. Converts only data type without changing a data (internal bit array state).
TOD_TO_DWORD	DWORD	Converts TOD into DWORD type. Converts only data type without changing a data (internal bit array state).
TOD_TO_STRING	STRING	Converts TOD into STRING type.

■ Program Example

LD	IL
 %M0 TOD_TO_STRING EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN AA LD IN_VAL DATE_TO_STRING ST OUT_VAL AA:

(1) If the transition condition (%M0) is ON, function TOD_TO_STRING will be executed.

(2) If input variable IN_VAL (TOD) = TOD#12:00:00, output variable OUT_VAL (STRING) = ‘ TOD#12:00:00’ .

Input (IN1): IN_VAL (TOD) = TOD#12:00:00

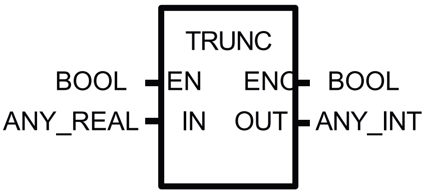
↓ (TOD_TO_STRING)

Output (IN2): OUT_VAL (STRING) = ‘ TOD#12:00:00’

TRUNC

Set TRUNC (Round off the decimal fraction of IN and converts into integer number)

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 TRUNC BOOL EN ENO BOOL ANY_REAL IN OUT ANY_INT	**Input** EN: executes the function in case of 1 IN: REAL value to be converted **Output** ENO: without an error, it will be 1. OUT: the Integer converted value

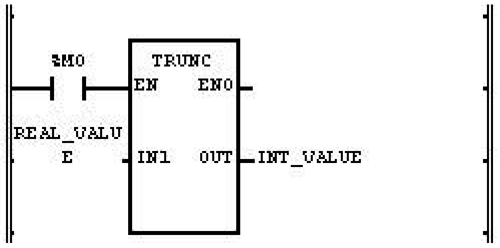
■ **Function**

Function	Input type	Output type	Description
TRUNC	REAL LREAL	DINT LINT	Round off the decimal fraction of input IN and outputs the Integer value as OUT.

■ **Error**

_ERR, _LER flags will be set: 1) if the converted value is greater than maximum value of data type connected to OUT; 2) if the variable connected to OUT is Unsigned Integer and the converted output value is a negative number, the output is 0.

■ **Program Example**

LD	IL
 %M0 TRUNC EN ENO REAL_VALU E IN1 OUT INT_VALUE	LD REAL_VALUE TRUNC ST INT_VALUE

(1) If the transition condition (%M0) is ON, function TRUNC will be executed.

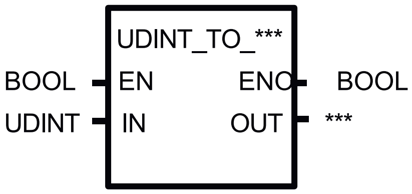
(2) If input variable REAL_VALUE (REAL) = 1.6, output variable INT_VALUE (INT) = 1.
If REAL_VALUE (REAL) = -1.6, INT_VALUE (INT) = -1.

Input (IN1): REAL_VALUE (REAL) = 1.6
↓ (TRUNC)
Output (OUT): INT_VALUE (INT) = 1

UDINT_TO_***

UDINT type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 UDINT_TO_*** BOOL EN ENO BOOL UDINT IN OUT ***	**Input** EN: executes the function in case of 1 IN: Unsigned Double Integer value to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

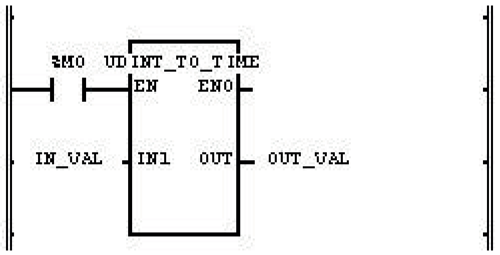
It converts the IN type and outputs it as OUT.

Function	Output type	Description
UDINT_TO_SINT	SINT	If input is 0~127, normal conversion. Otherwise an error occurs.
UDINT_TO_INT	INT	If input is 0~32767, normal conversion. Otherwise an error occurs.
UDINT_TO_DINT	DINT	If input is 0~2,147,483,64, normal conversion. Otherwise an error occurs.
UDINT_TO_LINT	LINT	Converts UDINT into LINT type normally.
UDINT_TO_USINT	USINT	If input is 0~255, normal conversion. Otherwise an error occurs.
UDINT_TO_UINT	UINT	If input is 0~65535, normal conversion. Otherwise an error occurs.
UDINT_TO_ULINT	ULINT	Converts UDINT into ULINT type normally.
UDINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
UDINT_TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
UDINT_TO_WORD	WORD	Takes the lower 16 bits and converts into WORD type.
UDINT_TO_DWORD	DWORD	Converts into DWORD type without changing the internal bit array.
UDINT_TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
UDINT_TO_BCD	DWORD	If input is 0 ~ 99,999,999, normal conversion. Otherwise an error occurs.
UDINT_TO_REAL	REAL	Converts UDINT into REAL type. During the conversion, an error caused by the precision may occur.
UDINT_TO_LREAL	LREAL	Converts UDINT into LREAL type. During the conversion, an error caused by the precision may occur.
UDINT_TO_TOD	TOD	Converts into TOD type without changing the internal bit array.
UDINT_TO_TIME	TIME	Converts into TIME type without changing the internal bit array.

■ Error

If a conversion error occurs, _ERR and _LER flags will be set. If an error occurs, take the lower bits as many as a bit number of an output data type and produces the output without changing the internal bit array.

■ **Program Example**

LD	IL
 %M0 UDINT_TO_TIME EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN ZZ LD IN_VAL UDINT_TO_TIME ST OUT_VAL ZZ:

(1) If the input condition (%M0) is ON, function UDINT_TO_TIME will be executed.

(2) If input variable IN_VAL (UDINT) = 123, output variable OUT_VAL (TIME) = T#123MS.

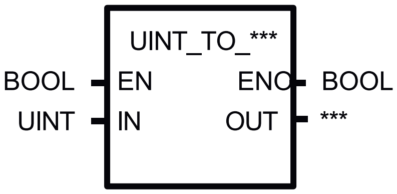
Input (IN1): IN_VAL (UDINT) = 123 ↓

Output (OUT): OUT_VAL (TIME) = T#123MS

UINT_TO_***

UINT type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN: Unsigned Integer value to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

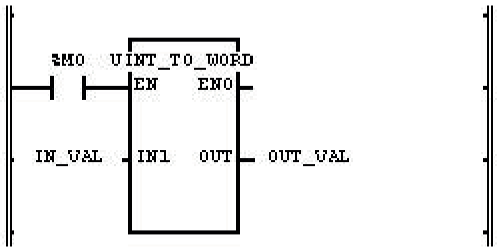
It converts the IN type and outputs it as OUT.

Function	Output type	Description
UINT_TO_SINT	SINT	If input is 0~127, normal conversion. Otherwise an error occurs.
UINT_TO_INT	INT	If input is 0~32,767, normal conversion. Otherwise an error occurs.
UINT_TO_DINT	DINT	Converts UINT into UDINT type normally.
UINT_TO_LINT	LINT	Converts UINT into ULINT type normally.
UINT_TO_USINT	USINT	If input is 0~255, normal conversion. Otherwise an error occurs.
UINT_TO_UDINT	UDINT	Converts UINT into UDINT type normally.
UINT_TO_ULINT	ULINT	Converts UINT into ULINT type.
UINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
UINT_TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
UINT_TO_WORD	WORD	Converts into WORD type without changing the internal bit array.
UINT_TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
UINT_TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
UINT_TO_BCD	BCD	If input is 0~99,999,999, normal conversion. Otherwise an error occurs.
UINT_TO_REAL	REAL	Converts UINT into REAL type.
UINT_TO_LREAL	LREAL	Converts UINT into LREAL type.
UNIT_TO_DATE	DATE	Converts into DATE type without changing the internal bit array.

■ Error

If a conversion error occurs, _ERR and _LER flags will be set. If error occurs, it takes as many lower bits as a bit number of output type and produces an output without changing its internal bit array.

■ **Program Example**

LD	IL
	LD %M0 JMPN PO LD IN_VAL UINT_TO_WORD ST OUT_VAL PO:

(1) If the input condition (%M0) is ON, function UINT_TO_WORD will be executed.

(2) If input variable IN_VAL (UINT) = 255 (2#0000_0000_1111_1111),
output variable OUT_VAL (WORD) = 2#0000_0000_1111_1111.

Input (IN1): IN_VAL (UINT) = 255

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

↓ (UINT_TO_WORD)

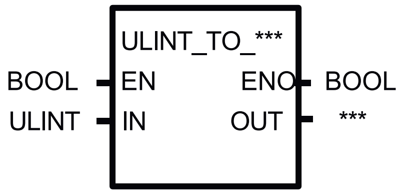
Output (OUT): OUT_VAL (WORD) = 16#FF

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

ULINT_TO_***

ULINT type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ▪	• ▪	• ▪	•	•	•	•

Function	Description
	**Input** EN: executes the function in case of 1 IN: Unsigned Long Integer value to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

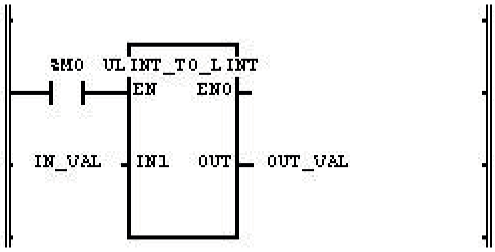
It converts the IN type and outputs it as OUT.

Function	Output type	Description
ULINT_TO_SINT	SINT	If input is 0~127, normal conversion. Otherwise an error occurs.
ULINT_TO_INT	INT	If input is 0~32,767, normal conversion. Otherwise an error occurs.
ULINT_TO_DINT	DINT	If input is 0~2^{31}-1, normal conversion. Otherwise an error occurs.
ULINT_TO_LINT	LINT	If input is 0~2^{63}-1, normal conversion. Otherwise an error occurs.
ULINT_TO_USINT	USINT	If input is 0~255, normal conversion. Otherwise an error occurs.
ULINT_TO_UINT	UINT	If input is 0~65,535, normal conversion. Otherwise an error occurs.
ULINT_TO_UDINT	UDINT	If input is 0~2^{32}-1, normal conversion. Otherwise an error occurs.
ULINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
ULINT_TO_BYTE	BYTE	Takes the lower 8 bits and converts into BYTE type.
ULINT_TO_WORD	WORD	Takes the lower 16 bits and converts into WORD type.
ULINT_TO_DWORD	DWORD	Takes the lower 32 bits and converts into DWORD type.
ULINT_TO_LWORD	LWORD	Converts into LWORD type without changing the internal bit array.
ULINT_TO_BCD	BCD	If input is 0~9,999,999,999,999,999, normal conversion. Otherwise an error occurs.
ULINT_TO_REAL	REAL	Converts ULINT into REAL type. During the conversion, an error caused by the precision may occur.
ULINT_TO_LREAL	LREAL	Converts ULINT into LREAL type. During the conversion, an error caused by the precision may occur.

■ Error

If a conversion error occurs, _ERR and _LER flags will be set. If error occurs, it takes as many lower bits as a bit number of output type and produces an output without changing its internal bit array.

■ **Program Example**

LD	IL
	LD %M0 JMPN PP LD IN_VAL ULINT_TO_LINT ST OUT_VAL PP:

(1) If the input condition (%M0) is ON, function ULINT_TO_LINT will be executed.

(2) If input variable IN_VAL (ULINT) = 123,567,899, then output variable OUT_VAL (LINT) = 123,567,899.

Input (IN1): IN_VAL (ULINT) = 123,567,899

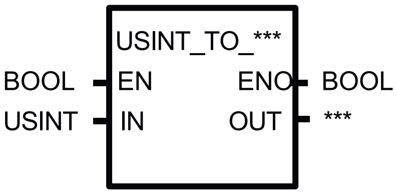
↓ (ULINT_TO_LINT)

Output (OUT): OUT_VAL (LINT) = 123,567,899

USINT_TO_***

USINT type conversion

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 USINT_TO_*** BOOL EN ENO BOOL USINT IN OUT ***	**Input** EN: executes the function in case of 1 IN: Unsigned Short Integer value to be converted **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ **Function**

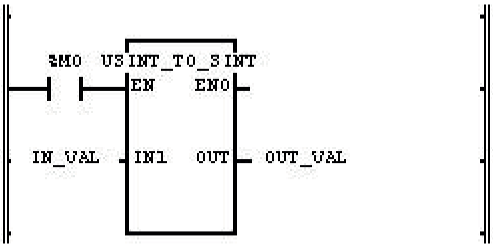
It converts the IN type and outputs it as OUT.

Function	Output type	Description
USINT_TO_SINT	SINT	If input is 0~127, normal conversion. Otherwise an error occurs.
USINT_TO_INT	INT	Converts USINT into INT type normally.
USINT_TO_DINT	DINT	Converts USINT into DINT type normally.
USINT_TO_LINT	LINT	Converts USINT into LINT type normally.
USINT_TO_UINT	UINT	Converts USINT into UINT type normally.
USINT_TO_UDINT	UDINT	Converts USINT into UDINT type normally.
USINT_TO_ULINT	ULINT	Converts USINT into ULINT type normally.
USINT_TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
USINT_TO_BYTE	BYTE	Converts into BYTE type without changing the internal bit array.
USINT_TO_WORD	WORD	Converts into WORD type filling the upper bits with 0.
USINT_TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
USINT_TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
USINT_TO_BCD	BCD	If input is 0 ~ 99, normal conversion. Otherwise an error occurs.
USINT_TO_REAL	REAL	Converts USINT into REAL type.
USINT_TO_LREAL	LREAL	Converts USINT into LREAL type.

■ **Error**

If a conversion error occurs, _ERR and _LER flags will be set. If error occurs, it takes as many lower bits as a bit number of output type and produces an output without changing its internal bit array.

■ **Program Example**

LD	IL
 %M0 USINT_TO_SINT EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN LL LD IN_VAL USINT_TO_SINT ST OUT_VAL LL:

(1) If the input condition (%M0) is ON, function ULINT_TO_SINT will be executed.

(2) If input variable IN_VAL (USINT) = 123, output variable OUT_VAL (SINT) = 123.

Input (IN1): IN_VAL (USINT) = 123 (16#7B)

0	1	1	1	1	0	1	1

↓ (ULINT_TO_SINT)

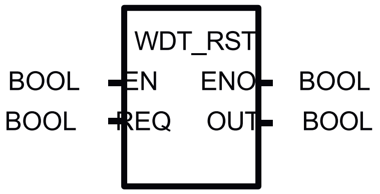
Output (OUT): OUT_VAL (SINT) = 123 (16#7B)

0	1	1	1	1	0	1	0

WDT_RST

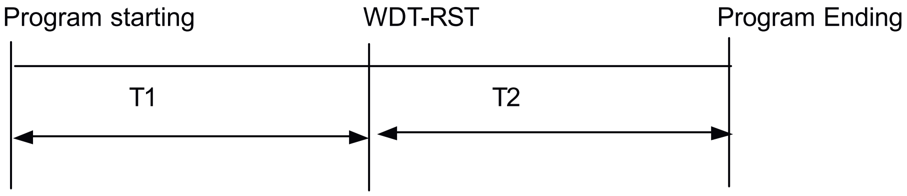
Initialize Watch_Dog timer

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
 WDT_RST BOOL EN ENO BOOL BOOL REQ OUT BOOL	**Input** EN: executes the function in case of 1 REQ: requires to initialize watchdog timer **Output** ENO: without an error, it will be 1 OUT: After Watch_Dog timer initialization, output will be 1.

■ **Function**

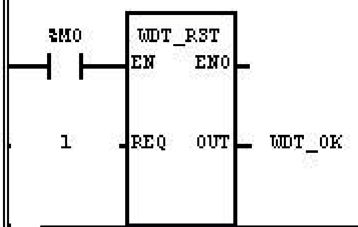
- • It resets Watch-Dog Timer among the programs.
- • Available to use in case that scan time exceeds Watch-Dog Time set by the condition in the program.
- • If scan time exceeds the scan Watch_Dog Time, please, change the scan time with the setting value of scan Watch_Dog Timer in the ‘ Basic Parameters’ of GMWIN.
- •Care must be taken so that either the time from 0 line of program to WDT_RST function T1 or the time from WDT_RST function to the time by the end of program T2 does not exceed the setting value of scan Watch_Dog Timer.



WDT_RST function is available to use several times during 1 scan.

■ **Program Example**

This is the program that the time to execute the program becomes 300ms according to the transition condition in the program of which scan Watch_Dog timer is set as 200ms.

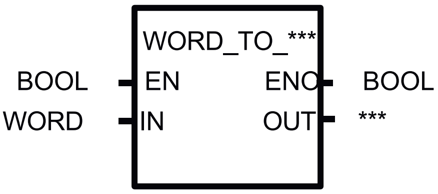
LD	IL
Program that has 300MS scan time. ↓ Program that has 150MS scan time.  Program that has 150MS scan time.	Program that has 300MS scan time. ↓ Program that has 150MS scan time. LD %M0 JMPN FG LD 1 WDT_RST ST WDT_OK Program that has 150MS scan time.

(1) If the transition condition (%M0) is ON, function WDT-RST will be executed.

(2) If WDT-RST function is executed, it is available to set the program that extends the scan time to 300ms according to the transition condition of program within the scan Watch_Dog Time (200mg).

WORD_TO_***

WORD type conversion

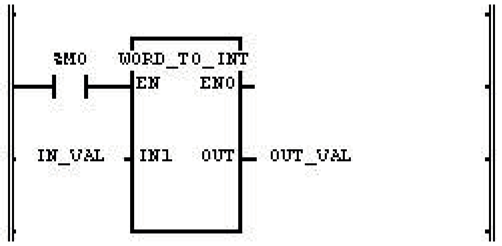
Function	Description
 WORD_TO_*** BOOL EN ENO BOOL WORD IN OUT ***	**Input** EN: executes the function in case of 1 IN: Bit string to be converted (16 bit) **Output** ENO: without an error, it will be 1 OUT: type-converted data

■ Function

It converts the IN type and outputs it as OUT.

Function	Output type	Description
WORD _TO_SINT	SINT	Takes the lower 8 bits and converts into SINT type.
WORD _TO_INT	INT	Converts into INT type without changing the internal bit array.
WORD _TO_DINT	DINT	Converts into DINT type filling the upper bits with 0.
WORD _TO_LINT	LINT	Converts into LINT type filling the upper bits with 0.
WORD _TO_USINT	USINT	Takes the lower 8 bits and converts into SINT type.
WORD _TO_UINT	UINT	Converts into INT type without changing the internal bit array.
WORD _TO_UDINT	UDINT	Converts into DINT type filling the upper bits with 0.
WORD _TO_ULINT	ULINT	Converts into LINT type filling the upper bits with 0.
WORD _TO_BOOL	BOOL	Takes the lower 1 bit and converts into BOOL type.
WORD _TO_BYTE	BYTE	Takes the lower 8 bits and converts into SINT type.
WORD _TO_DWORD	DWORD	Converts into DWORD type filling the upper bits with 0.
WORD _TO_LWORD	LWORD	Converts into LWORD type filling the upper bits with 0.
WORD _TO_DATE	DATE	Converts into DATE type without changing the internal bit array.
WORD _TO_STRING	STRING	Converts WORD into STRING type.

■ Program Example

LD	IL
 %M0 WORD_TO_INT EN ENO IN_VAL IN1 OUT OUT_VAL	LD %M0 JMPN P0 LD IN_VAL WORD_TO_INT ST OUT_VAL PO:

(1) If the input condition (%M0) is ON, function WORD-TO-INT will be executed.

(2) If input variable IN_VAL (WORD) = 2#0001_0001_0001_0001, output variable OUT_VAL (INT) = 4096 + 256 + 16 + 1 = 4,369.

Input (IN1): IN_VAL (WORD) = 16#1111

0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1

↓ (WORD-TO-INT)

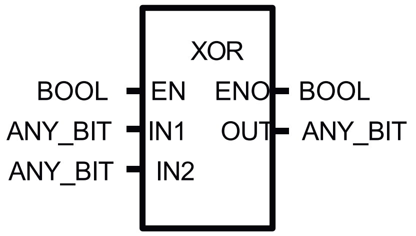
Output(OUT): OUT_VAL(INT) = 4,369 (16#1111)

0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1

XOR

Exclusive OR

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪	• ▪

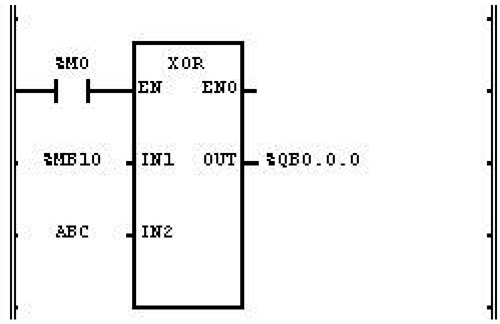
Function	Description
 XOR BOOL EN ENO BOOL ANY_BIT IN1 OUT ANY_BIT ANY_BIT IN2	**Input** EN: executes the function in case of 1 IN1: the value to be XOR IN2: the value to be XOR Input variable number can be extended up to 8. **Output** ENO: without an error, it will be 1. OUT: the result of XOR operation IN1, IN2, OUT should be all the same data type.

■ **Function**

Do XOR operation for IN1 and IN2 per bit and produces OUT.

IN1 1111 0000
XOR
IN2 1010 1010
OUT 0101 1010

 Program Example

LD	IL
 %M0 XOR EN ENO %MB10 IN1 OUT %QB0.0.0 ABC IN2	LD %M0 JMPN ZZ LD %MB10 XOR IN1:= CURRENT RESULT IN2:= ABC ST %QB0.0.0 ZZ:

(1) If the transition condition (%M0) is ON, function XOR will be executed.

(2) If input variable %MB10 = 11001100, ABC = 11110000, the result of XOR operation for two inputs will be %QB0.0.0 = 00111100.

Input (IN1): %MB10 (BYTE) = 16#CC	1	1	0	0	1	1	0	0
					(XOR)			
(IN2): ABC (BYTE) = 16#F0	1	1	1	1	0	0	0	0
					↓			
Output (OUT): %QB0.0.0 (BYTE) = 16#3C	0	0	1	1	1	1	0	0

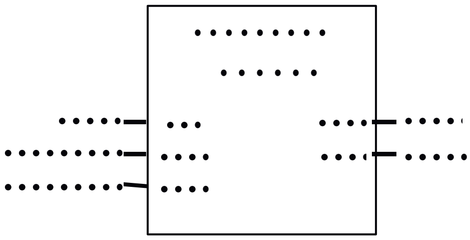
8.2 Application Function Library

This chapter describes application function library (MASTER-K and others).

ARY_ASC_TO_BCD

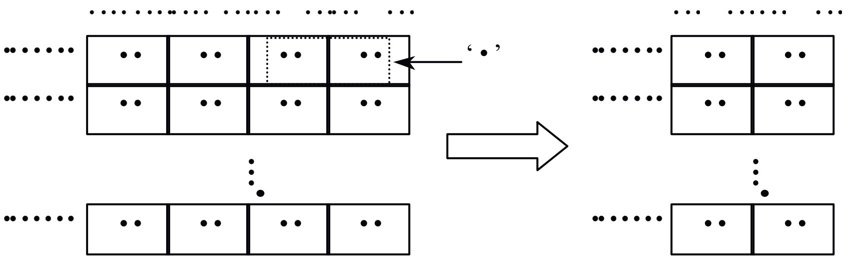
••••••••••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: ASCII Array input **Output** ENO: without an error, it will be 1 OUT: Dummy output **In/Out** IN2: BCD Array output

■ Function

It converts a word array input (ASCII data) to a byte array output (BCD data).

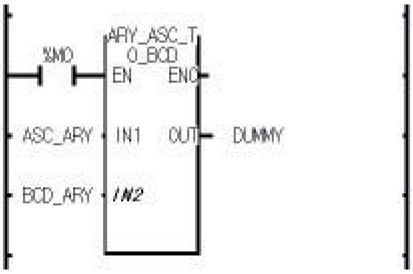


■ Error

- •If the number of each input array is different, there' s no change in IN2 data, and _ERR and _LER flags are set.
- •If the elements of IN1 array are not between 0 and 9 (hexadecimal), its responding elements of IN2 array are 16#00 (while other elements of IN1 are normally converted), and _ERR and _LER flags are set.

■ Program example

LD



(1) If the transition condition (%M0) is on, ARY_ASC_TO_BCD function is executed.
(2) If the input ASC_ARY data is:

ASC_ARY[0]	3031H
ASC_ARY[1]	3839H
ASC_ARY[2]	3334H

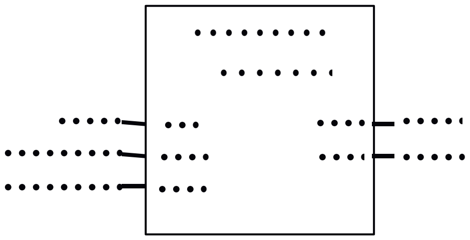
In/Out BCD_ARY data is as follows:

BYTE_ARY[0]	01H
BYTE_ARY[1]	89H
BYTE_ARY[2]	34H

ARY_ASC_TO_BYTE

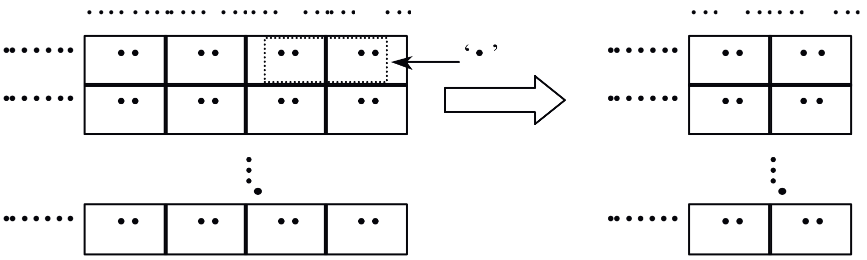
••••••••••••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: ASCII Array input **Output** ENO: without an error, it will be 1 OUT: Dummy Output **In/Out** IN2: BYTE Array Output

■ Function

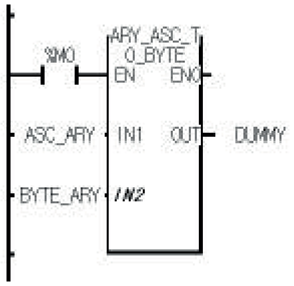
It converts a word array input (ASCII data) to a byte array output (hexadecimal).



■ Error

- •If the number of each input array is different, there' s no change in IN2 data, and _ERR and _LER flags are set.
- •If the elements of IN1 array are not between 0 and F (hexadecimal), its responding elements of IN2 array are 0 (while other elements of IN1 are normally converted), and _ERR and _LER flags are set.

■ Program example

LD


(1) If the transition condition is (%M0) is on, ARY_ASC_TO_BYTE function is executed.
(2) If Input ASC_ARY is as below:

ASC_ARY[0]	3441H
ASC_ARY[1]	3346H
ASC_ARY[2]	3239H

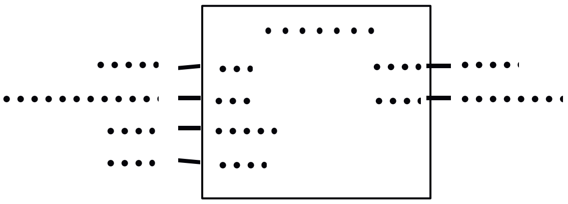
In/Out BYTE_ARY data is as follows:

BYTE_ARY[0]	4AH
BYTE_ARY[1]	3FH
BYTE_ARY[2]	29H

ARY_AVE_***

••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN: data array for average INDX: starting point to average in an array LEN: number of array elements for average **Output** ENO: without an error, it will be 1 OUT: average of an array

■ Function

- •ARY_AVE_*** function finds an average for a specified length of an array .
- •Input and output array is the same type.
- •If LEN is a minus value, it finds an average between INDX (Array index) and ‘ INDX – |LEN|’ .
- •Its output is rounded off.

Function	Output type	Description
ARY_AVE_SINT	SINT	Finds an average for SINT value (decimal is rounded off)
ARY_AVE_INT	INT	Finds an average for INT value (decimal is rounded off)
ARY_AVE_DINT	DINT	Finds an average for DINT value (decimal is rounded off)
ARY_AVE_LINT	LINT	Finds an average for LINT value (decimal is rounded off)
ARY_AVE_USINT	USINT	Finds an average for USINT value (decimal is rounded off)
ARY_AVE_UINT	UINT	Finds an average for UINT value (decimal is rounded off)
ARY_AVE_UDINT	UDINT	Finds an average for UDINT value (decimal is rounded off)
ARY_AVE_ULINT	ULINT	Finds an average for ULINT value (decimal is rounded off)
ARY_AVE_REAL	REAL	REAL.
ARY_AVE_LREAL	LREAL	LREAL.

■ Error

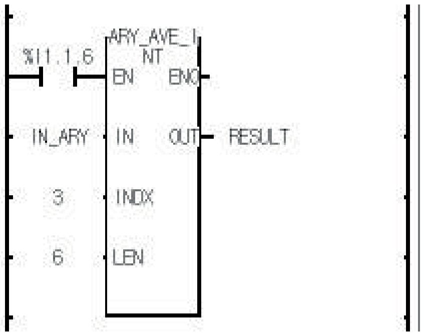
- •If it is designated beyond the array range, _ERR and _LER flags are set.
- •If an error occurs, the output is 0.

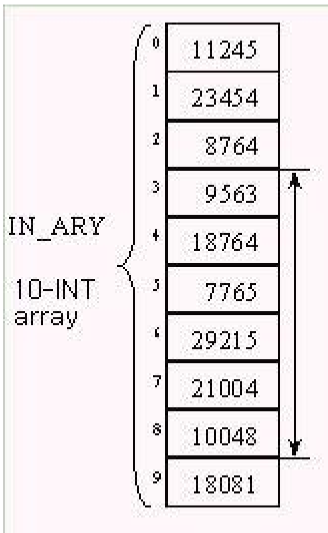
- •An error occurs when:

INDX < 0 or INDX > max. number of IN
INDX + LEN > max. number of IN

■ **Program example**

LD





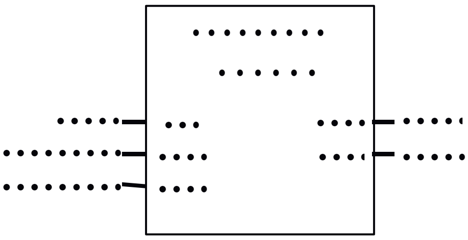
$$\frac{9563+18764+7765+29215+21004+10048}{6}=16044.83=16045$$

(1) If input transition condition (%I1.1.6) is on, ARY_AVE_INT function is executed.
(2) If an array is as the above, it finds an average between INDX 3 and 9.
(3) The output value is rounded off.

ARY_BCD_TO_ASC

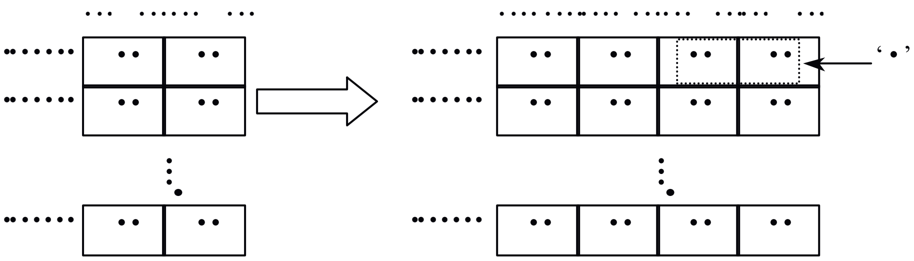
••••••••••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: BCD array input **Output** ENO: without an error, it will be 1 OUT: dummy output **In/Out** IN2: ASCII array output

■ Function

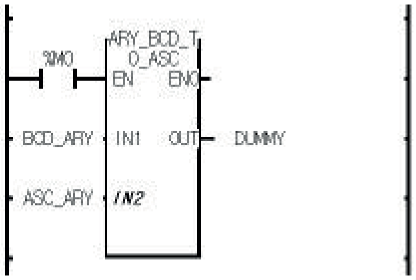
It converts a byte array input (BCD) to a word array (ASCII).



■ Error

- •If the number of each input array is different, there's no change in IN2 data, and _ERR and _LER flags are set.
- •If the elements of IN1 array are not between 0 and 9 (hexadecimal), its responding elements of IN2 array are 16#3030 ("00") (while other elements of IN1 are normally converted), and _ERR and _LER flags are set.

■ Program example

LD


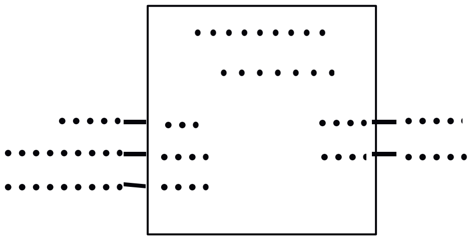
(1) If the transition condition (%M0) is on, ARY_BCD_TO_ASC function is executed.
(2) If the input BCD_ARY is as below:

BYTE_ARY[0]	01H
BYTE_ARY[1]	89H
BYTE_ARY[2]	45H

The In/out ASC_ARY is as follows:

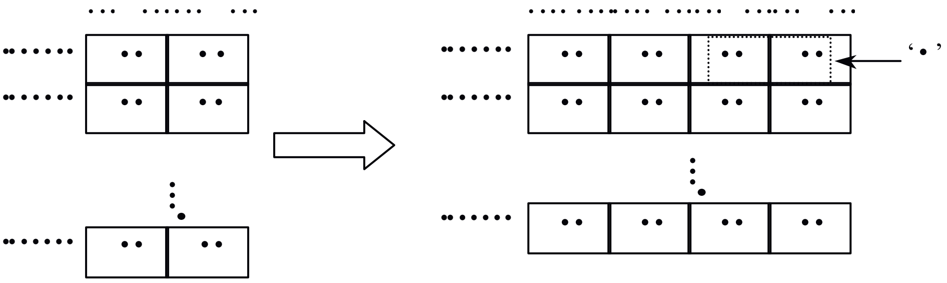
ASC_ARY[0]	3031H
ASC_ARY[1]	3839H
ASC_ARY[2]	3435H

ARY_BYTE_TO_ASC

Function	Description
	**Input** EN: executes the function in case of 1 IN1: BYTE array input **Output** ENO: without an error, it will be 1 OUT: Dummy output **In/Out** IN2: ASCII Array Output

■ Function

It converts a byte array input (HEX) to a word array (ASCII).

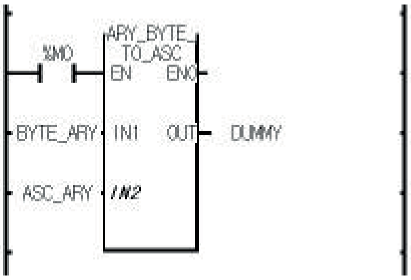


■ Error

If the number of each input array is different, there's no change in IN2 data, and _ERR and _LER flags are set.

■ Program example

LD



(1) If the transition condition (%M0) is on, ARY_BYTE_TO_ASC function is executed.
(2) If the input BYTE_ARY is as below:

BYTE_ARY[0]	4AH
BYTE_ARY[1]	3FH
BYTE_ARY[2]	29H

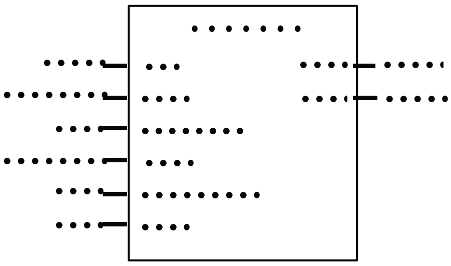
The output ASC_ARY is as follows:

ASC_ARY[0]	3441H
ASC_ARY[1]	3346H
ASC_ARY[2]	3239H

ARY_CMP_***

•••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	••	••	••	••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1 IN1: first array to compare IN1_INDX : starting point in 1st array for comparison IN2: second array to compare IN2_INDX : starting point in 2nd array for comparison LEN: number of elements to compare **Output** ENO: without an error, it will be 1 OUT: if two arrays are equal, it will be 1

■ Function

- •It compare two arrays whether they have the same value.
- •If LEN is minus, it compare two arrays between IN*_INDX (Array INDX) and “ Array INDX – |LEN|” .

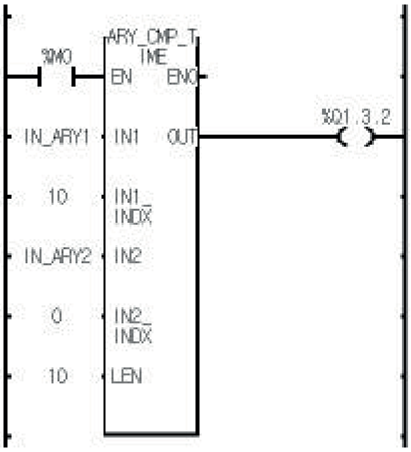
Function	Input array type	Description
ARY_CMP_BOOL	BOOL	Compares two BOOL Arrays.
ARY_CMP_BYTE	BYTE	Compares two BYTE Arrays.
ARY_CMP_WORD	WORD	Compares two WORD Arrays.
ARY_CMP_DWORD	DWORD	Compares two DWORD Arrays.
ARY_CMP_LWORD	LWORD	Compares two LWORD Arrays.
ARY_CMP_SINT	SINT	Compares two SINT Arrays.
ARY_CMP_INT	INT	Compares two INT Arrays.
ARY_CMP_DINT	DINT	Compares two DINT Arrays.
ARY_CMP_LINT	LINT	Compares two LINT Arrays.
ARY_CMP_USINT	USINT	Compares two USINT Arrays.
ARY_CMP_UINT	UINT	Compares two UINT Arrays.
ARY_CMP_UDINT	UDINT	Compares two UDINT Arrays.
ARY_CMP_ULINT	ULINT	Compares two ULINT Arrays.
ARY_CMP_REAL	REAL	Compares two REAL Arrays.
ARY_CMP_LREAL	LREAL	Compares two LREAL Arrays.
ARY_CMP_TIME	TIME	Compares two TIME Arrays.
ARY_CMP_DATE	DATE	Compares two DATE Arrays.
ARY_CMP_TOD	TOD	Compares two TOD Arrays.
ARY_CMP_DT	DT	Compares two DT Arrays.

■ **Error**

- •If it is designated beyond the array range, _ERR and _LER flags are set.

- •An error occurs when:
  IN1_INDX < 0 or IN1_INDX > max. number of IN1
  IN2_INDX < 0 or IN2_INDX > max. number of IN2
  IN1_INDX + LEN ≥ max. number of IN1
  IN2_INDX + LEN ≥ max. number of IN2

■ **Program example**

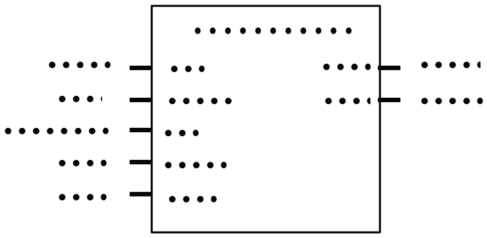
LD


(1) If the input transition condition (%M0) is on, ARY_CMP_TIME function is executed.
(2) When IN_ARY1 is a time array with 100 elements and IN_ARY2 is a time array with 10 elements, if the elements from 11th to 20th of IN_ARY1 and the elements of IN_ARY 2 are equal, the output %Q1.3.2 is on.

ARY_FLL_***

••••••••••••••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 DATA: the data to fill an array INDX: starting point of an array to be filled LEN: number of array elements to be filled **Output** ENO: without an error, it will be 1 OUT: without an error, it will be 1 **In/Out** IN: an array to be filled

■ Function

- •It fills an array with the input data.
- •If LEN is minus, it fills an array from INDX to “ INDX – |LEN|” .

Function	In/Out Array type	Description
ARY_FLL_BOOL	BOOL	Fills a BOOL Array with the input data.
ARY_FLL_BYTE	BYTE	Fills a BYTE Array with the input data.
ARY_FLL_WORD	WORD	Fills a WORD Array with the input data.
ARY_FLL_DWORD	DWORD	Fills a DWORD Array with the input data.
ARY_FLL_LWORD	LWORD	Fills a LWORD Array with the input data.
ARY_FLL_SINT	SINT	Fills a SINT Array with the input data.
ARY_FLL_INT	INT	Fills a INT Array with the input data.
ARY_FLL_DINT	DINT	Fills a DINT Array with the input data.
, ARY_FLL_LINT	LINT	Fills a LINT Array with the input data.
ARY_FLL_USINT	USINT	Fills a USINT Array with the input data.
ARY_FLL_UINT	UINT	Fills a UINT Array with the input data.
ARY_FLL_UDINT	UDINT	Fills a UDINT Array with the input data.
ARY_FLL_ULINT	ULINT	Fills a ULINT Array with the input data.
ARY_FLL_REAL	REAL	Fills a REAL Array with the input data.
ARY_FLL_LREAL	LREAL	Fills a LREAL Array with the input data.
ARY_FLL_TIME	TIME	Fills a TIME Array with the input data.
ARY_FLL_DATE	DATE	Fills a DATE Array with the input data.
ARY_FLL_TOD	TOD	Fills a TOD Array with the input data.
ARY_FLL_DT	DT	Fills a DT Array with the input data.

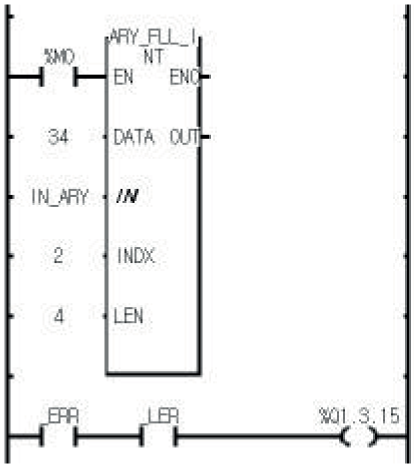
■ Error

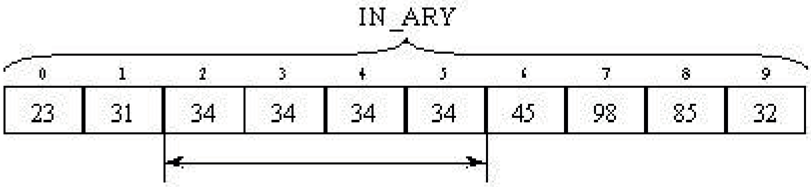
- •If it is designated beyond the array range, _ERR and _LER flags are set.
- •If an error occurs, there' s no change in arrays and OUT is off.

- •An error occurs when:
  INDX < 0 or INDX > max. element number of IN
  INDX + LEN ≥ max. element number of IN

■ Program example

LD





0	1	2	3	4	5	6	7	8	9
23	31	34	34	34	34	45	98	85	32

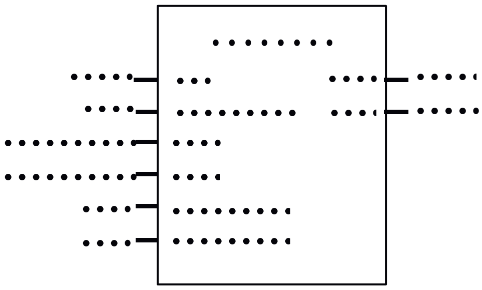
Fills 4 elements starting from INDX.

(1) If input condition (%M0) is on, ARY_FLL_INT function is executed.
(2) It fills 4 elements of IN_ARY starting from INDX with 34.
(3) If LEN is 9, it is beyond the array range and an error occurs; _ERR and _LER flags are set and the output (%Q1.13.15) is on.

ARY_MOVE

••••••••••• •

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

• • • •	• • • •
	**Input** EN : executes the function in case of 1 MOVE_NUM: array number to move IN1: array variable to move (STRING type, unavailable) IN2: array variable to be moved (STRING type, unavailable) IN1_INDX: starting pointer of array to move IN2_INDX: starting pointer of array to be moved **Output** ENO: without an error, it will be 1 OUT: without an error, it will be 1

■ Function

- •If EN is 1, it moves IN1 data to IN2.
- •It copies MOVE_NUM elements of IN1 (from IN1_INDX) and pastes it in IN2 (from IN2_INDX).
- •IN1 and IN2 are the same data type (The number of each array can be different).
- •The data size is as follows:

Data size	Variable type
1 Bit	BOOL
8 Bit	BYTE, SINT, USINT
16 Bit	WORD / INT / UINT / DATE
32 Bit	DWORD / DINT / UDINT / TIME / TOD
64 Bit	DT

■ Error

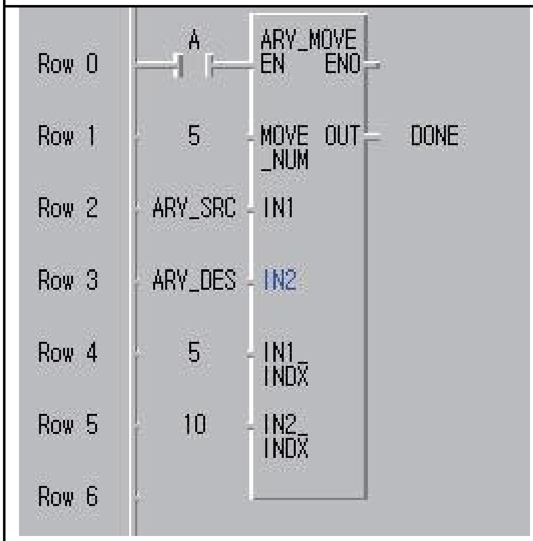
- •An error occurs when IN1 and IN2 data size are different.
- •An error occurs when:

1) the array number of IN1 Array < (IN1_INDX + MOVE_NUM)
2) the array number of IN2 Array < (IN2_INDX + MOVE_NUM)

Then ARY_MOVE function is not executed, OUT is 0, ENO is off and _ERR and _LER flags are set.

■ **Program example**

LD



Variable name	Variable type	Array number
ARY_SRC	INT	10
ARY_DES	WORD	15

(1) If the transition condition (A) is on, ARY_MOVE function is executed.

(2) It moves 5 elements from ARY_SRC[5] to ARY_DES[10].

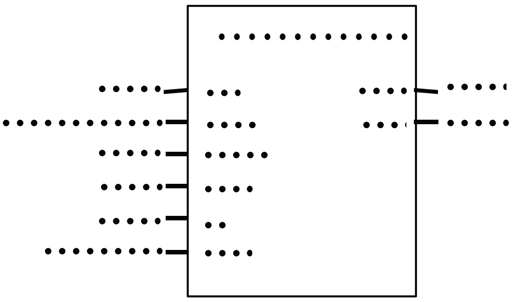
Now the data type of ARY_DES is WORD, it' s hexadecimal.

Before				After			
ARY_SRC[0]	0	ARY_DES[0]	16#0	ARY_SRC[0]	0	ARY_DES[0]	16#0
ARY_SRC[1]	11	ARY_DES[1]	16#1	ARY_SRC[1]	11	ARY_DES[1]	16#1
ARY_SRC[2]	22	ARY_DES[2]	16#2	ARY_SRC[2]	22	ARY_DES[2]	16#2
ARY_SRC[3]	33	ARY_DES[3]	16#3	ARY_SRC[3]	33	ARY_DES[3]	16#3
ARY_SRC[4]	44	ARY_DES[4]	16#4	ARY_SRC[4]	44	ARY_DES[4]	16#4
ARY_SRC[5]	55	ARY_DES[5]	16#5	ARY_SRC[5]	55	ARY_DES[5]	16#5
ARY_SRC[6]	66	ARY_DES[6]	16#6	ARY_SRC[6]	66	ARY_DES[6]	16#6
ARY_SRC[7]	77	ARY_DES[7]	16#7	ARY_SRC[7]	77	ARY_DES[7]	16#7
ARY_SRC[8]	88	ARY_DES[8]	16#8	ARY_SRC[8]	88	ARY_DES[8]	16#8
ARY_SRC[9]	99	ARY_DES[9]	16#9	ARY_SRC[9]	99	ARY_DES[9]	16#9
		ARY_DES[10]	16#A			ARY_DES[10]	16#37
		ARY_DES[11]	16#B			ARY_DES[11]	16#42
		ARY_DES[12]	16#C			ARY_DES[12]	16#4D
		ARY_DES[13]	16#D			ARY_DES[13]	16#58
		ARY_DES[14]	16#E			ARY_DES[14]	16#63

ARY_ROT_C_***

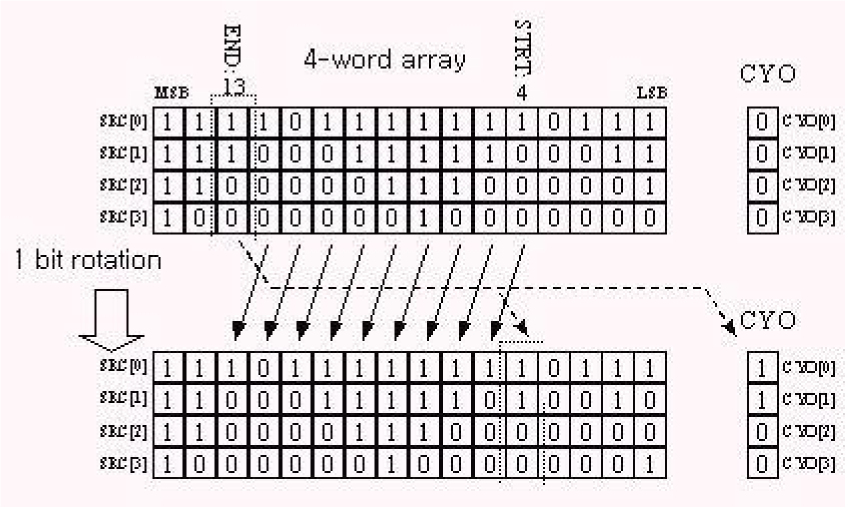
....
.

....	....	....	....	....	....	....	....
....	. .	. .	. .	. .	. .	. .	. .

Function	Description
	**Input** EN: executes the function in case of 1 STRT: starting bit to rotate END: ending bit to rotate N: number to rotate **Output** ENO: without an error, it will be 1 OUT: without an error, it will be 1 **In/Out** SRC: Source Array to rotate CYO: output Carry bit Array

■ **Function**

- •It rotates as many bits of array elements as they' re specified.
- • Setting:
  - Scope: it sets a rotation scope with STRT and END.
  - Rotation direction and time: it rotates N times from STRT to END.
  - Output: the result is stored in ANY_BIT_ARY and a bit array data from END to STRT is written at CYO.



																		CYO	
SRC[0]	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1		0	CYO[0]
SRC[1]	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	1		0	CYO[1]
SRC[2]	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1		0	CYO[2]
SRC[3]	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		0	CYO[3]

. ...

																		CYO	
SRC[0]	1	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1		1	CYO[0]
SRC[1]	1	1	0	0	0	1	1	1	1	1	0	1	0	0	1	0		1	CYO[1]
SRC[2]	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0		0	CYO[2]
SRC[3]	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1		0	CYO[3]

. .....

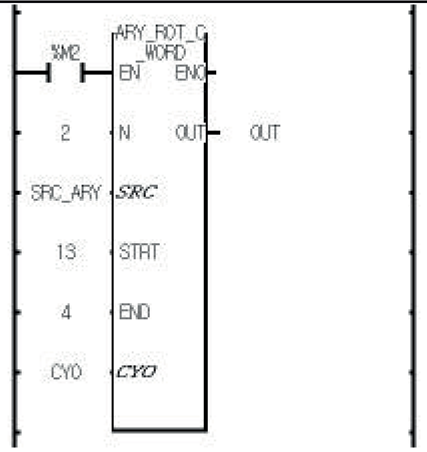
Function	In/out Array type	Description
ARY_ROT_C_BYTE	BYTE	It rotates elements of an array as many bits as they' re specified.
ARY_ROT_C_WORD	WORD	
ARY_ROT_C_DWORD	DWORD	
ARY_ROT_C_LWORD	LWORD	

■ Error

- •If the number of SRC and CYO Arrays are different, _ERR and _LER flags are set.
- •If STRT and END are out of bit range of SRC, an error occurs.
- •When an error occurs, there' s no change in SRC and CYO.

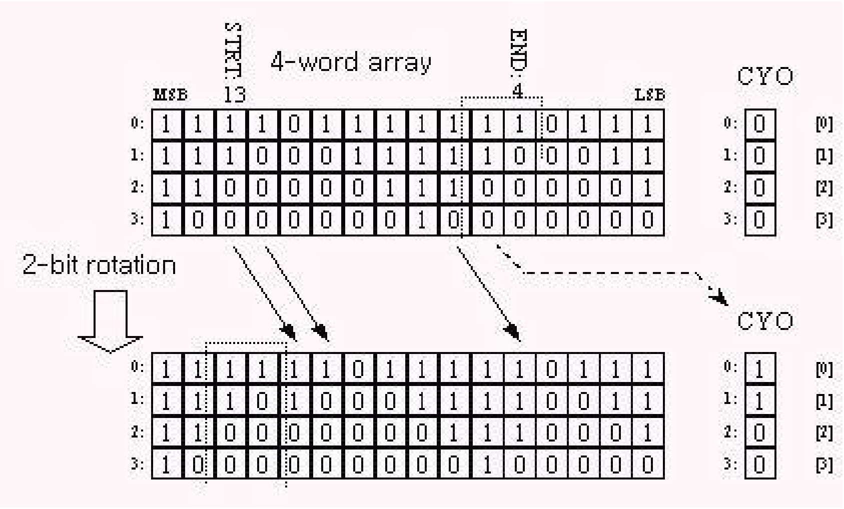
■ Program example

LD



(1) If the input condition (%M2) is on, ARY_ROT_C_WORD function is executed.
(2) It rotates 2 times the bit (from 4 to 13 bit) arrays of SRC_ARY from STRT to END.
(3) The result is stored at SRC_ARY and the carry bit arrays are written in CYO BOOL Array.

(Before)
SRC_ARY : 16#F7F7
16#E3E3
16#C1C1
16#8080
(N) : 2
(After)
SRC_ARY : 16#FDF7
16#E8F3
16#C071
16#8020
CYO : 2#1100



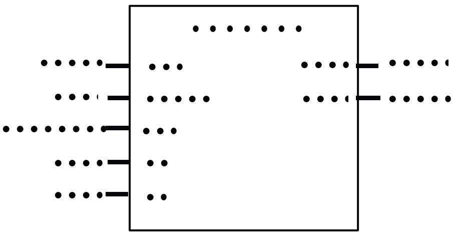
																	CYO	
0:	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0: 0	[0]
1:	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	1	1: 0	[1]
2:	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	2: 0	[2]
3:	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	3: 0	[3]

																	CYO	
0:	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	0: 1	[0]
1:	1	1	1	0	1	0	0	0	1	1	1	1	0	0	1	1	1: 1	[1]
2:	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	2: 0	[2]
3:	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	3: 0	[3]

ARY_SCH_***

•••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	••	••	••	••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1 DATA: data to search IN: array to search **Output** ENO: without an error, it will be 1 OUT: if it finds, it will be 1 **In/Out** P: first position of an object array N: total number of array elements equal to an object

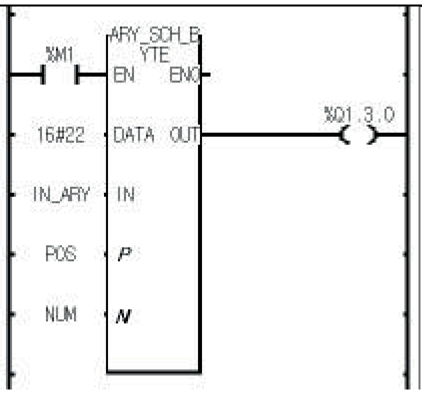
■ Function

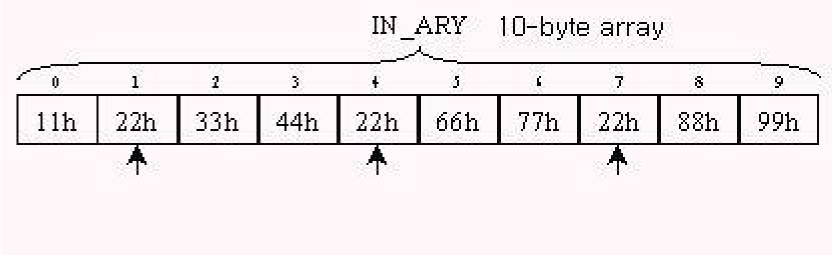
It finds an equal value of input in arrays and produces its first position and total number. When it finds at least one which is equal to an object in arrays, OUT is 1.

Function	Input Array type	Description
ARY_SCH_BOOL	BOOL	Search in BOOL Array.
ARY_SCH_BYTE	BYTE	Search in BYTE Array.
ARY_SCH_WORD	WORD	Search in WORD Array.
ARY_SCH_DWORD	DWORD	Search in DWORD Array.
ARY_SCH_LWORD	LWORD	Search in LWORD Array.
ARY_SCH_SINT	SINT	Search in SINT Array.
ARY_SCH_INT	INT	Search in INT Array.
ARY_SCH_DINT	DINT	Search in DINT Array.
ARY_SCH_LINT	LINT	Search in LINT Array.
ARY_SCH_USINT	USINT	Search in USINT Array.
ARY_SCH_UINT	UINT	Search in UINT Array.
ARY_SCH_UDINT	UDINT	Search in UDINT Array.
ARY_SCH_ULINT	ULINT	Search in ULINT Array.
ARY_SCH_REAL	REAL	Search in REAL Array.
ARY_SCH_LREAL	LREAL	Search in LREAL Array.
ARY_SCH_TIME	TIME	Search in TIME Array.
ARY_SCH_DATE	DATE	Search in DATE Array.
ARY_SCH_TOD	TOD	Search in TOD Array.
ARY_SCH_DT	DT	Search in DT Array.

■ **Program example**

LD





0	1	2	3	4	5	6	7	8	9
11h	22h	33h	44h	22h	66h	77h	22h	88h	99h

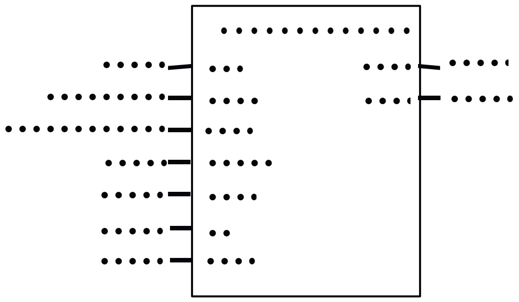
(1) If the input condition (%M1) is on, ARY_SCH_BYTE function is executed.
(2) When IN_ARY is a 10-byte array, if you search for “22h” in this array, three bytes are found as the above.
(3) The result is: 1) 1, the first position of an array, is stored at POS; 2) 3, the total number, is stored at NUM. The total number is 3, so the output %Q1.3.0 is on.

•

ARY_SFT_C_***

•••••••••••••••••••••••••••••••• •

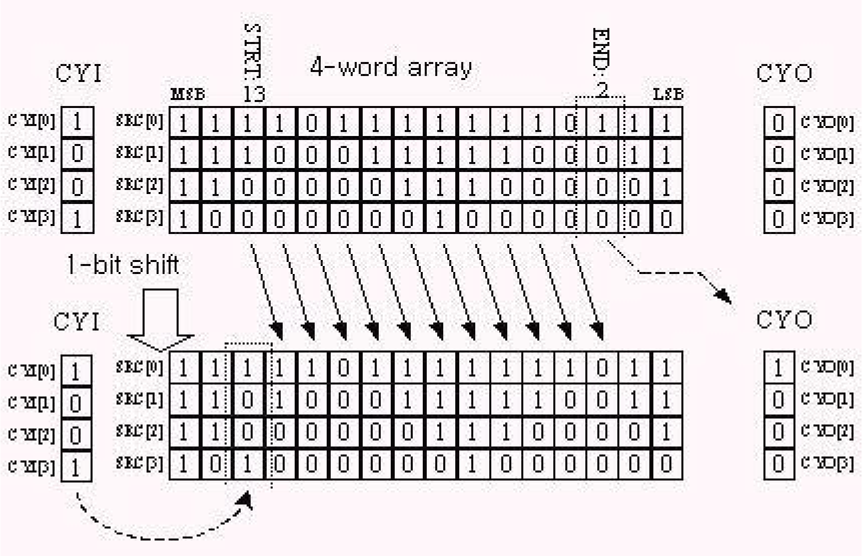
•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 CYI: Input Carry bit Array STRT: starting bit to shift END: ending bit to shift N: bit number to shift **Output** ENO: without an error, it will be 1 OUT: without an error, it will be 1 **In/Out** SRC: Source Array to shift CYO: Output Carry bit Array after shift

■ Function

- •It shifts as many bits of array elements as they’ re specified.
- • Setting:
  - Scope: it sets a shifting scope with STRT and END.
  - Shifting direction and time: it shifts N times from STRT to END.
  - Input data: it fills the empty bits with input data (CYI).

- Output: the result is stored in ANY_BIT_ARY and an overflowing bit array data from END is written at CYO.



CYI		SRC																	CYO	
CYI[0]	1	SRC[0]	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	0	CYO[0]
CYI[1]	0	SRC[1]	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	1	0	CYO[1]
CYI[2]	0	SRC[2]	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	CYO[2]
CYI[3]	1	SRC[3]	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	CYO[3]

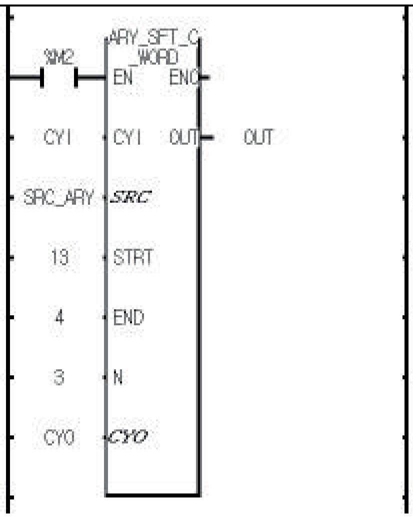
CYI		SRC																	CYO	
CYI[0]	1	SRC[0]	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	CYO[0]
CYI[1]	0	SRC[1]	1	1	0	1	0	0	0	1	1	1	1	1	0	0	1	1	0	CYO[1]
CYI[2]	0	SRC[2]	1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0	CYO[2]
CYI[3]	1	SRC[3]	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	CYO[3]

Function	In/Out Array type	Description
ARY_SFT_C_BYTE	BYTE	It shifts as many bits of array elements as they’ re specified.
ARY_SFT_C_WORD	WORD	
ARY_SFT_C_DWORD	DWORD	
ARY_SFT_C_LWORD	LWORD	

■ **Error**

- •If the number of CYI, SRC and CYO Array are different, _ERR and _LER flags are set.
- •An error occurs if STRT and END are out of SRC range.
- •When an error occurs, there' s no change in SRC and CYO.

■ **Program example**

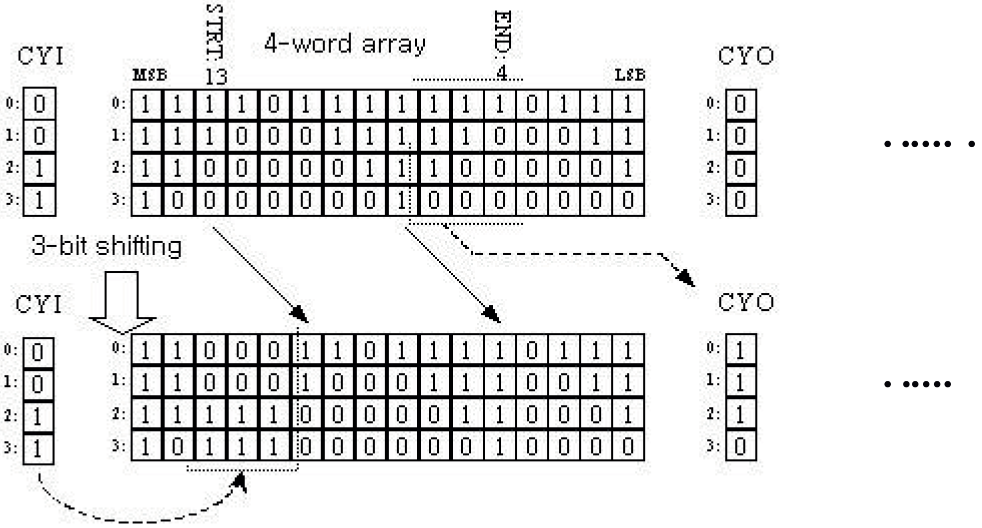
LD
 %M2 ARY_SFT_C _WORD EN ENO CYI CYI OUT OUT SRC_ARY SRC 13 STRT 4 END 3 N CYO CYO

(1) If input condition (%M2) is on, ARY_SFT_C_WORD function is executed.
(2) It shifts a bit array (from 4 to 13 bit) of SRC 3 times from STRT to END.
(3) The bit array after shifting is filled with CYI (2#0011).
(4) It produces its shifting result at SRC_ARY and a carry bit array is written at CYO.

(Before)
CYI: 2#0011
SRC_ARY: 16#F7F7
16#E3E3
16#C1C1
16#8080

(N): 3

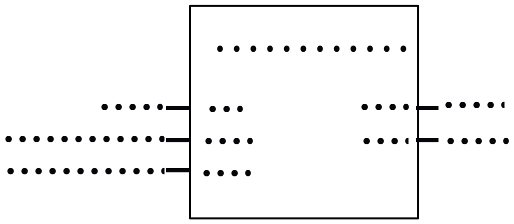
(After)
SRC_ARY: 16#C6F7
16#C473
16#F831
16#B810
CYO: 2#1110



ARY_SWAP_***

••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1 IN1: array input **Output** ENO: without an error, it will be 1 OUT: Dummy output **In/Out** IN2: array output after swapping

■ **Function**

It swaps upper/lower elements after dividing an array.

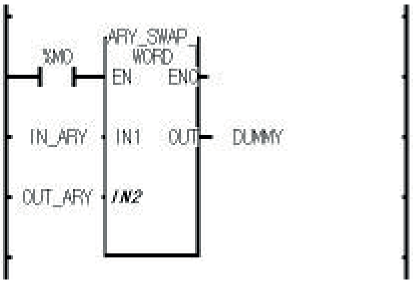
Function	Input type	Description
ARY_SWAP_BYTE	BYTE	Swaps upper/lower nibble of byte elements.
ARY_SWAP_WORD	WORD	Swaps upper/lower byte of WORD elements.
ARY_SWAP_DWORD	DWORD	Swaps upper/lower WORD of DWORD elements.
ARY_SWAP_LWORD	LWORD	Swaps upper/lower DWORD of LWORD elements.

■ **Error**

_ERR and _LER flags are set if two arrays are different; there' s no change in an IN2 array.

■ **Program example**

LD



(1) If the transition condition (%M0) is on, ARY_SWAP_WORD function is executed.

(2) If IN_ARY data is as below:

IN_ARY[0]	12ABH
IN_ARY[1]	23BCH
IN_ARY[2]	34CDH

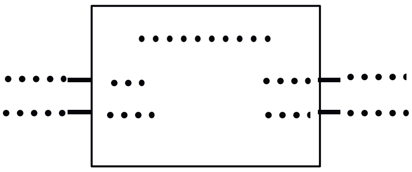
OUT_ARY data is as follows:

OUT_ARY[0]	AB12H
OUT_ARY[1]	BC23H
OUT_ARY[2]	CD34H

ASC_TO_BCD

••••••••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: ASCII input **Output** ENO: without an error, it will be 1 OUT: BCD output

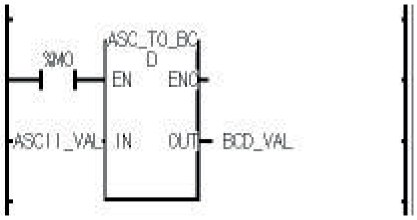
■ **Function**

It converts two ASCII data into two-digit BCD (Binary Coded Decimal) data.

■ **Error**

If IN is not hexadecimal number between 0 ~ 9, the output is 16#00 and _ERR and _LER flags will be set.

■ **Program example**

LD
 %M0 ASC_TO_BCD EN ENO ASCII_VAL IN OUT BCD_VAL

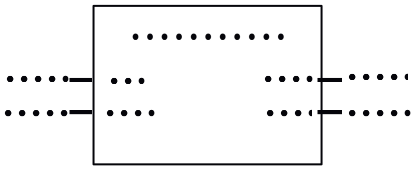
(1) If the transition condition (%M0) is on, ASC_TO_BCD function is executed.

(2) If input variable ASCII_VAL (WORD) = 16#3732 = “ 72” , output variable BCD_VAL (BYTE) = 16#72.

ASC_TO_BYTE

••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN : executes the function in case of 1. IN : ASCII input **Output** ENO : without an error, it will be 1 OUT : BYTE Output

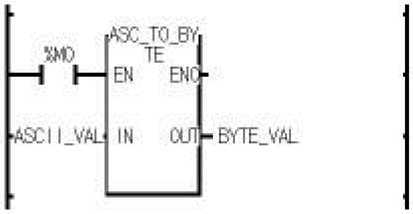
■ Function

It converts two ASCII data to 2-digit hexadecimal (HEX).

■ Error

If IN is not between ‘ 0’ and ‘ F’ , its output is 0 and _ERR/_LER flags are set.

■ Program example

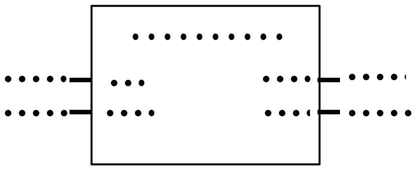
LD


(1) If the transition condition (%M0) is on, ASC_TO_BYTE function is executed.
(2) If input ASCII_VAL (WORD) = 16#4339, output BYTE_VAL (BYTE) = 16#C9.

BCD_TO_ASC

••••••••••••••••••••••••••• •

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: BCD input **Output** ENO: without an error, it will be 1 OUT: ASCII Output

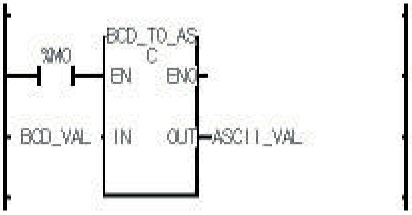
■ **Function**

It converts two BCD data to two ASCII data.

■ **Error**

If IN is not between 0 and 9, its output is 16#3030 (“ 00”) and _ERR/_LER flags are set.

■ **Program example**

LD
 %M0 BCD_TO_AS C EN ENO BCD_VAL IN OUT ASCII_VAL

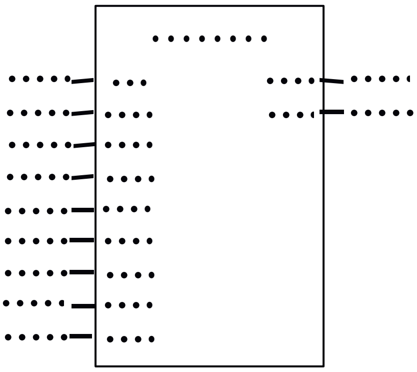
(1) If the transition condition (%M0) is on, BCD_TO_ASC function is executed.

(2) If input BCD_VAL (BYTE) = 16#85, output ASCII_VAL (WORD) = 16#3835 = “ 85” .

BIT_BYTE

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

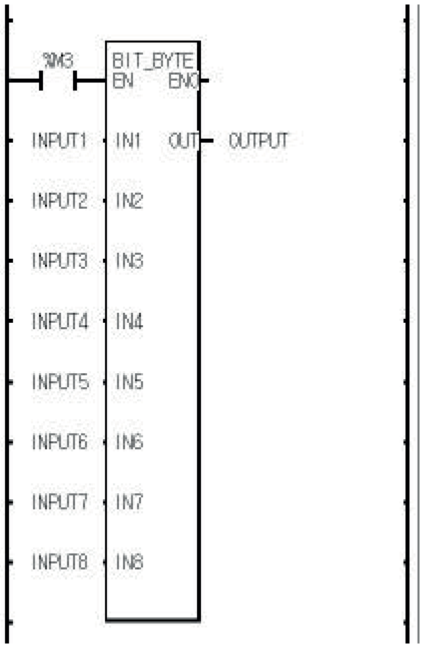
Function	Description
	**Input** EN: executes the function in case of 1. IN1 ~ IN8: Bit input **Output** ENO: without an error, it will be 1 OUT: Byte output

■ **Function**

It combines 8 bits into one byte.
IN8: MSB (Most Significant Bit), IN1: LSB (Least Significant Bit)

■ **Program example**

LD

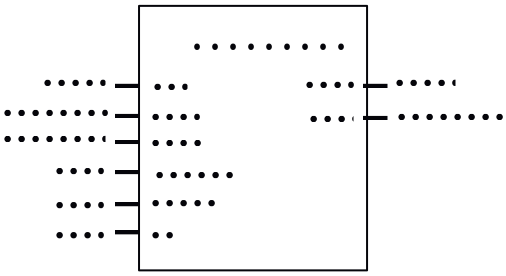


(1) If the transition condition (%M3) is on, BIT_BYTE function is executed.
(2) If 8 input are (from INPUT1 to INPUT 8) {0,1,1,0,1,1,0,0}, OUTPUT (BYTE) = 2#00110110.

BMOV_***

Moves part of a bit string•
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN : executes the function in case of 1. IN1: String data having bit data to be combined IN2: String data having bit data to be combined IN1_P: Start bit position on IN1 set data IN2_P: Start bit position on IN2 set data N: Bit number to be combined **Output** ENO: without an error, it will be 1 OUT: Combined bit string data output

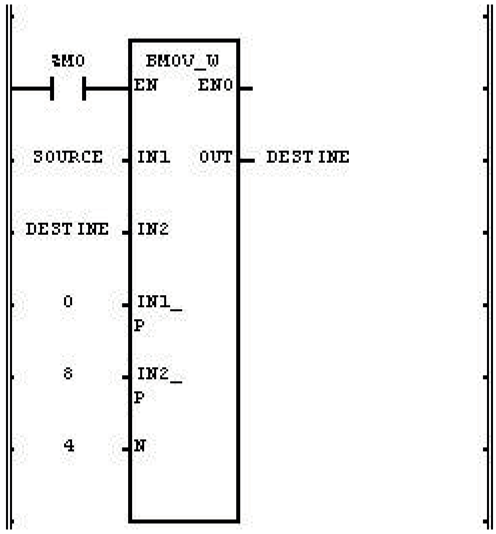
■ Function

- If EN is 1, it takes N bits of IN1 starting from the IN1_P bit and moves it to IN2 starting from IN2_P bit.
- If N1 = 1111 0000 1111 0000, IN2 = 0000 1010 1010 1111, IN1_P = 4, IN2_P = 8, N = 4, then output data is 0000 1111 1010 1111. Input data types are B (BYTE), W (WORD), D (DWORD), L (LWORD); L (LWORD) are available for GM1/2. You can use one of functions (' ENCO_B' , ' ENCO_W' , ' ENCO_D' , ' ENCO_L') according to input data.

■ Error

If IN1_P and IN2_P exceed the data range or N is negative or N bit of IN1_P and IN2_P exceeds the data range, _ERR and _LER flags are set.

■ Program example

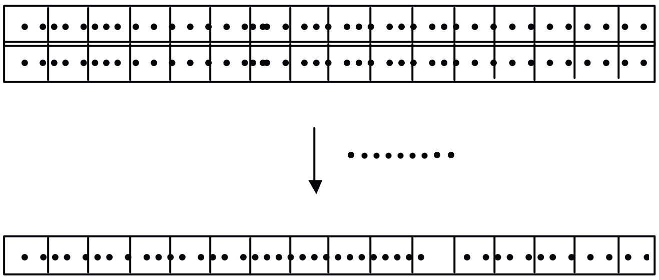
LD	IL
	LD %M0 JMPN LSB LD SOURCE BMOV_W IN1:= CURRENT RESULT IN2:= DESTINE IN1_P:= 0 IN2_P:= 8 N:= 4 ST DESTINE LSB :

(1) If the transition condition (%M0) is on, BMOV_W function is executed.

(2) If input SOURCE = 2#0101 1111 0000 1010, DESTINE = 2#0000 0000 0000 0000, IN1_ P = 0, IN2_P = 8, N = 4, then the result DESTINE is 2#0000 1010 0000 0000.

Input (IN1): SOURCE (WORD) = 16#5F0A
(IN2): DESTINE (WORD) = 16#0000
(IN1_ P) = 0
(IN2_P) = 8
(N) = 4

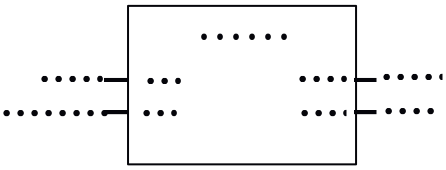
Output (OUT): DESTINE (WORD) = 16#0A00



BSUM_***

••••••••••••••••••••••••••••••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

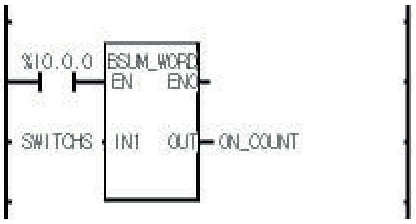
Function	Description
	**Input** EN: executes the function in case of 1. IN: input data to detect ON bit **Output** ENO: without an error, it will be 1 OUT: Result data (sum of on-bit number)

■ Function

If EN is 1, it counts bit number of 1 among IN bit string and produces output OUT. Input data types are BYTE, WORD, DWORD, LWORD. LWORD is available only for GM1/2.

FUNCTION	IN type	Description
BSUM_BYTE	BYTE	You can select one of these functions according to input data.
BSUM _WORD	WORD	
BSUM _DWORD	DWORD	
BSUM _LWORD	LWORD	

Program example

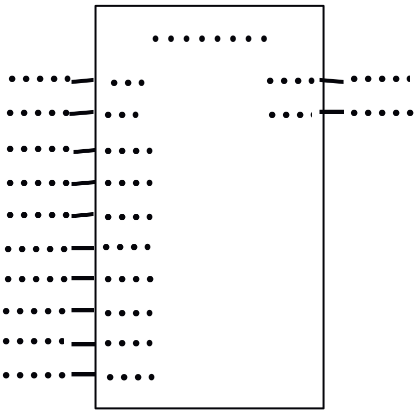
LD	IL
 %I0.0.0 BSUM_WORD EN ENO SWITCHS IN1 OUT ON_COUNT	LD %I0.0.0 JMPN AAA LD SWITCHS BSUM_WORD ST ON_COUNT AAA:

(1) If the transition condition (%M0) is on, BSUM_WORD function is executed.
(2) If input SWITCHS (WORD) = 2#0000 0100 0010 1000, then it counts on-bit number, 3. So the output ON_COUNT (INT) = 3.

BYTE_BIT

••••••••••••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: byte input **Output** ENO: without an error, it will be 1 OUT: Dummy output **In/Out** QO1~8: bit output

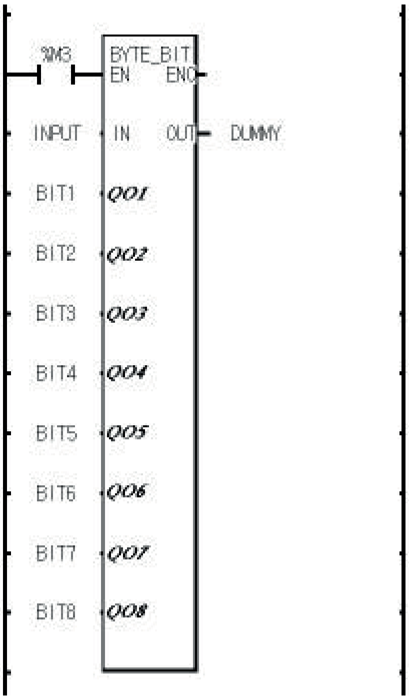
■ Function

It divides one byte into 8 bits (QO1~QO2).

QO8: MSB (Most Significant Bit), QO1: LSB (Least Significant Bit)

■ Program example

LD



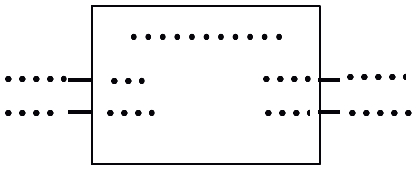
(1) If the transition condition (%M0) is on, BYTE_BIT function is executed.

(2) If INPUT = 16#AC = 2#10101100, it distributes INPUT from Q01 to Q08 in order.
The order is 2#{0, 0, 1, 1, 0, 1, 0, 1}.

BYTE_TO_ASC

••••••••••••••••••••••••••   •

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

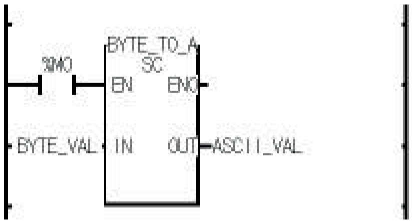
Function	Description
	**Input** EN: executes the function in case of 1. IN: BYTE input **Output** ENO: without an error, it will be 1 OUT: ASCII output

■ Function

- •It converts 2-digit hexadecimal into two ASCII data.
  Ex) 16#12 -> 3132
- •In case of 16#A~F, it produces ASCII data for character.

■ Program example

LD

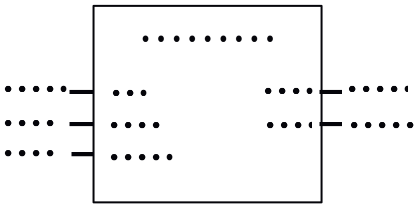


(1) If the transition condition (%M0) is on, BYTE_TO_ASC function is executed.
(2) If input BYTE_VAL (BYTE) = 16#3A, output ASCII_VAL (WORD) = 16#3341 = ‘ 3’ , ‘ A’ .

BYTE_WORD

•••••••••••••••••••••••••••
•

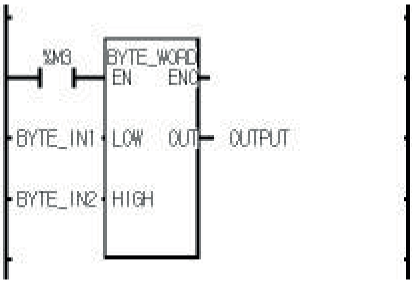
•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	••	••	••	••	••	••	••

Function	Description
	**Input** EN: executes the function in case of 1. LOW: lower BYTE Input HIGH: upper BYTE Input **Output** ENO: without an error, it will be 1 OUT: WORD output

■ Function

It combines two bytes into one word.
LOW: lower byte input, HIGH: upper byte input

■ Program example

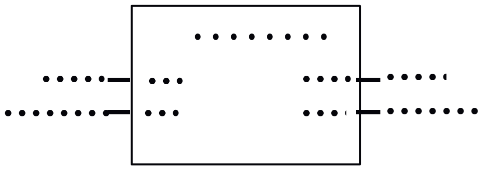
LD


(1) If the transition condition (%M3) is on, BYTE_WORD function is executed.
(2) If input BYTE_IN1 = 16#56 and BYTE_IN2 = 16#AD, output variable OUTPUT = 16#AD56.

DEC_***

•••••••••••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	• • • •
	**Input** EN: executes the function in case of 1. IN: input data to decrease **Output** ENO: without an error, it will be 1 OUT: result data

Function

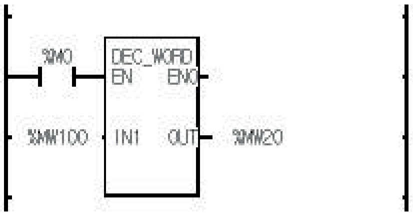
If EN is 1, it produces an output after decreasing bit-string data of IN by 1.

Even though the underflow occurs, an error won' t occur and if the result is 16#0000, then the output result data is 16#FFFF.

Input data types are BYTE, WORD, DWORD and LWORD. LWORD is available only for GM1/2.

FUNCTION	IN/OUT type	Description
DEC_BYTE	BYTE	You can select one of these functions according to in/out data type.
DEC_WORD	WORD	
DEC_DWORD	DWORD	
DEC_LWORD	LWORD	

Program example

LD	IL
 %M0 DEC_WORD EN ENO %MW100 IN1 OUT %MW20	LD %M0 JMPN KKK LD %MW100 DEC_WORD ST %MW20 KKK:

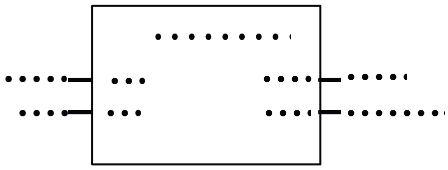
(1) If the transition condition (%M0) is on, DEC_WORD function is executed.

(2) If input variable %MW100 = 16#0007 (2#0000 0000 0000 0111), output variable %MW20 = 16#0006 (2#0000 0000 0000 0110).

DECO_***

Decodes the designated bit position•
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: input data for decoding **Output** ENO: without an error, it will be 1 OUT: decoding result data

 Function

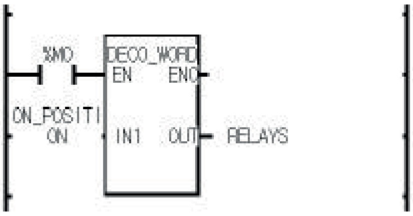
If EN is 1, it turns on ‘ the designated position bit of output bit-string data’ according to the value of IN, and produces an output. Output data types are BYTE, WORD, DWORD and LWORD. LWORD is available only for GM1/2.

FUNCTION	OUT type	Description
DECO_BYTE	BYTE	You can select one of these functions according to output data type.
DECO_WORD	WORD	
DECO_DWORD	DWORD	
DECO_LWORD	LWORD	

 Error

If input data is a negative number or bit position data is out of output-type range, (in case of DECO_WORD, it’ s more than 16), then OUT is 0 and _ERR/_LER flags are set.

 Program example

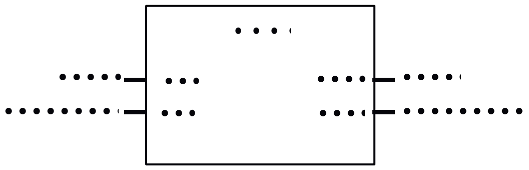
LD	IL
 %M0 DECO_WORD EN ENO ON_POSITI ON IN1 OUT RELAYS	LD %M0 JMPN AAA LD ON_POSITION DECO_WORD ST RELAYS AAA:

(1) If the transition condition (%M0) is on, DECO_WORD function is executed.
(2) If ON_POSITON (INT) = 5, then RELAYS (WORD) = 2#0000 0000 0010 0000.

DEG_***

•••••••••••••••••••••••••••
•

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	•	•	•	•

Function	Description
	**Input** EN: executes the function in case of 1. IN: radian input **Output** ENO: without an error, it will be 1 OUT: degree output

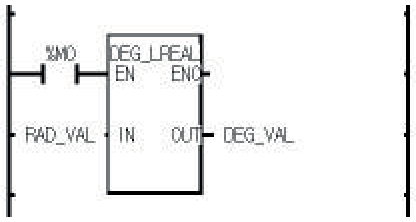
■ **Function**

It converts radian input into degree output.

Function	Input type	Output type	Description
DEG_REAL	REAL	REAL	It converts input (radian) into output (degree).
DEG_LREAL	LREAL	LREAL	

■ **Program example**

LD

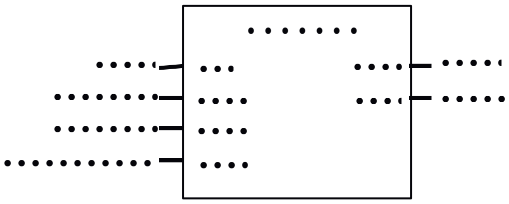


(1) If the transition condition (%M0) is on, DEG_LREAL function is executed.
(2) If input variable RAD_VAL = 1.0, then output variable DEG_VAL = 5.7295779513078550e+001.

DIS_***

••••••••••••••••••
•

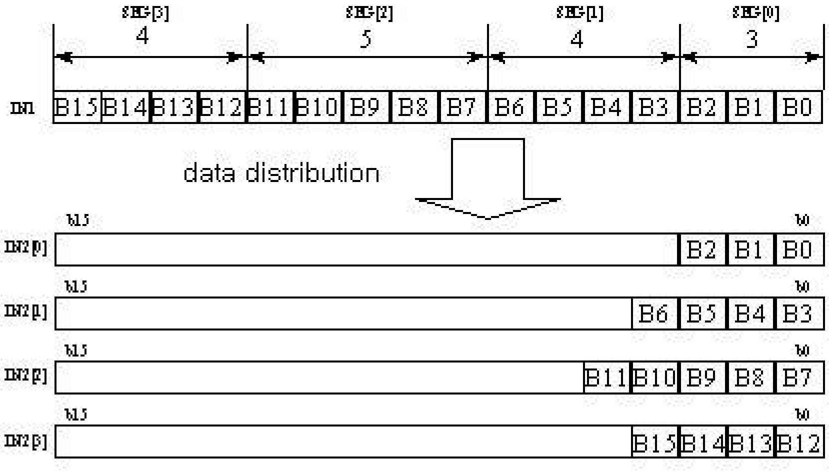
•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	• •	• •	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN1: input data SEG: designated bit array for data distribution **Output** ENO: without an error, it will be 1 OUT: Dummy Output **In/Out** IN2: distributed WORD-array Output

■ Function

It distributes input data over IN2 after segmenting input data by bit number set by SEG.

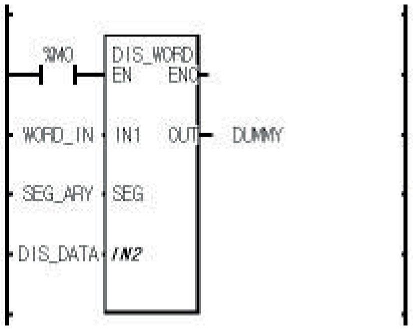
Function	Input type	Description
DIS_BYTE	BYTE	It segments IN1 input by bit number set by SEG and produces IN2 array.
DIS_WORD	WORD	
DIS_DWORD	DWORD	
DIS_LWORD	LWORD	



■ Error

If the sum of designated number of SEG exceeds input variable bit number, _ERR/_LER flags are set.

■ **Program example**

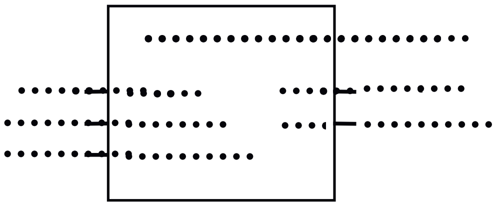
LD
 %M0 DIS_WORD EN ENO WORD_IN IN1 OUT DUMMY SEG_ARY SEG DIS_DATA IN2

(1) If the transition condition (%M0) is on, DIS_WORD function is executed.

(2) If input variable WORD_IN = 16#3456, SEG_ARY = {3, 4, 5, 4}, then, output variable DIS_DATA is:
DIS_DATA[0] = 16#0006
DIS_DATA[1] = 16#000A
DIS_DATA[2] = 16#0008
DIS_DATA[3] = 16#0003

DWORD_LWORD

[illegible]

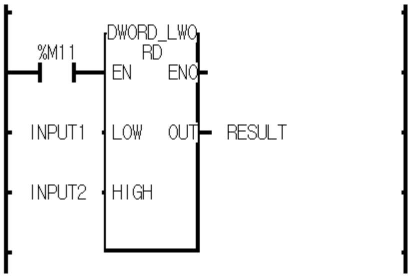
Function	Description
	**Input** EN: executes the function in case of 1. LOW: lower DWORD Input HIGH: upper DWORD Input **Output** ENO: without an error, it will be 1. OUT: LWORD Output

■ Function

It combines 2 DWORD data into one LWORD data.
LOW: lower DWORD Input, HIGH: upper DWORD Input

■ Program example

LD

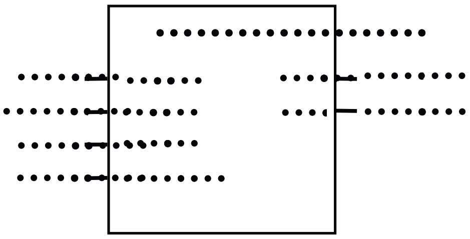


(1) If the transition condition (%M11) is on, DWORD_LWORD function is executed.
(2) If input variable INPUT1 = 16#1A2A3A4A5A6A7A8A and INPUT2 = 16#8C7C6C5C4C3C2C1C, then, output variable RESULT = 16#8C7C6C5C4C3C2C1C1A2A3A4A5A6A7A8A.

DWORD_WORD

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1. IN: DWORD Input **Output** ENO: without an error, it will be 1. OUT: Dummy Output **In/Out** LOW: lower WORD Output HIGH: upper WORD Output

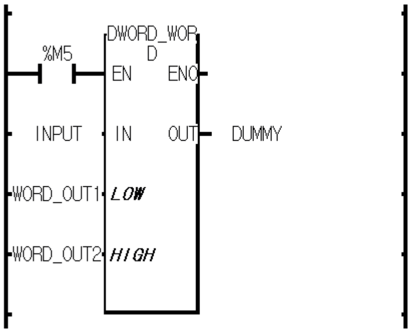
■ Function

It divides one DWORD into two WORD data.

LOW: lower WORD Output, HIGH: upper WORD Output

■ Program example

LD



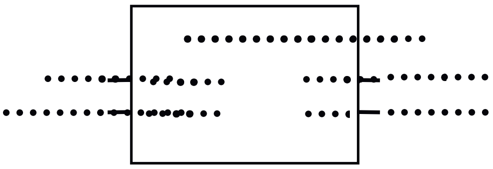
(1) If the transition condition (%M5) is on, DWORD_WORD function is executed.

(2) If input variable INPUT = 16#11223344AABBCCDD, then,

WORD_OUT1 = 16#AABBCCDD and WORD_OUT2 = 16#11223344.

ENCO_***

[illegible]

Function	Description
	**Input** EN: executes the function in case of 1. IN: input data to be encoded **Output** ENO: without an error, it will be 1 OUT: result data after encoding

■ Function

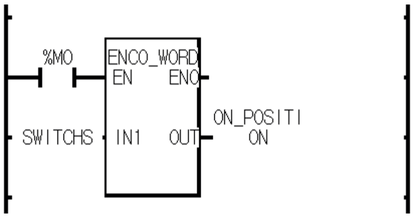
If EN is 1, the output is the highest on-bit position among IN bit string. Input data types are BYTE, WORD, DWORD and LWORD. LWORD is available only for GM1/2.

FUNCTION	IN type	Description
ENCO_BYTE	BYTE	You can select one of these functions according to the input data type.
ENCO_WORD	WORD	
ENCO_DWORD	DWORD	
ENCO_LWORD	LWORD	

■ Error

_ERR and _LER flags are set and OUT is –1 if no bit is 1.

■ Program example

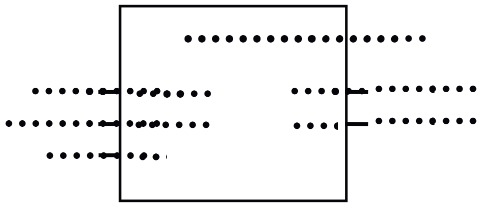
LD	IL
 %M0 ENCO_WORD EN ENO SWITCHS IN1 OUT ON_POSITI ON	LD %M0 JMPN AAA LD SWITCHS ENCO_W ST ON_POSITION AAA:

(1) If the transition condition (%M0) is on, ENCO_WORD function is executed.
(2) If SWITCHS (WORD) = 2#0000 1000 0000 0010, then, the highest on-bit position is 11. Therefore, output ON_POSITON (INT) is '11'.

GET_CHAR

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1. IN: STRING input N: position in a character STRING **Output** ENO: without an error, it will be 1. OUT: Byte Output

■ Function

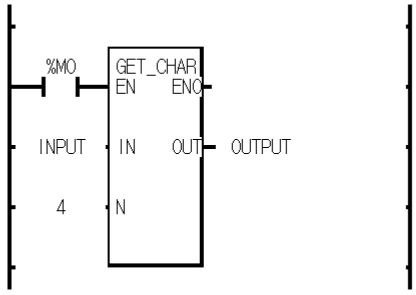
It extracts one byte from a character STRING starting from N.

■ Error

- • _ERR/_LER flags are set if N exceeds the number of byte in STRING.
- • If an error occurs, the output is 16#00.

■ Program example

LD

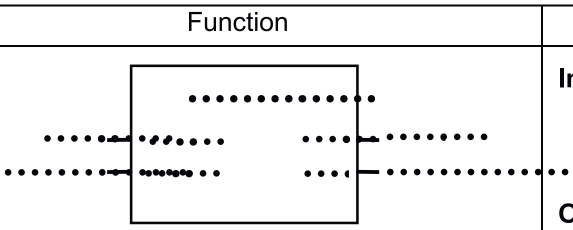


(1) If the transition condition (%M0) is on, GET_CHAT function is executed.
(2) When input INPUT (STRING) = “LG GLOFA PLC”, if you extract 4th character from this string, output variable OUTPUT is 16#47 (“G”).

INC_***

•••
••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	•••••	•• ••	•• •	• •	• •	• •	• •

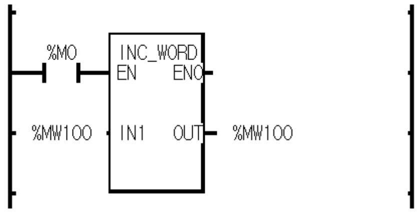
Function	Description
	**Input** EN: executes the function in case of 1. IN: Input data to increase **Output** ENO: without an error, it will be 1 OUT: result data after increase

■ Function

If EN is 1, it increases IN bit string data by 1 and produces an output.
An error does not occur when there’s an overflow; the result is 16#0000 in case of 16#FFFF.
Input data types are BYTE, WORD, DWORD and LWORD. LWORD is available only for GM1/2.

FUNCTION	IN/OUT type	Description
INC_BYTE	BYTE	You can select one of these functions according to the data type.
INC_WORD	WORD	
INC_DWORD	DWORD	
INC_LWORD	LWORD	

■ Program example

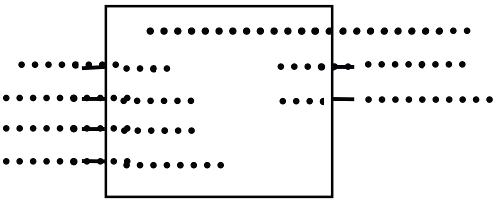
LD	IL
	LD %M0 JMPN BBB LD %MW100 INC_WORD ST %MW100 AAA:

(1) If the transition condition (%M0) is on, INC_WORD function is executed.
(2) If input variable %MW100 = 16#0007 (2#0000 0000 0000 0111), then
output variable %MW100 = 16#0008(2#0000 0000 0000 1000).

LWORD_DWORD

••
••

••••••••••••••	•••••	•••••	•••••	•••••	•••••	•••••	•••••
••••••••••••••	•••	••	•	••	••	••	••

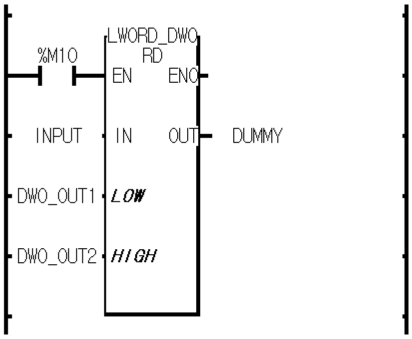
Function	Description
	**Input** EN: executes the function in case of 1. IN: LWORD Input **Output** ENO: without an error, it will be 1. OUT: Dummy Output **In/Out** LOW: lower DWORD Output HIGH: upper DWORD Output

■ Function

- • It divides one LWORD into two DWORD data.
  LOW: lower DWORD Output, HIGH: upper DWORD Output

■ Program example

LD

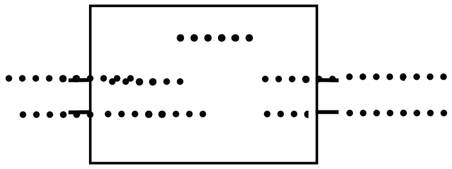


(1) If the transition condition (%M10) is on, LWORD_DWORD function is executed.
(2) If the input variable INPUT = 16#AAAABBBBCCCCDDDDABCDABCDABCDABCD, then,
DWO_OUT1 = 16#ABCDABCDABCDABCD
DWO_OUT2 = 16#AAAABBBBCCCCDDDD.

MCS

•••••••••••••••••••••••••••••   ••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	•••••	•• ••	••	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. NUM: Nesting (0~15) **Output** ENO: If MCS is executed, it will be 1 OUT: Dummy (always 0)

■ Function

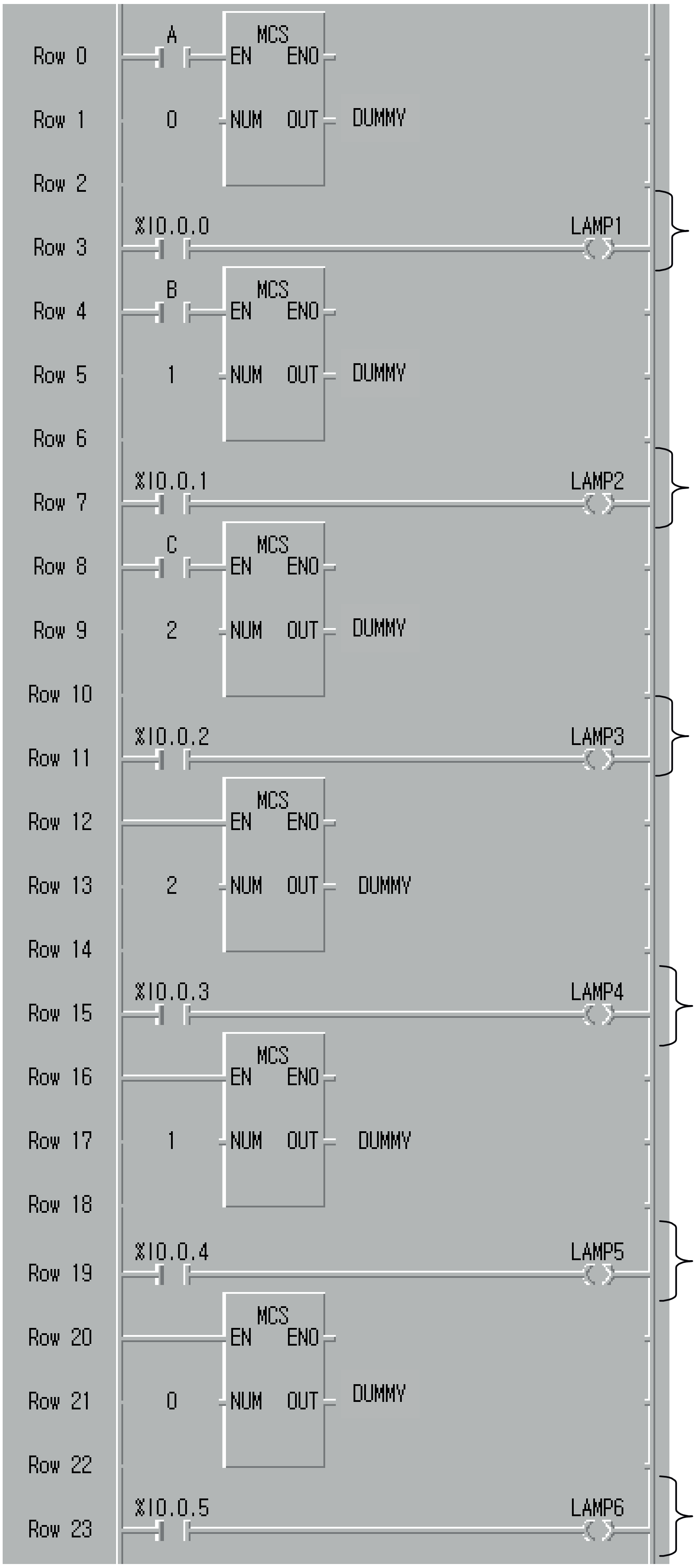
- • If EN is on, MCS function is executed and the program between MCS and MCSCLR function is normally executed.
- • If EN is off, the program between MCS and MCSCLR function is executed as follows:

Instruction	Description
Timer	Current value (CV) becomes 0 and the output (Q) becomes off.
Counter	Output (Q) becomes off and CV retains its present state.
Coil	All becomes off.
Negated coil	All becomes off.
Set coil, reset coil	All retains its current value.
Function, function block	All retains its current value.

- • Even when EN is off, scan time is not shortened because the instructions between MCS and MCSCLR function are executed as the above.
- • Nesting is available in MCS. That is to say, Master Control is divided by Nesting (NUM). You can set up Nesting (NUM) from 0 to 15 and if you set it more than 16, MCS is not executed normally.

Note: if you use MSC without ‘MCSCLR’, MCS function is executed till the end of the program.

■ **Program example**



[illegible]

[illegible]

[illegible]

[illegible]

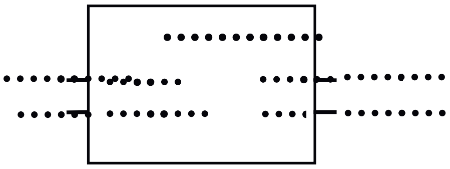
[illegible]

[illegible]

MCSCLR

•••
••

•••••••	••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	•••••	•• ••	••	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. NUM: Nesting (0~15) **Output** ENO: if MCSCLR is executed, it will be 1 OUT: if MCSCLR is executed, it will be 1

■ Function

- • It clears Master Control instruction. And it indicates the end of Master Control.
- • If MCSCLR function is executed, it clears all the MCS instructions which are less than or equal to Nesting (NUM).

* There's no contact before MCSCLR function.

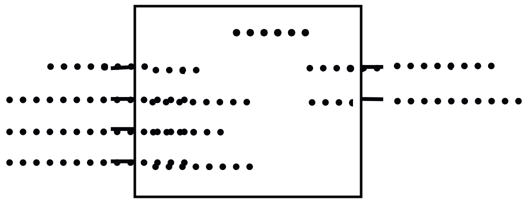
■ Program example

Refer to the MCS function example.

MEQ_***

••••••••••••••••••••••••••
••

•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	••••	••••	••	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN1: Input1 IN2: Input2 MASK: input data to mask **Output** ENO: without an error, it will be 1. OUT: when equal, it will be 1

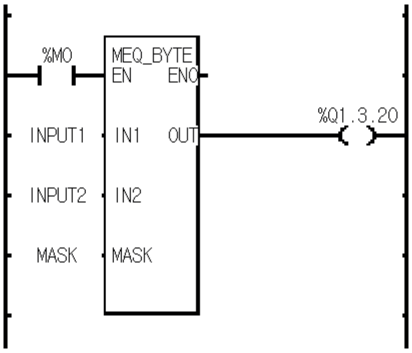
■ Function

- •It compares whether two input variables are equal after masking. If it masks an 8-bit variable with 2#11111100, then, lower 2 bits are excluded when it compares input values.
- •It's available to see whether or not specific bits are on in a variable. For example, in case of comparing 8-bit variables, IN1 is an input variable, IN2 is 16#FF, and MASK for masking is a bit array 2#00101100. If IN1 and IN2 after masking are equal, then output OUT is 1.

Function	Input type	Description
MEQ_BYTE	BYTE	It compares whether two variables are equal after making.
MEQ_WORD	WORD	
MEQ_DWORD	DWORD	
MEQ_LWORD	LWORD	

■ **Program example**

LD



(1) If the transition condition (%M0) is on, MEQ_BYTE function is executed.

(2) Input variable INPUT1 (BYTE) = 2#01011100

INPUT2 (BYTE) = 2#01110101

MASK (BYTE) = 2#11010110

Then, the comparing bits of input variables after masking are as follows:

INPUT1 (BYTE) = 2#01010100

INPUT2 (BYTE) = 2#01010100

INPUT1 and INPUT2 are equal, therefore, output contact %Q1.3.20 is on.

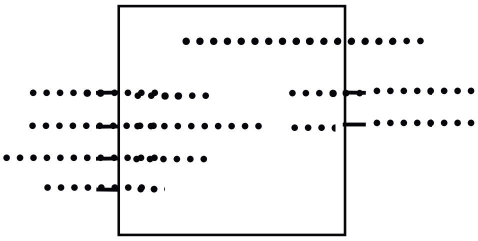
PUT_CHAR

Puts a character in a string••

••

••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	••••	••••	••••	• •	• •	• •	• •

••

Function	Description
	**Input** EN: executes the function in case of 1. DATA: Byte input to insert a string IN: string input N: setting position in a string **Output** ENO: without an error, it will be 1. OUT: string output

■ Function

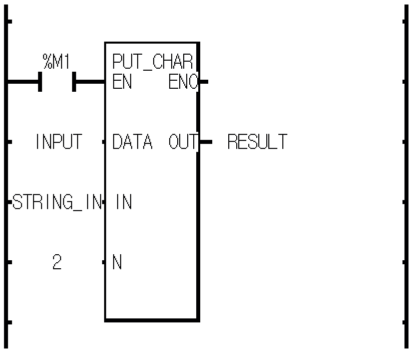
It overwrites one byte input on a specific position (N) string.

■ Error

- •If N value exceeds a byte number of a string, _ERR/_LER flags are set.
- •If an error occurs, the output is 16#00.

■ Program example

LD



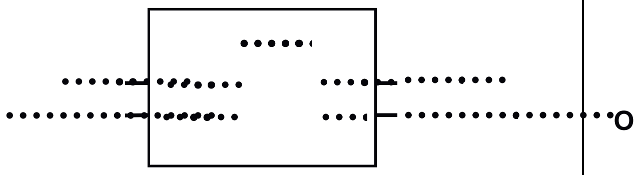
(1) If the transition condition (%M1) is on, PUT_CHAR function is executed.

(2) If input variable INPUT = 16#41 (“A”) and STRING_IN = “TOKEN”, and N = 2, then, output RESULT is “TAKEN”.

RAD_***

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function	Description
	**Input** EN: executes the function in case of 1. IN: degree Input **Output** ENO: without an error, it will be 1. OUT: radian output

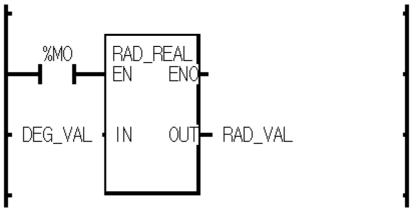
■ Function

- • It converts a degree value (°) into a radian value.
- • If the degree is over 360°, its converts normally.
  For example, if input is 370°, output is 370° - 360° = 10°.

Function	Input type	Output type	Description
RAD_REAL	REAL	REAL	It converts a degree value (°) into a radian value.
RAD_LREAL	LREAL	LREAL	

■ Program example

LD

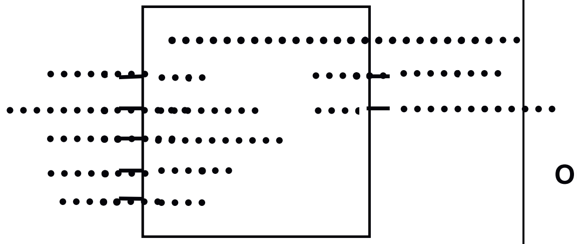


(1) If the transition condition (%M0) is on, RAD_REAL function is executed.
(2) If input variable DEG_VAL = 127(°), its output RAD_VAL = 2.21656823.

ROTATE_A_***

•••
••

•••••••	••••	•••••	•••••	•••••	•••••	•••••	•••••••
••••••••••••••••••••	••••	••••	••	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. N: element number to rotate STRT: starting position to rotate in an array block END: ending position to rotate in an array block **Output** ENO: without an error, it will be 1 OUT: overflowing data **In/Out** SRC: array block to rotate

■ **Function**

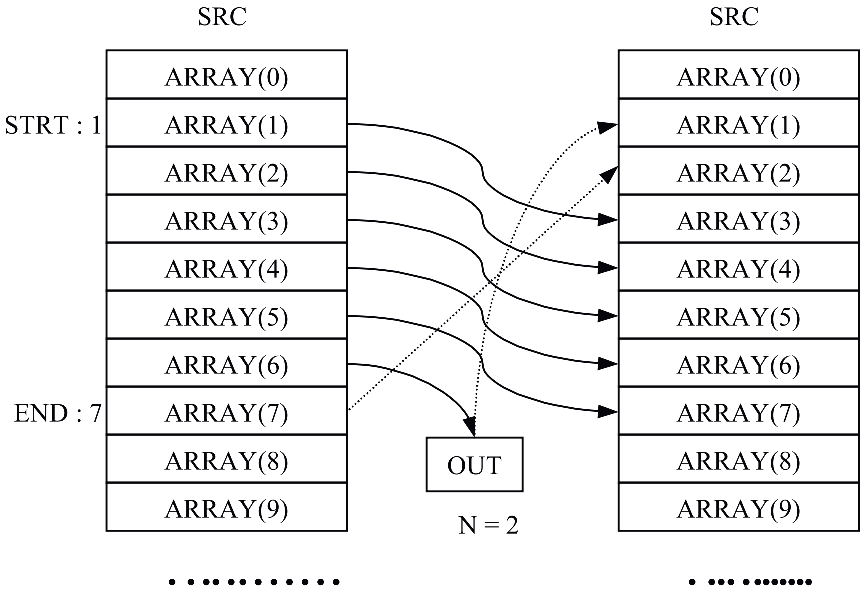
- • It rotates designated elements of an array block in the chosen direction.
- • Setting:

- Scope: STRT and END set a data array to rotate

- Rotation direction and time: rotates N times in the chosen direction set by STRT and END (STRT → END)

- Input data setting: fills an empty element after rotation with Input data (IN)

- Output: the result is written at ANY_ARY designated by ***SRC***, and the data to rotate from END to STRT is written at OUT.



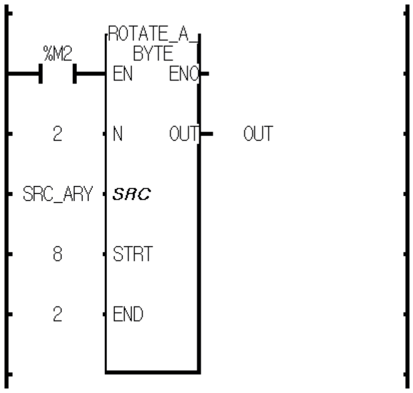
Function	In/Out array type	Description
ROTATE_A_BOOL	BOOL	It rotates designated elements of an array block in the chosen direction.
ROTATE_A_BYTE	BYTE	
ROTATE_A_WORD	WORD	
ROTATE_A_DWORD	DWORD	
ROTATE_A_LWORD	LWORD	
ROTATE_A_SINT	SINT	
ROTATE_A_INT	INT	
ROTATE_A_DINT	DINT	
ROTATE_A_LINT	LINT	
ROTATE_A_USINT	USINT	
ROTATE_A_UINT	UINT	
ROTATE_A_UDINT	UDINT	
ROTATE_A_ULINT	ULINT	
ROTATE_A_REAL	REAL	
ROTATE_A_LREAL	LREAL	
ROTATE_A_TIME	TIME	
ROTATE_A_DATE	DATE	
ROTATE_A_TOD	TOD	
ROTATE_A_DT	DT	

■ Error

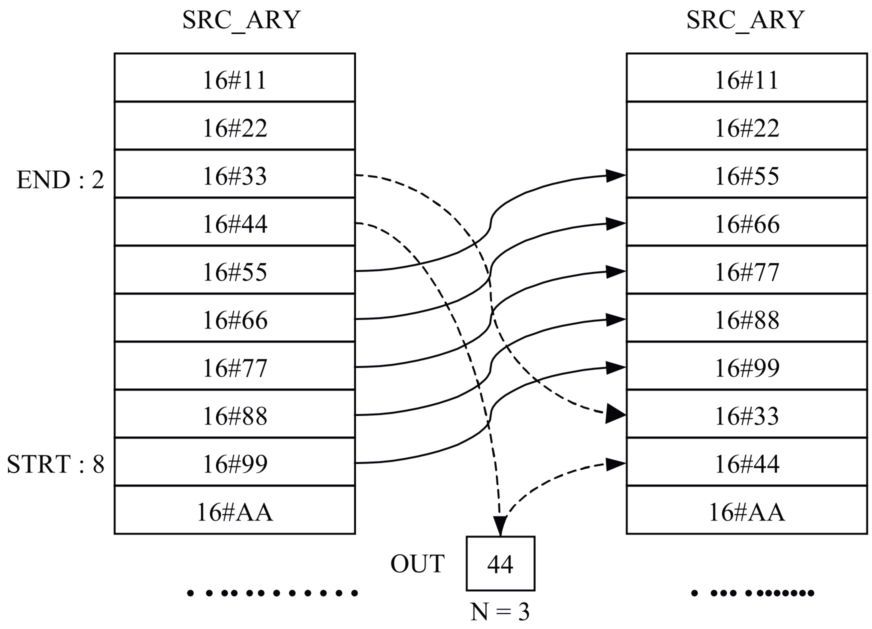
- • If STRT or END exceed the range of SRC array element, _ERR/_LER flags are set.
- • If an error occurs, there’s no change in SRC and output OUT is the initial value of each variable type (i.e. INT=0, TIME=T#0S).

■ Program example

LD



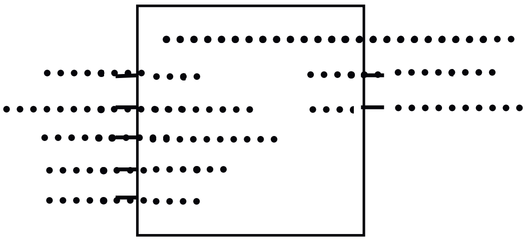
(1) If input condition (%M2) is on, ROTATE_A_BYTE function is executed.
(2) It rotates designated elements (from 2nd to 8th elements) of SRC_ARY in the chosen direction set by STRT and END (from index 8 to index 2): refer to the diagram on the opposite page.
(3) The overflowing data (16#44) is written at OUT.


SRC_ARY
16#11
16#22
END : 2
16#33
16#44
16#55
16#66
16#77
16#88
STRT : 8
16#99
16#AA
SRC_ARY
16#11
16#22
16#55
16#66
16#77
16#88
16#99
16#33
16#44
16#AA
OUT
44
N = 3

ROTATE_C_***

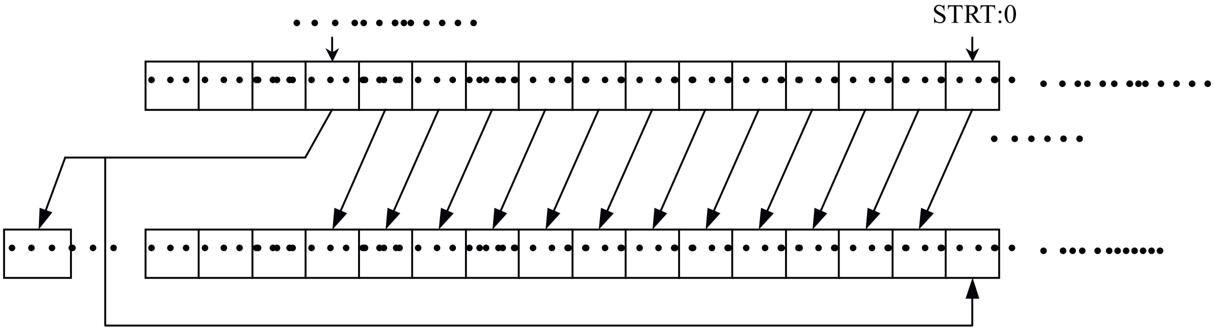
Rotates a designated bit array of SRC bit arrays• •
••

•••••••	••••	•••	•••	••••	•••	•••	••••
•••••••••••	•••••	•• ••	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. STRT: starting bit position of SRC bit array to rotate END: ending bit position of SRC bit array to rotate N: bit number to shift **Output** ENO: without an error, it will be 1 OUT: carry output **In/Out** ***SRC***: variable for rotation

■ Function

- • It rotates a designated bit array of SRC bit arrays in the chosen direction.
- • Setting:
  - Scope: STRT and END set a bit data to rotate.
  - Rotation direction and time: rotates N times in the chosen direction set by STRT and END (STRT → END)
  - Output: the result is written at ANY_ARY designated by SRC, and the data to rotate from END to STRT is written at OUT.



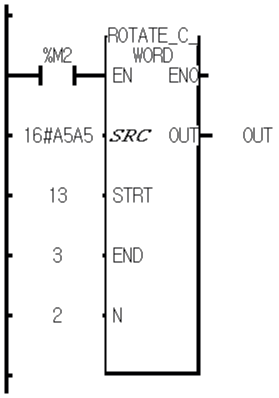
Function	SRC type	Description
ROTATE_C_BYTE	BYTE	It rotates a designated bit array of SRC bit arrays N times in the chosen direction.
ROTATE_C_WORD	WORD	
ROTATE_C_DWORD	DWORD	
ROTATE_C_LWORD	LWORD	

■ Error

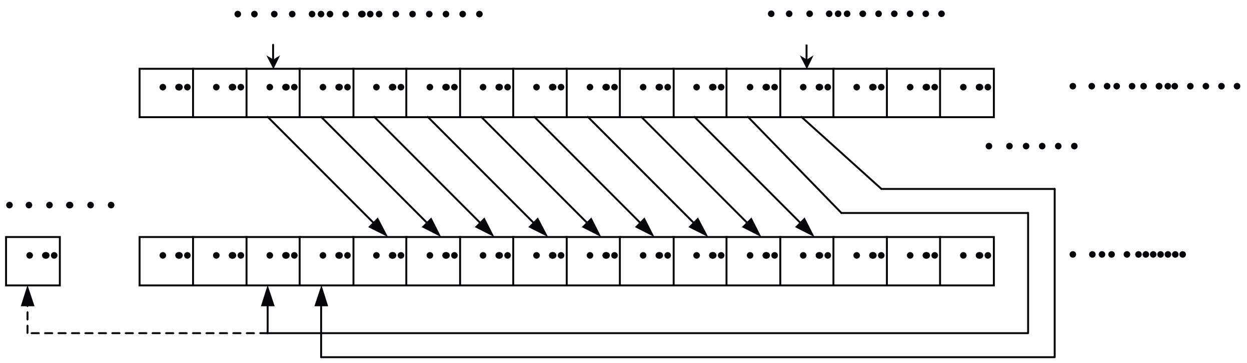
- • If STRT or END exceed the bit number of SRC variable type, _ERR and _LER flags are set.
- • There’s no change in SRC data.

■ **Program example**

LD



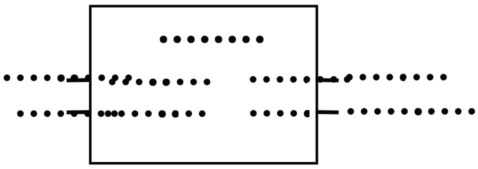
(1) If the transition condition (%M2) is on, ROTATE_C_WORD function is executed.
(2) It rotates the designated bit array, from STRT (13) to END (3), of SRC (16#A5A5) 2 times in the chosen direction set by STRT and END (from STRT to END): refer to the diagram as below.
(3) The result data after rotation is written at SRC (16#896D), and the overflowing bit (0) is written at OUT.



RTC_SET

Writes Time data••
••

•••••••	••••	•••••	•••••	••••	•••	••••	•••••••
•••••••••••	•••••	•• ••	• •	• •	• •	• •	• •

Function	Description
	**Input** REQ: executes the function with rising pulse input DATA: TIME data to input **Output** DONE: without an error, it will be 1 STAT: If an error occurs, an error code is written

■ Function

• •It writes RTC data to Clock Device with a rising pulse input.

Variable	Content	Example	Variable	Content	Example
DATA[0]	Last 2-digit number of years	16#01	DATA[4]	Minutes	16#30
DATA[1]	Months	16#03	DATA[5]	Seconds	16#45
DATA[2]	Dates	16#15	DATA[6]	Days	16#03
DATA[3]	Hours	16#18	DATA[7]	First 2-digit number of years	16#20

* The above example is "2001-03-15 18:30:45, Thursday".
* Days are indicated as follows: Mon (0), Tue (1), Wed (2), Thu (3), Fri (4), Sat (5), Sun (6).
• • The above DATA variables are declared as array Byte variables and set as BCD data.

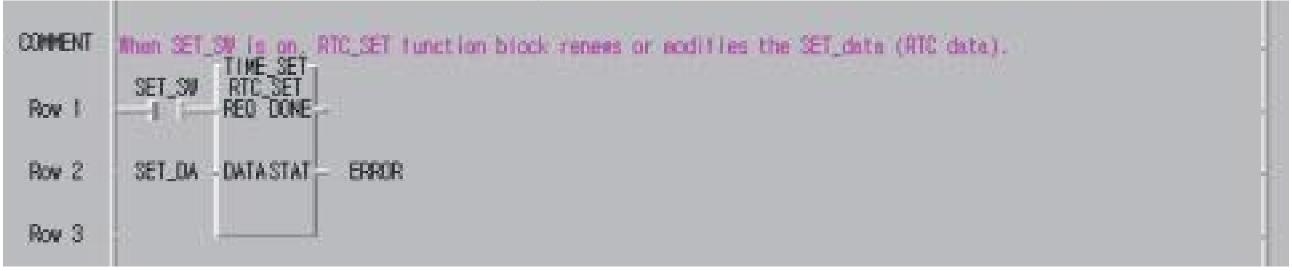
■ Error

If CPU does not support RTC function or RTC data is out of range, the output is 0 and the error code is written at STAT.

Error code	Description
00	No error
01	No RTC module installed. * GM6: GM6-CPUB and GM6-CPUC support RTC. * GM7: G7E-RTCA should be installed.
02	Wrong RTC data. Example: 14 (Months) 32 (Dates) 25 (Hours) * Modify RTC data.

■ Program example

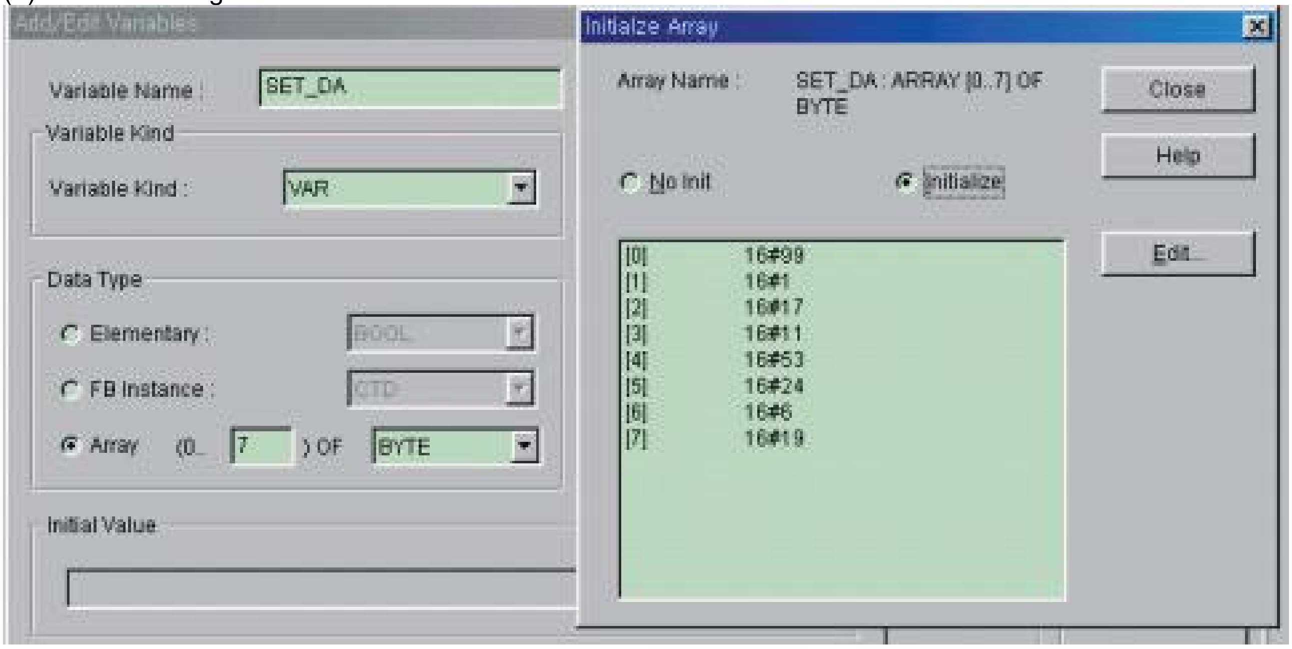
Its RTC data is 1999-01-17 11:53:24, Sunday.



```
COMMENT  When SET_SW is on, RTC_SET function block renews or modifies the SET_data (RTC data).
                  TIME_SET
          SET_SW  RTC_SET
Row 1   ---| |----REQ  DONE-
Row 2     SET_DA -DATA STAT-   ERROR
Row 3
```

(1) When SET_SW is on, RTC_SET function block renews or modifies the SET_data (RTC data).

(2) Variable setting is shown as below.



Add/Edit Variables

Variable Name : SET_DA

Variable Kind

Variable Kind : VAR

Data Type

○ Elementary : BOOL

○ FB Instance : CTD

● Array [0.. 7] OF BYTE

Initial Value

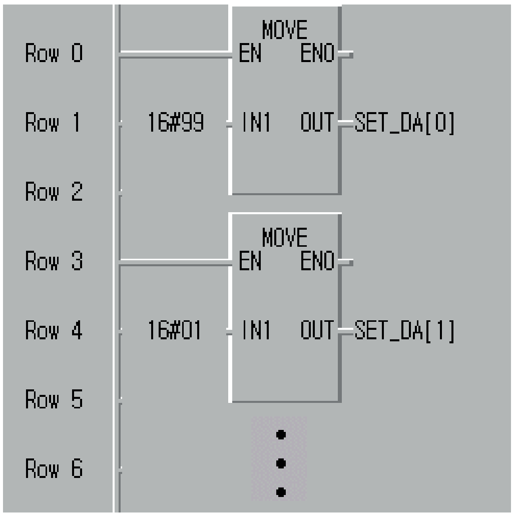
Initialze Array

Array Name : SET_DA : ARRAY [0..7] OF BYTE

○ No Init ● Initialize

[0]	16#99
[1]	16#1
[2]	16#17
[3]	16#11
[4]	16#53
[5]	16#24
[6]	16#6
[7]	16#19

(3) You can set each TIME data using MOVE function.



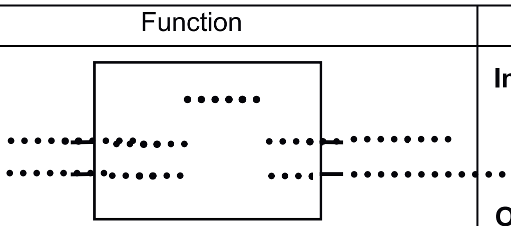
(4) Use the following flags to read RTC data.
e.g. 1998-12-22 19:37:46, Tuesday

Flag	Type	Description	Data
_RTC_TOD	TOD	Current time of RTC	TOD#19:37:46
_RTC_WEEK	UINT	Current day of RTC *(0: Mon, 1: Tue, 2: Wed, 3: Thu, 4: Fri, 5: Sat, 6: Sun)	1
_RTC_DATE	DATE	Current date of RTC (1984-01-01 ~ 2083-12-31)	D#1998-12-22
_RTC_ERR	BOOL	When RTC data is wrong, it is 1.	0
_RTC_TIME[n] * n: 0 ~ 7	BCD	BCD data of current time of RTC _RTC _TIME[0]: Last 2-digit number of years _RTC _TIME[1]: Months _RTC _TIME[2]: Dates _RTC _TIME[3]: Hours _RTC _TIME[4]: Minutes _RTC _TIME[5]: Seconds _RTC _TIME[6]: Days _RTC _TIME[7]: First 2-digit number of years Days (0: Mon, 1: Tue, 2: Wed, 3: Thu, 4: Fri, 5: Sat, 6: Sun)	_RTC _TIME[0]: 16#98 _RTC _TIME[1]: 16#12 _RTC _TIME[2]: 16#22 _RTC _TIME[3]: 16#19 _RTC _TIME[4]: 16#37 _RTC _TIME[5]: 16#46 _RTC _TIME[6]: 16#1 _RTC _TIME[7]: 16#19

SEG

Converts BCD or HEX into 7 segment display code••
••

•••••••	••••	•••••	•••••	•••	•••	•••	••••••
•••••••••••••	•••••	••••	••	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: Input data to covert into 7 segment code **Output** ENO: without an error, it will be 1. OUT: result data converted into 7 segment data

■ **Function**

If EN is 1, it converts BCD or HEX (hexadecimal) of IN into 7 segment display code as below and produces output OUT. If an input is BCD type, it is available to display a number between 0000 and 9999. And in case of HEX input, it's available to display a number between 0000 and FFFF on 4-digit 7 segment display.

Display example

1) 4-digit BCD -> 4-digit 7 segment code: use SEG function

2) 4-digit HEX -> 4-digit 7 segment code: use SEG function

3) INT -> 4-digit BCD-type 7 segment code: use INT_TO_BCD function first and SEG function

4) INT -> 4-digit HEX-type 7 segment code: use INT_TO_WORD function first and SEG function

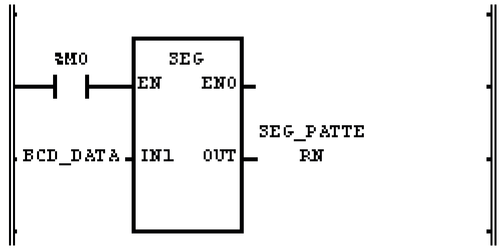
5) When 7 segment display digits are more than 4,

A) in case of BCD, HEX type, use SEG function, after dividing them into 4 digits.

B) INT -> 8-digit BCD-type 7 segment code:

Divide INT by 10,000 and convert ‘quotient’ and ‘remainder’ into upper/lower 4-digit 7 segment code using INT_TO_BCD and SEG function.

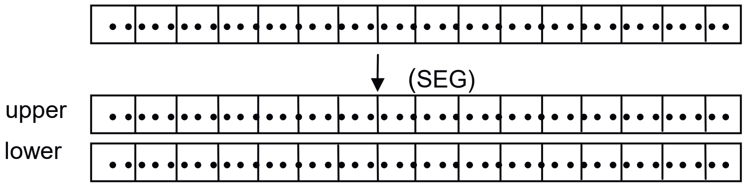
■ **Program example**

LD	IL
	LD %M0 JMPN BBB LD BCD_DATA SEG ST SEG_PATTERN BBB:

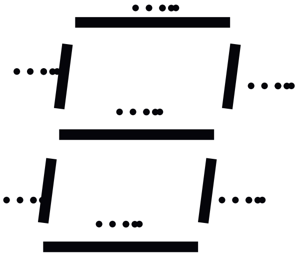
(1) If the transition condition (%M0)• •On• • •SEGfunction is executed.
(2) If input variable BCD_DATA (WORD) = 16#1234,
the output is ‘2#00000110_01011011_01001111_01100110’ which is displayed as a 7 segment code (1234) and written at SEG_PATTERN (DWORD).

Input (IN1): BCD_DATA (WORD) = 16#1234

Output (OUT): SEG_PATTERN (DWORD) = 16#065B4F66



7 segment configuration



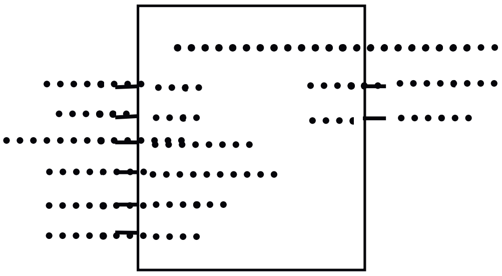
Conversion table for 7 segment code

Input (BCD)	Input (HEX)	INT	Output								Display Data
			B7	B6	B5	B4	B3	B2	B1	B0	
0	0	0	0	0	1	1	1	1	1	1	***0***
1	1	1	0	0	0	0	0	1	1	0	***1***
2	2	2	0	1	0	1	1	0	1	1	***2***
3	3	3	0	1	0	0	1	1	1	1	***3***
4	4	4	0	1	1	0	0	1	1	0	***4***
5	5	5	0	1	1	0	1	1	0	1	***5***
6	6	6	0	1	1	1	1	1	0	1	***6***
7	7	7	0	0	1	0	0	1	1	1	***7***
8	8	8	0	1	1	1	1	1	1	1	***8***
9	9	9	0	1	1	0	1	1	1	1	***9***
	A	10	0	1	1	1	0	1	1	1	***A***
	B	11	0	1	1	1	1	1	0	0	***B***
	C	12	0	0	1	1	1	0	0	1	***C***
	D	13	0	1	0	1	1	1	1	0	***D***
	E	14	0	1	1	1	1	0	0	1	***E***
	F	15	0	1	1	1	0	0	0	1	***F***

SHIFT_A_***

••
••

•••••••	••••	••••	••••	•••	•••	•••	••••
•••••••••••	•••••	•• ••	•	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: Input data to empty element after shifting N: number to shift STRT: starting position to shift in an array block END: ending position to shift in an array block **Output** ENO: without an error, it will be 1 OUT: overflowing data **In/Out** SRC: array block to shift

■ Function

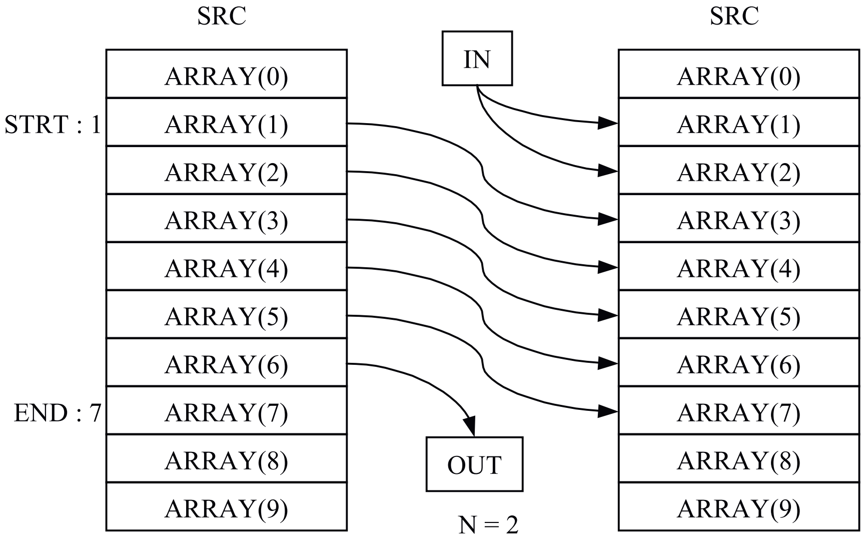
- • It shifts designated elements of an array block in the chosen direction.
- • Setting

  - Scope: STRT and END set a data array to rotate.

  - Shifting direction and time: rotates N times in the chosen direction set by STRT and END (STRT → END)

  - Input data setting: fills an empty element after shifting with input data (IN).

  - Output: the result is written at ANY_ARY designated by ***SRC***, and the overflowing data by shifting from END to STRT is written at OUT.



• •••••••••••   • ••• •••••••

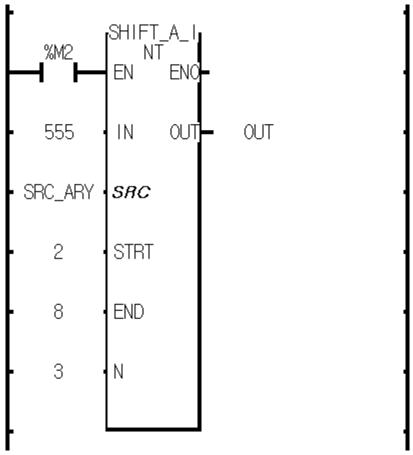
Function	In/Out Array Type	Description
SHIFT_A_BOOL	BOOL	It shifts designated elements of an array block in the chosen direction.
SHIFT_A_BYTE	BYTE	
SHIFT_A_WORD	WORD	
SHIFT_A_DWORD	DWORD	
SHIFT_A_LWORD	LWORD	
SHIFT_A_SINT	SINT	
SHIFT_A_INT	INT	
SHIFT_A_DINT	DINT	
SHIFT_A_LINT	LINT	
SHIFT_A_USINT	USINT	
SHIFT_A_UINT	UINT	
SHIFT_A_UDINT	UDINT	
SHIFT_A_ULINT	ULINT	
SHIFT_A_REAL	REAL	
SHIFT_A_LREAL	LREAL	
SHIFT_A_TIME	TIME	
SHIFT_A_DATE	DATE	
SHIFT_A_TOD	TOD	
SHIFT_A_DT	DT	

■ Error

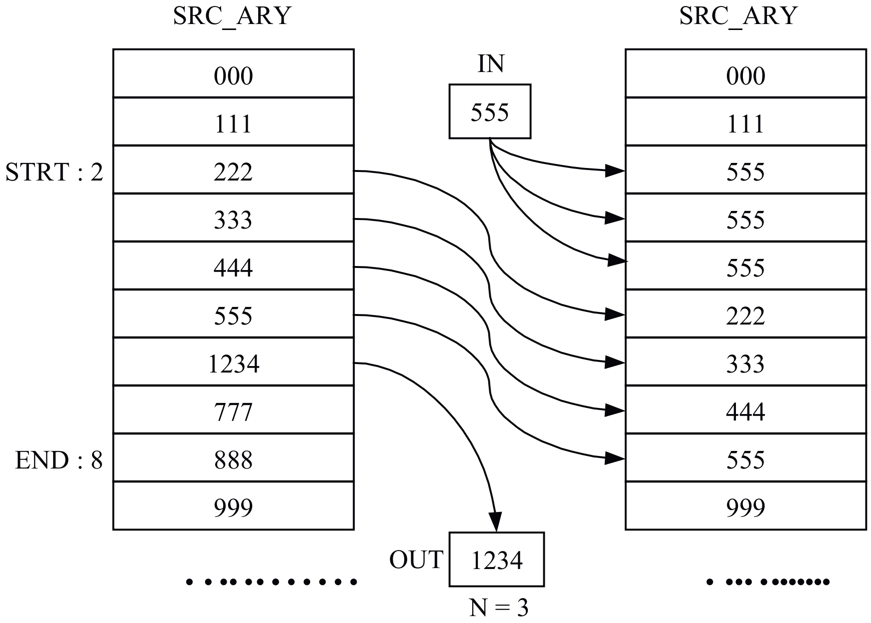
- • If STRT or END exceed the range of SRC array element, _ERR and _LER flags are set.
- • If an error occurs, there's no change in SRC and output OUT is the initial value of each variable type (i.e. INT=0, TIME=T#0S).

■ Program example

LD



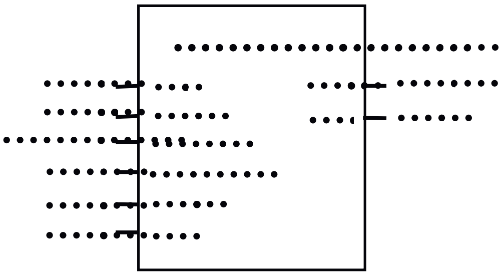
(1) If the input condition (%M2) is on, SHIFT_A_INT function is executed.
(2) It shifts designated elements (from 2nd to 8th elements) of SRC_ARY.
(3) It shifts three times the designated elements.
(4) The empty elements after shifting, from array index 2 to array index 3, are filled with input '555'.
(5) The overflowing data (1234), carry output, is written at OUT.


SRC_ARY
SRC_ARY
IN
555
STRT : 2
END : 8
000
111
222
333
444
555
1234
777
888
999
000
111
555
555
555
222
333
444
555
999
OUT
1234
N = 3

SHIFT_C_***

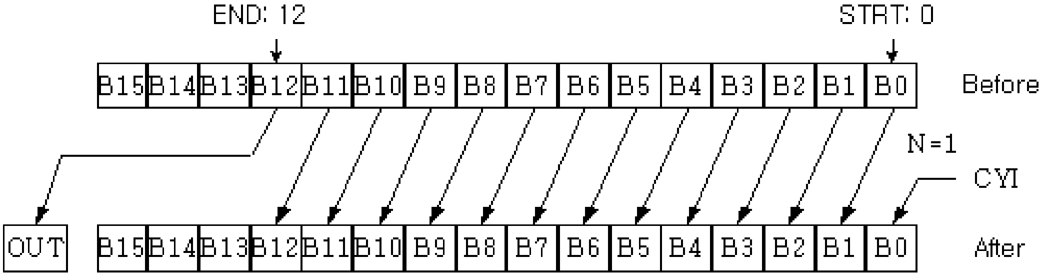
••••••••••••••••••••••••••••••••••••
••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	••••	••••	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. ***CYI: Carry Input*** STRT: starting bit position of SRC bit array to shift END: ending bit position of SRC bit array to shift N: bit number to shift **Output** ENO: without an error, it will be 1 OUT: carry output **In/Out** ***SRC***: variable for shifting

■ **Function**

- • It shifts a designated bit array of SRC bit arrays N times in the chosen direction.
- • Setting:
  - Scope: STRT and END set a bit data to shift.
  - Shifting direction and time: shifts N times from STRT to END.
  - Input data setting: fills empty bit after shifting with input data (CYI).
  - Output: the result is written at ANY_BIT designated by ***SRC***, and the overflowing bit data by shifting from END to STRT is written at OUT.


END: 12
STRT: 0
B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
Before
N=1
CYI
OUT
B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
After

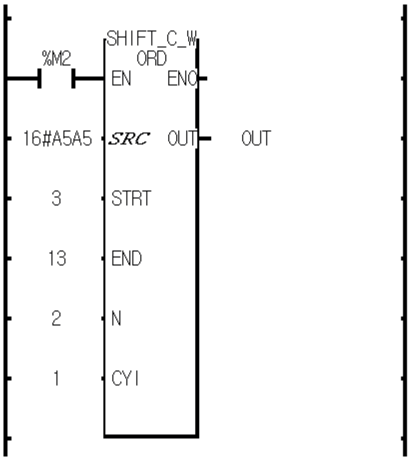
Function	SRC type	Description
SHIFT_C_BYTE	BYTE	It shifts a designated bit array of SRC bit arrays N times.
SHIFT_C_WORD	WORD	
SHIFT_C_DWORD	DWORD	
SHIFT_C_LWORD	LWORD	

■ Error

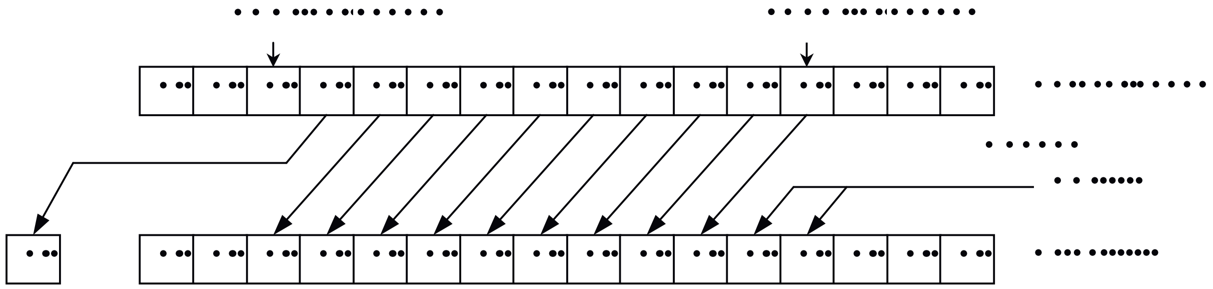
- • If STRT or END exceed the bit number of SRC variable type, _ERR and _LER flags are set.
- • There's no change in SRC data.

■ Program example

LD



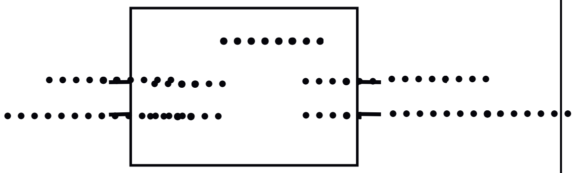
(1) If the transition condition (%M2) is on, SHIFT_C_WORD function is executed.
(2) 16#A5A5 is shifted from STRT to END by 2 bits and the empty bits after shifting are filled with 1 (CYI).
(3) SRC after shifting is 16#969D and the overflowing bit data (0) is written at OUT after 2-bit shifting.



SWAP_***

Swaps upper data for lower data••
••

•••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	•••••	•• ••	••	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: Input **Output** ENO: without an error, it will be 1. OUT: swapped data

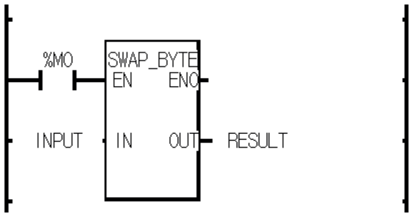
■ Function

It swaps upper data for lower data.

Function	Input type	Description
SWAP_BYTE	BYTE	Swaps upper nibble for lower nibble data.
SWAP_WORD	WORD	Swaps upper byte for lower byte data.
SWAP_DWORD	DWORD	Swaps upper word for lower word data.
SWAP_LWORD	LWORD	Swaps upper double word for lower double word data.

■ Program example

LD

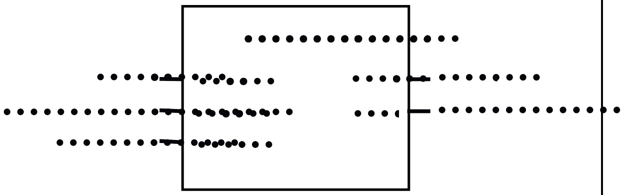


(1) If the transition condition (%M0) is on, SWAP_BYTE function is executed.
(2) If INPUT (BYTE) = 16#5F, RESULT (BYTE) = 16#F5.

UNI_***

•••••••••••••••••••••••
••

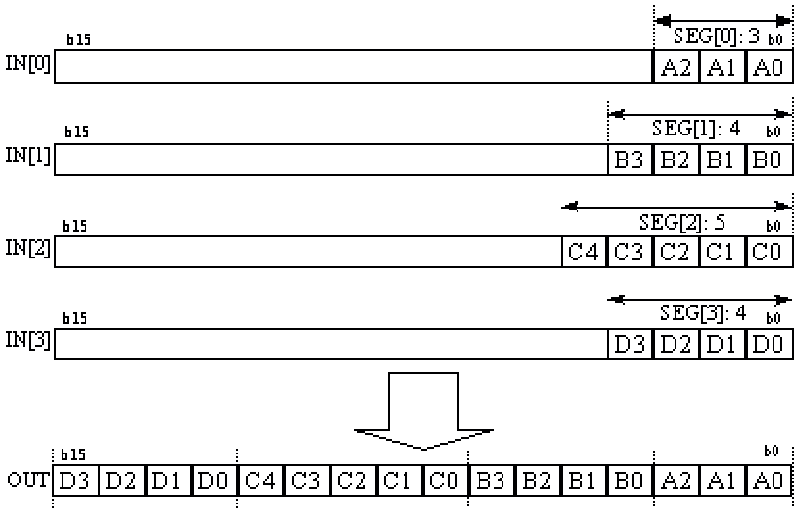
•••••••	••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	•••••	••••	••••	• •	• •	• •	• •	• •

Function	Description
	**Input** EN: executes the function in case of 1. IN: input data array SEG: bit-number-designate array to unite data **Output** ENO: without an error, it will be 1 OUT: united data

■ Function

It unites an input data array from the lower bit to a designated bit set by SEG and produces an output.

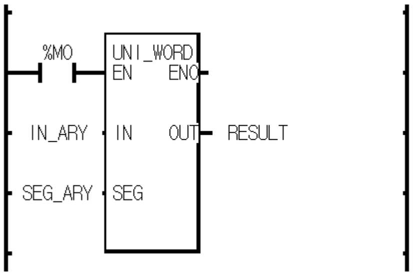
Function	Input type	Output type	Description
UNI_BYTE	BYTE	BYTE	It cuts an input array into bit data set by SET and produces an output (united data) with the same array type of input.
UNI_WORD	WORD	WORD	
UNI_DWORD	DWORD	DWORD	
UNI_LWORD	LWORD	LWORD	



- • If the sum of value set by SEG exceeds the bit number of input data type, _ERR and _LER flags are set.
- • If the number of arrays of IN and SEG is different, output OUT is 0 and _ERR and _LER flags are set.

■ **Program example**

LD

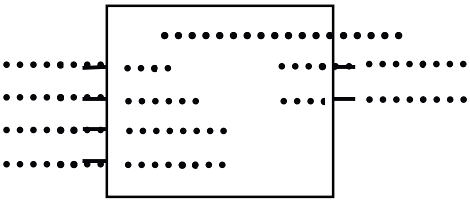


(1) If the transition condition (%M0) is on, UNI_WORD function is executed.
(2) If input IN_ARY and SEG_ARY are as below,

A 3 B 5	3
B 4 C 6	4
C 5 D 7	7
D 6 E 8	2

output RESULT = 2#00 1010111 0110 101 = 16#2BB5.

WORD_BYTE

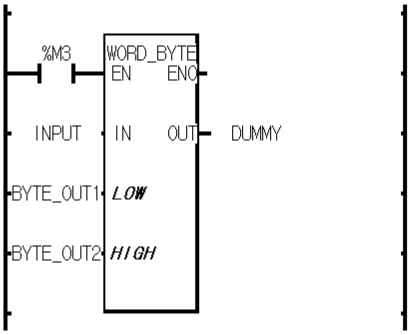
Function	Description
	**Input** EN: executes the function in case of 1. IN: WORD Input **Output** ENO: without an error, it will be 1. OUT: dummy output **In/Output** LOW: lower BYTE output HIGH: upper BYTE output

■ Function

- • It divides one word data into two byte data.
  LOW: lower byte output, HIGH: upper byte output

■ Program example

LD

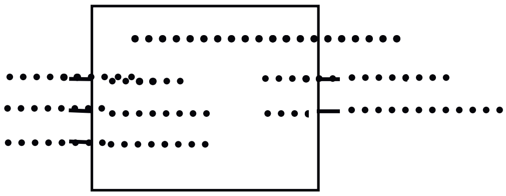


(1) If the transition condition (%M3) is on, WORD_BYTE function is executed.
(2) If input variable INPUT is 16#ABCD, then BYTE_OUT1 = 16#CD and BYTE_OUT2 = 16#AB.

WORD_DWORD

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

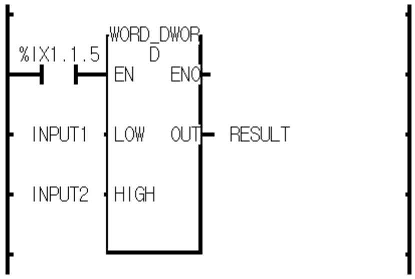
Function	Description
	**Input** EN: executes the function in case of 1. LOW: lower WORD input HIGH: upper WORD input **Output** ENO: without an error, it will be 1. OUT: DWORD output

■ Function

It combines two WORD data into one DWORD.
LOW: lower WORD input, HIGH: upper WORD input

■ Program example

LD



(1) If the transition condition (%IX1.1.5) is on, WORD_DWORD function is executed.
(2) If input variable INPUT1 = 16#10203040 and INPUT2 = 16#A0B0C0D0,
output variable RESULT = 16#A0B0C0D010203040.

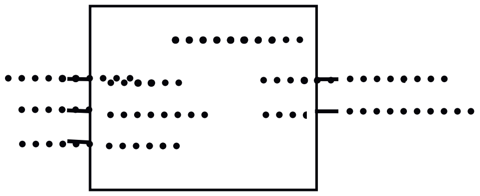
XCHG_ ***

Exchanges two input data• •

••

•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••	•••••	••••	• •	• •	• •	• •	• •

••

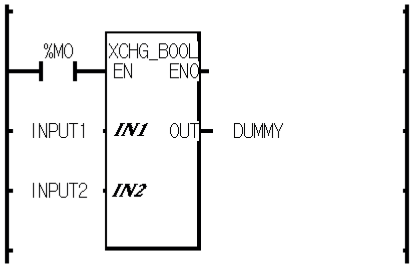
Function	Description
	**Input** EN: executes the function in case of 1. **Output** ENO: Without an error, it will be 1. OUT: Dummy Output **In/Out** IN1: In/Output 1 IN2: In/Output 2

■ Function

Exchanges input1 data with input2 data.

Function	In/Out type	Description
XCHG_BOOL	BOOL	Exchanges two BOOL input data.
XCHG_BYTE	BYTE	Exchanges two BYTE input data.
XCHG_WORD	WORD	Exchanges two WORD input data.
XCHG_DWORD	DWORD	Exchanges two DWORD input data.
XCHG_LWORD	LWORD	Exchanges two LWORD input data.
XCHG_SINT	SINT	Exchanges two SINT input data.
XCHG_INT	INT	Exchanges two INT input
XCHG_DINT	DINT	Exchanges two DINT input data.
XCHG_LINT	LINT	Exchanges two LINT input data.
XCHG_USINT	USINT	Exchanges two USINT input data.
XCHG_UINT	UINT	Exchanges two UINT input data.
XCHG_UDINT	UDINT	Exchanges two UDINT input data.
XCHG_ULINT	ULINT	Exchanges two ULINT input data.
XCHG_REAL	REAL	Exchanges two REAL input data.
XCHG_LREAL	LREAL	Exchanges two LREAL input data.
XCHG_TIME	TIME	Exchanges two TIME input data.
XCHG_DATE	DATE	Exchanges two DATE input data.
XCHG_TOD	TOD	Exchanges two TOD input data.
XCHG_DT	DT	Exchanges two DT input data.
XCHG_STRING	STRING	Exchanges two STRING input data.

■ **Program example**

LD


(1) If the transition condition (%M0) is on, XCHG_BOOL function is executed.

(2) If INPUT1 = 0 and INPUT2 = 1, it will exchange two input data. After the function execution, INPUT1 = 1 and INPUT2 = 0.

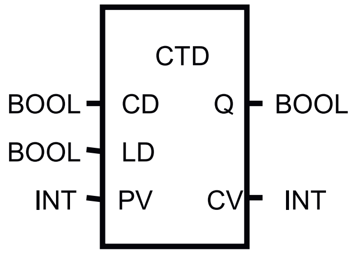
8.3 Basic Function Block Library

1. This chapter describes basic function blocks respectively.
2. It’s much easier to apply function block library to your program after grasping the general of function blocks.

CTD

Down Counter (function block)

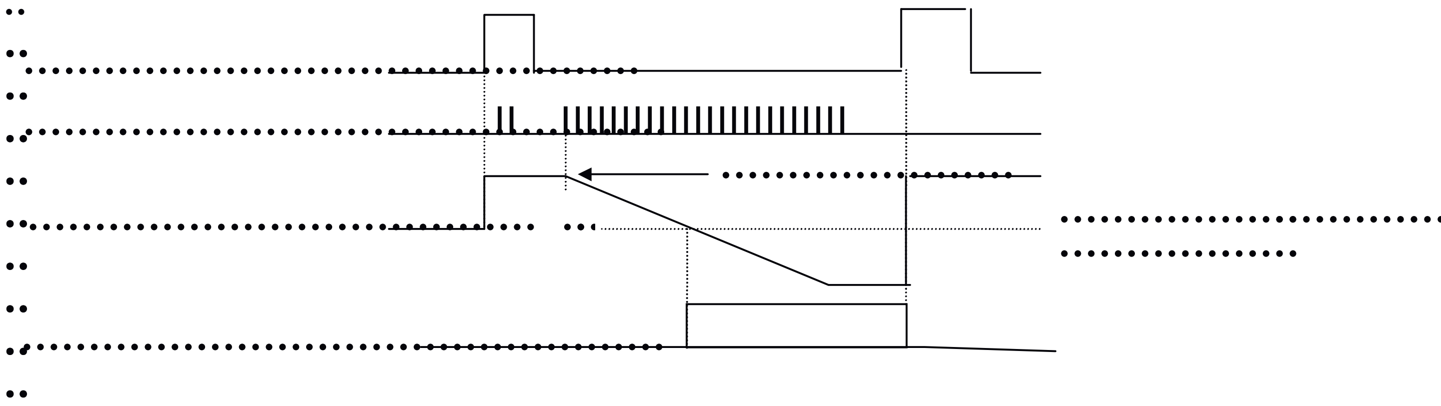
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function block	Description
 CTD BOOL CD Q BOOL BOOL LD INT PV CV INT	**Input** CD: down counter pulse input LD: loads a preset value PV: preset value **Output** Q: down counter output CV: current value

■ Function

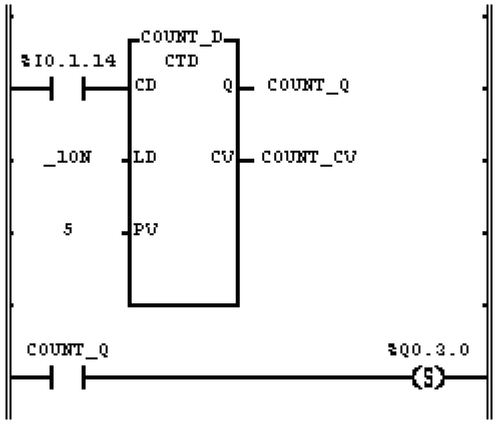
- •Down counter function block CTD decreases the current value (CV) by 1 with every rising pulse input.
- •CV decreases only when CV is more than the minimum value of INT (-32768); after reaching it, CV does not change its value.
- •When LD is 1, PV is loaded into CV (CV=PV).
- •Output Q is 1 when CV is 0 or a negative number.

■ Time Chart



■ **Program Example**

This is the program that sets the output contact (%O0.3.0) when the down counter pulse input enters the input contact (%I0.1.14) five times.

LD	IL
	CAL CTD COUNT_0 CD %I0.1.14 LD _1ON PV 5 LD COUNT_D.Q ST COUNT_Q LD COUNT_D.CV ST COUNT_CV LD COUNT_Q S %Q0.3.0

(1) Register the name of CTD function block (COUNT_D).

(2) Make the input contact (%I0.1.14) attached to CD.

(3) Make the flag _10N (1 scan ON contact) that loads PV into CV.

(4) Set the PV value as 5.

(5) Set the CV value as the random output variable (COUNT_CV).

(6) Set the Q value as the random output variable (COUNT_Q).

(7) Compile and write your program to the PLC after completing the program.

(8) After writing, change the PLC mode (Stop -> Run).

(9) If program runs, PV 5 will be loaded into CV (Count_CV).

(10) The current value CV (COUNT_CV) decreases by 1 when the pulse input enters the input contact (%I0.1.14).

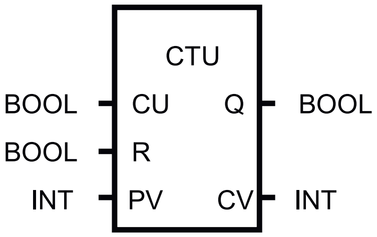
(11) When the down counter pulse input enters the input contact (%I0.1.14) five times, CV (COUNT_CV) will be 0 and Q (COUNT_CV) 1

(12) If Q (COUNT_Q) is 1, the output contact (%Q0.3.0) will be set.

CTU

Up Counter (function block)

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** CU: up counter pulse input R: reset input PV: loads a preset value **Output** Q: increase counter output CV: current value

■ Function

- • Up counter function block CTU increases the current value (CV) by 1 with every rising pulse input.
- • CV increases only when CV is less than the maximum value of INT (32767); after reaching it, CV does not change its value.
- • When the reset input (R) is 1, CV is cleared (0).
- • Output Q is 1 when CV is equal to or more than PV.

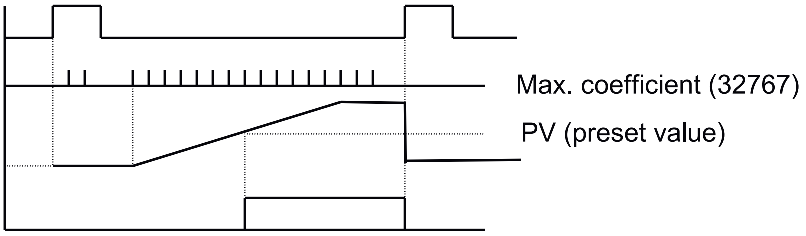
■ Time Chart

R (Reset input)

CU (CTU input)

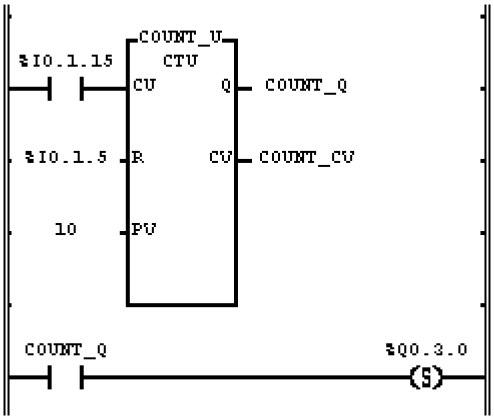
CV (current value)

Q (CTU output)



■ Program Example

This is the program that sets the output contact (%O0.3.1) when the increase counter pulse input enters the input contact (%I0.1.15) ten times.

LD	IL
	CAL CTU COUNT_U CU %I0.1.15 R %I0.1.5 PV 10 LD COUNT_V.Q ST COUNT_Q LD COUNT_CV.Q ST COUNT_CV LD COUNT_Q S %Q0.3.0

(1) Register the name of CTU function block (COUNT_U).

(2) Make the input contact %I0.1.15 attached to CU.

(3) Set the PV value as 10.

(4) Assign input contact %I0.1.5 to the reset input R.

(5) Set the CV value as the random output variable (COUNT_CV).

(6) Set the Q value as the random output variable (COUNT_Q).

(7) Compile and write your program to the PLC after completing the program.

(8) After writing, change the PLC mode (Stop - Run).

(9) The current value CV (COUNT_CV) increases by 1 when the pulse input enters the input contact (%I0.1.15).

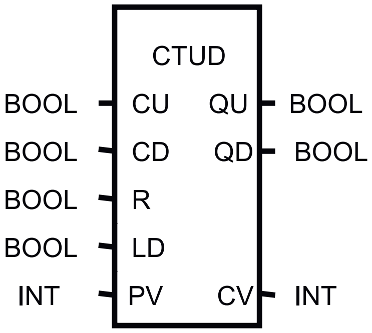
(10) When the up counter pulse input enters the input contact (%I0.1.15) ten times, CV (COUNT_CV) will be 10 and Q (COUNT_CV) 1

(12) If Q (COUNT_Q) is 1, the output contact (%Q0.3.0) will be set.

CTUD

Up/Down Counter (function block)

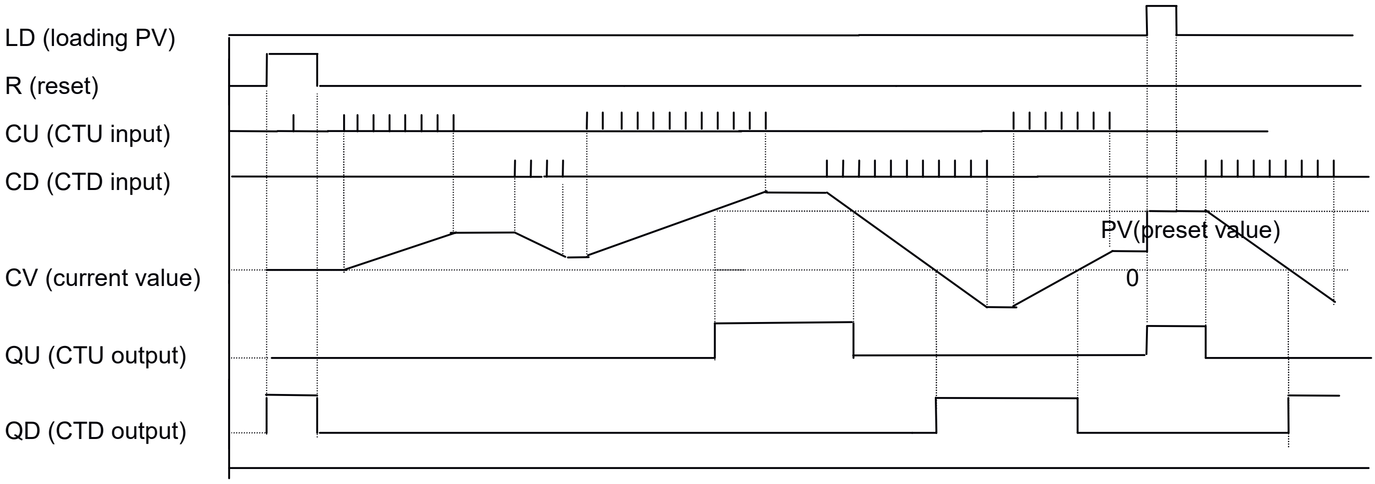
•••••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	••••	••••	• •	• •	• •	• •	• •

Function Block	Description
 CTUD BOOL CU QU BOOL BOOL CD QD BOOL BOOL R BOOL LD INT PV CV INT	**Input** CU: up counter pulse input CD: down counter pulse input R: reset LD: loads a preset value PV: preset value **Output** QU: up counter output QD: down counter output CV: current value

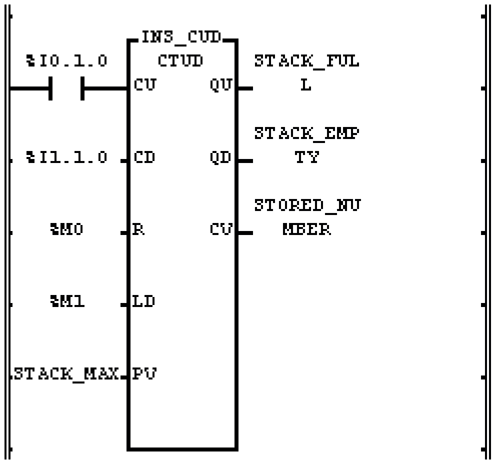
■ Function

- • Up/Down counter function block CTUD increases the current value (CV) by 1 with every rising up-counter pulse input (CU) and decreases CV by 1 with every rising down-counter pulse input (CD). Note that CV is between -32768 and 32767 (INT).
- • When LD is 1, PV is loaded into CV (CV=PV).
- • When the reset input R is 1, CV is cleared (0).
- • When CV reaches PV, the output QV is 1; when CV is 0 or a negative integer, the output QD is 1.
- • The operation for each input signal is executed in order of R > LD > CU > CD. Note that if the input signals are fed to the input (CU, CD, R, and LD) of CTUD at the same time, the operation of CTU follows the above priority.

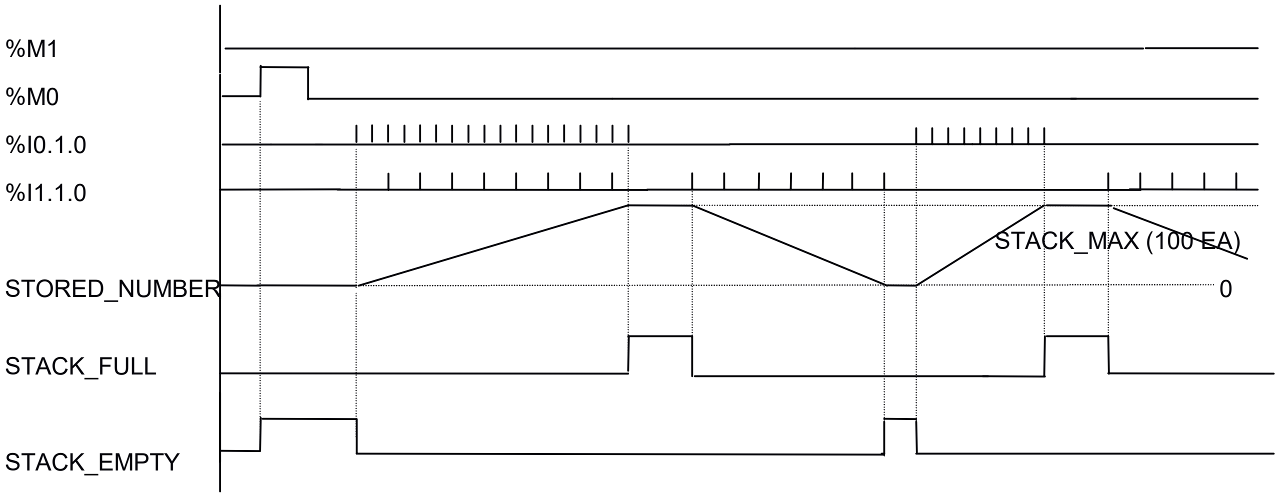
■ Time Chart



■ **Program Example**

LD	IL
 %I0.1.0 %I1.1.0 %M0 %M1 STACK_MAX INS_CUD CTUD CU CD R LD PV QU QD CV STACK_FULL STACK_EMPTY STORED_NUMBER	CAL CTUD INS_CUD CU:= %I0.1.0 CD:= %I1.1.0 R := %M0 LD:= %M1 PV:= STACK_MAX LD INS_CUD.QU ST STACK_FULL LD INS_CUD.QD ST STACK_EMPTY LD INS_CU.CV ST STORED_NUMBER

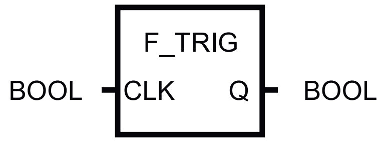
Conditions are: the temporary loading part STACK_MAX is 100; IN is 1 with every material-input signal while OUT is 1 with every material-output signal. If the material input process is faster than the material-output one and every material is loaded so that the STACK_MAX is equal to or more than 100, then QU is 1 (STACK_FULL = 1); if there's no material left in the loading part, QD is 1 (STACK_EMPTY = 1). At the STORED_NUMBER, the number of remaining material in the loading part is shown.



F_TRIG

Falling Edge Detection (function block)••

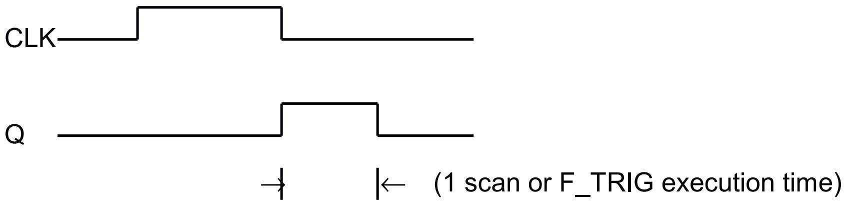
••••••	•••	•••••	•••••	•••••	•••••	•••••••
••••••••••••	• •	• •	•• ••	•• ••	••••	••••• •

Function Block	Description
 BOOL CLK F_TRIG Q BOOL	**Input** CLK: input signal **Output** Q: falling edge detection result

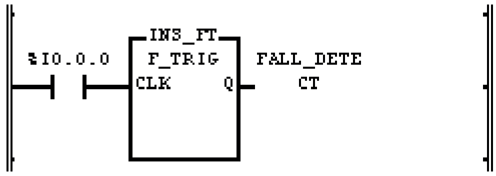
■ Function

The output Q of function block F_TRIG is 1 with the falling pulse input to CLK. And 1 scan later, without further falling pulse input, the output Q is 0 ever after.

■ Time Chart



■ Program Example

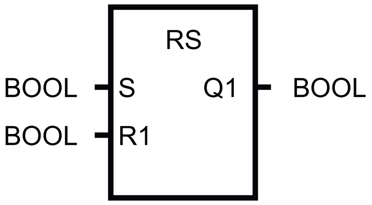
LD	IL
	CAL F_TRIG INS_FT CLK:= %I0.0.0 LD INS_FT.Q ST FALL_DETECT

If the input variable (%I0.0.0) changes from 1 to 0, while detecting its state, the output variable FALL_DETECT will be 1. And 1 scan later, the output variable FALL_DETECT will be 0.

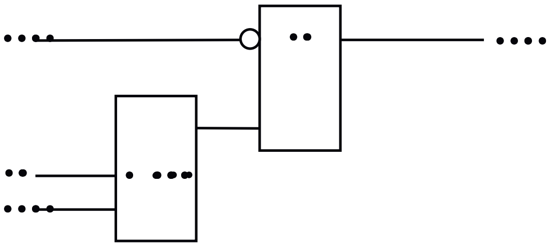
RS

Reset Priority Bistable (function block)

•••••••	••••	••••	••••	••••	••••	••••	••••	••••
••••••••••••	••••	••••	• •	• •	• •	• •	• •	• •

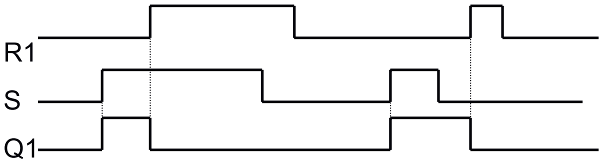
Function Block	Description
	**Input** R1: Reset condition S: Set condition **Output** Q1: Operation result

■ Function

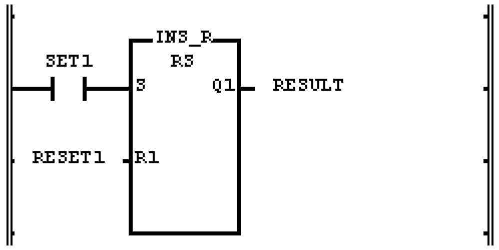


If R1 is 1, output Q1 will be 0 regardless of the state of S.
The output variable Q1 is 1 when it maintains the previous state, R1 is 0, and S is 1, it will be 1.
The initial state of Q1 is 0.

■ Time Chart



■ Program Example

LD	IL
	CAL RS INS_R R1: = RESET1 S: = SET1 LD INS_R.Q1 ST RESULT

(1) The output variable RESULT is 0 and maintains its value when the input variables SET1 and RESET1 become simultaneously ON.

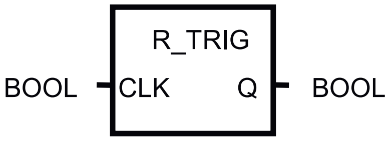
(2) The output variable RESULT is 0 and maintains its value when RESET1 becomes ON and SET1 is OFF.

(3) The output variable RESULT is 1 and maintains its value when SET1 becomes ON and RESET1 is OFF,

R_TRIG

Rising Edge Detection (function block)

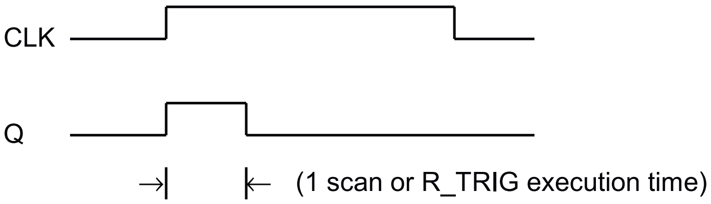
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** CLK: input signal **Output** Q: rising edge detection result

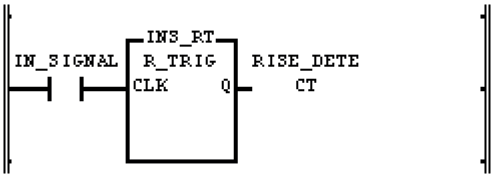
■ Function

The output Q of function block R_TRIG is 1 with the rising pulse input to CLK. And 1 scan later, without further falling pulse input, the output Q is 0 ever after.

■ Time Chart



■ Program Example

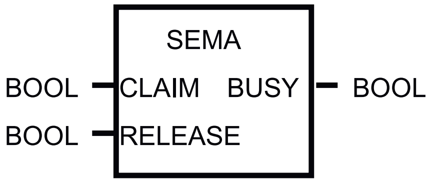
LD	IL
	CAL R_TRIG INS_RT CLK: = IN_SIGNAL LD INS_RT.Q ST RISE_DETECT

If the input variable IN_SIGNAL changes from 0 to 1, while detecting its state, the output variable RISE_DETECT will be 1. And 1 scan later, the output variable RISE_DETECT will be 0.

SEMA

Semaphore (System resource allocation)

•••••••	••••••••	••••••	•••••	•••••	•••••	•••••	••••••
•••••••••••	•••••••	•••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** CLAIM: signal to claim a resource monopoly RELEASE: release signal **Output** BUSY: waiting signal not to obtain the claimed resource

■ Function

This function block is used to get an exclusive control right for system resources.

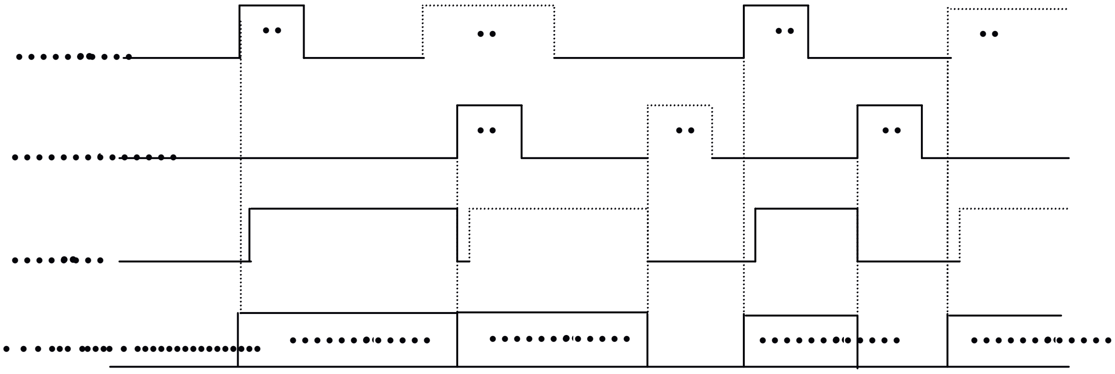
BUSY is 1 when SEMA function is executed (CLAIM = 1 or 0, RELEASE = 0) and other program is using the resource. If you want to obtain the resource control right, wait until BUSY will be 0 after executing SEMA function block (CLAIM = 1, RELEASE = 0). When BUSY is 0, it controls the associate resource and after completing the control, it transfers the control right executing SEMA function block once again with CLAIM = 0 and RELEASE = 1. (At this time, the program that has the control right can execute SEMA function block with CLAIM = 0 and RELEASE = 1)

- The instance of SEMA should be declared as "GLOBAL" so that its access is available in the programs requiring the resource.
- Each program to claim the same resource should be designated as the same priority.
- Not available to use between multi-CPU modules in GM1.
- Internal execution structure of SEMA function block

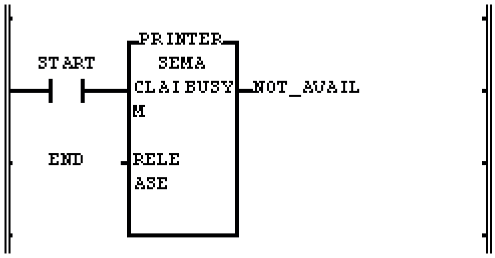
```
VAR   X : BOOL : = 0 ;   END_VAR
    BUSY : = X ;
    IF   CLAIM   THEN   X : = 1 ;
    ELSIF   RELEASE   THEN   BUSY : = 0; X : = 0 ;
    END_IF
```

■ Time Chart

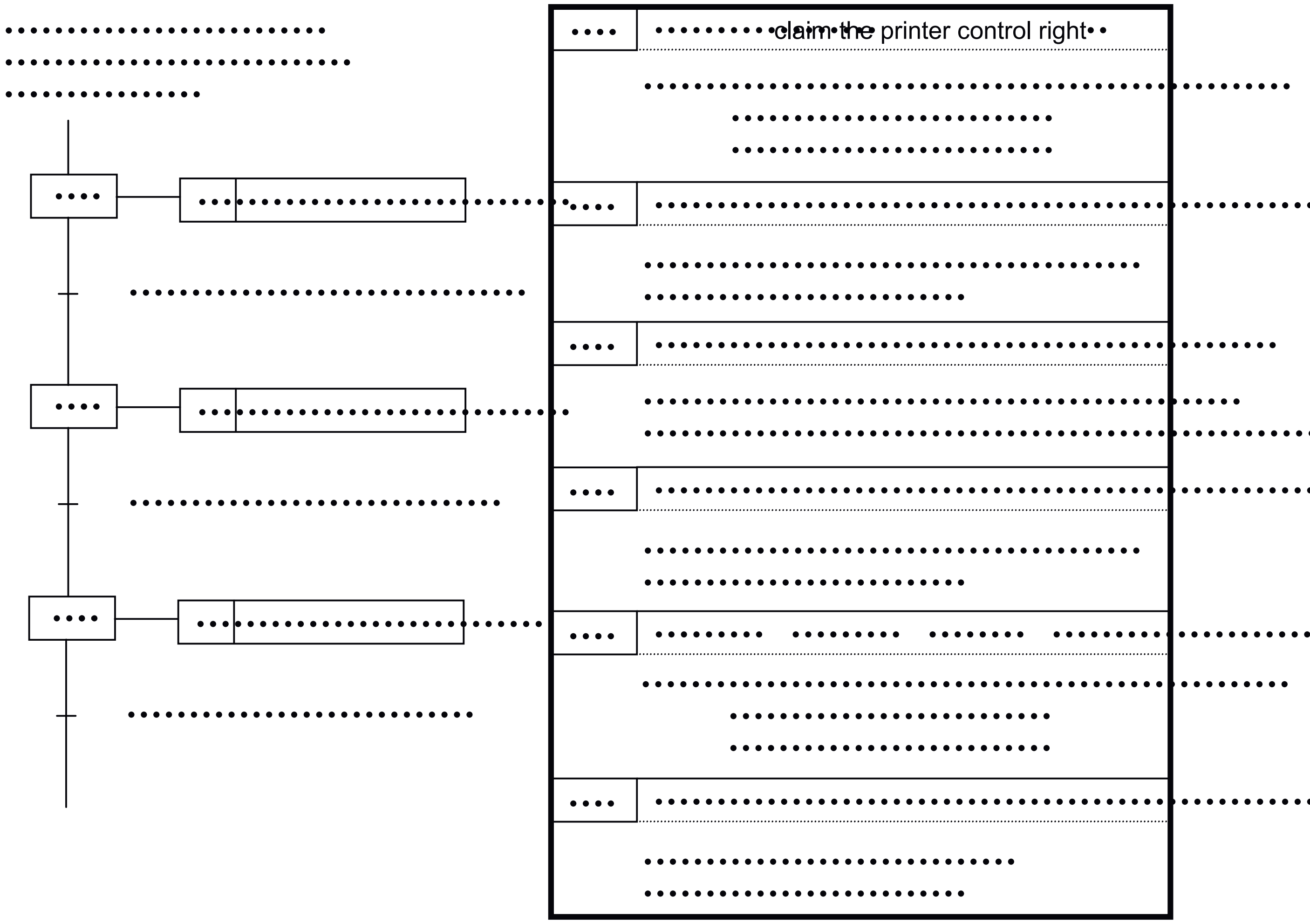
The access right to control the same resource is transferred between the program block A and the program block B.

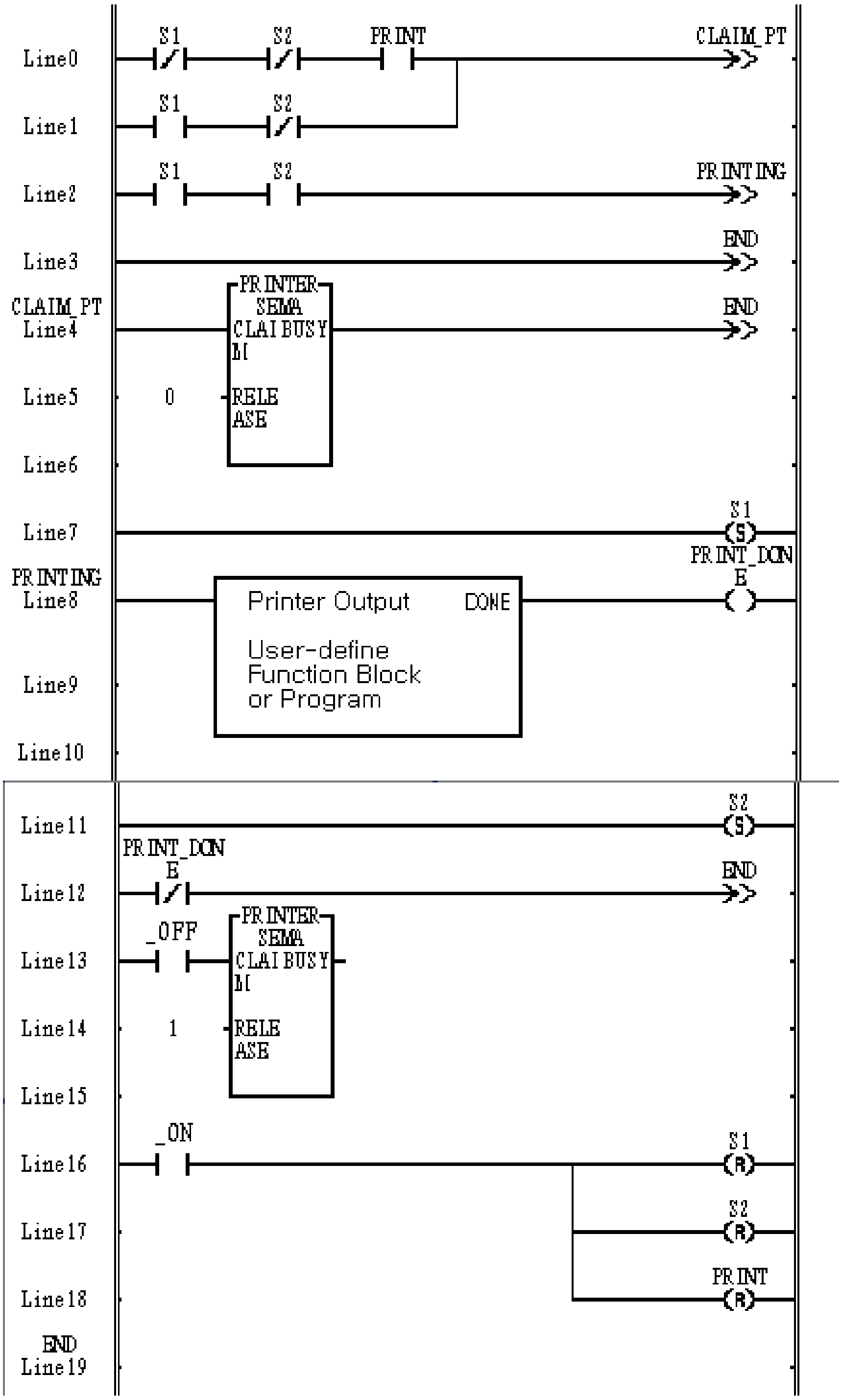


■ **Program Example**

LD	IL
 PRINTER SEMA START CLAIM BUSY NOT_AVAIL END RELEASE	CAL SEMA PRINTER CLAIM:= START RELEASE:= END LD PRINTER.BUSY ST NOT_AVAIL

When you want to produce a printer output in different program blocks with the printer attached to the PLC system, you can easily control it by declaring the instance 'PRINTER' 'GLOBAL' and using SEMA function block named as 'PRINTER' in each program. If you execute SEMA function block (PRINTER), when START is 1 and END is 0, and claim the right to control the printer, while the printer is used in other program block, BUSY is 1. If the printer is not used in other program block, BUSY will be 0, which means you can start the program to produce the printer output with it. After completing the print control, execute SEMA with START = 0 and END = 1 so that other program can get the right to control it.




Line0
S1
S2
PRINT
CLAIM_PT
Line1
S1
S2
Line2
S1
S2
PRINTING
Line3
END
CLAIM_PT
Line4
PRINTER-SEMA
CLAIBUSYM
RELEASE
END
Line5
0
Line6
Line7
S1
S
PRINTING
Line8
Printer Output
User-define Function Block or Program
DONE
PRINT_DONE
Line9
Line10
Line11
S2
S
Line12
PRINT_DONE
END
Line13
_OFF
PRINTER-SEMA
CLAIBUSYM
RELEASE
Line14
1
Line15
Line16
_ON
S1
R
Line17
S2
R
Line18
PRINT
R
END
Line19

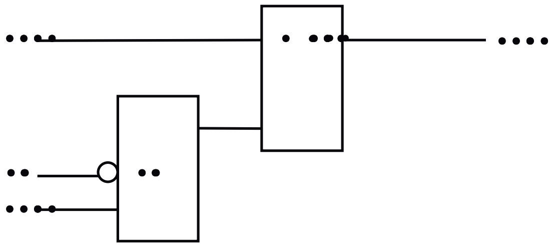
SR

Set Priority Bistable (function block)

•••••••••	•••••	•••••	•••••	•••••	•••••	•••••	••••••••
••••••••••••	•••••	•••••	• •	• •	• •	• •	• •

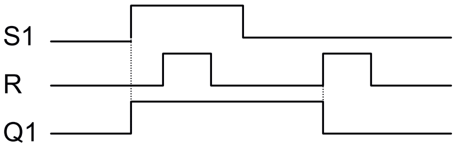
Function Block	Description
BOOL - S1 SR Q1 - BOOL BOOL - R	**Input** S1: set condition R: reset condition **Output** Q1: operation result

■ Function

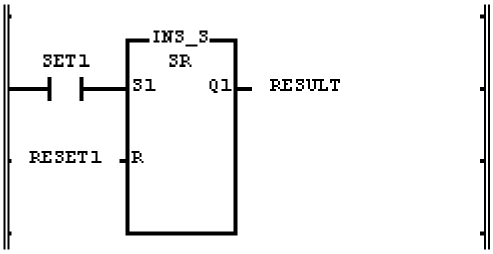


If S1 is 1, output Q1 will be 1 regardless of the state of R.
The output variable Q1 is 0 and it maintains the previous state when S1 is 0, and R is 1.
The initial state of Q1 is 0.

■ Time Chart



■ Program Example

LD	IL
	CAL SR INS_S S1: = SET1 R: = RESET1 LD INS_S.Q1 ST RESULT

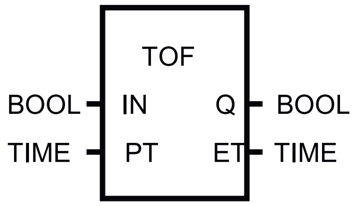
(1) If input variable SET1 becomes 1, output variable RESULT will be ON.

(2) The output variable RESULT becomes 0 when input variable SET1 becomes 0 and RESET1 ON.

TOF

OFF Delay Timer (function block)

•••••••••	••••••	•••••	••••••	•••••	•••••	•••••	•••••••
•••••••••••	••••••	••••	• •	• •	• •	• •	• •

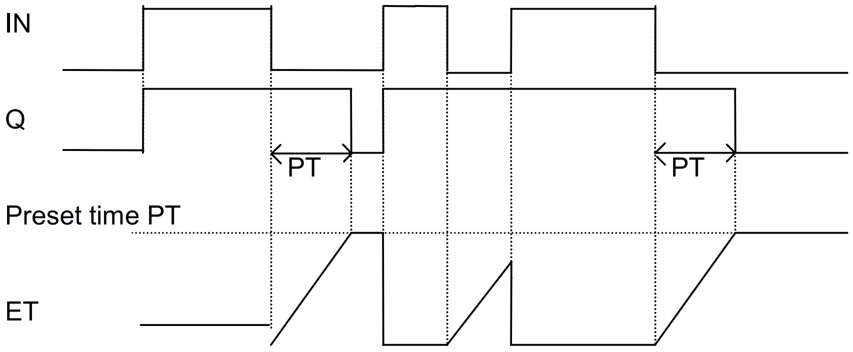
Function Block	Description
	**Input** IN: timer operation condition PT: preset time **Output** Q: timer output ET: elapsed time

■ Function

If IN is 1, Q will be 1. And after IN becomes 0 and the preset time (PT) of TOF passes, Q becomes 0.

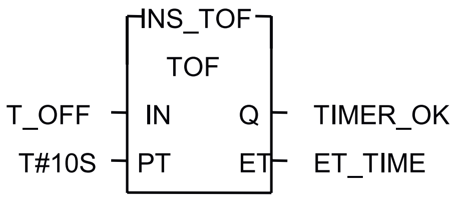
After IN becomes 0, the elapsed time (ET) will be shown. If IN becomes 1 before ET reaches the preset time, ET will be 0 again.

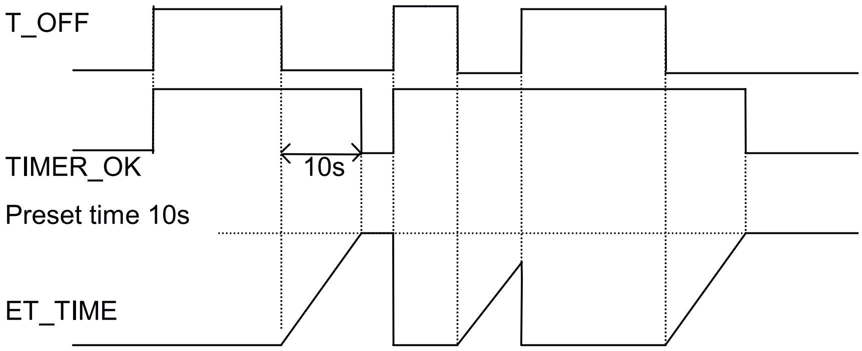
■ Time Chart



■ Program Example

LD	IL

 INS_TOF
TOF
T_OFF
IN
Q
TIMER_OK
T#10S
PT
ET
ET_TIME
	```
	CAL      TOF    INS_TOF
	         IN: =   T_OFF
	         PT: =    T#10S
	LD               INS_TOF.Q
	ST               TIMER_OK
	LD               INS_TOF.ET
	ST               ET_TIME
	```



(1) Output variable TIMER_OK is 1 when input variable T_OFF becomes 1.

(2) TIMER_OK is 0 only if 10 seconds passes after T_OFF becomes 0.

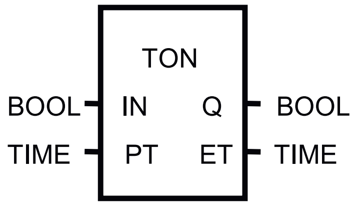
(3) If T_OFF becomes 1 again in 10 seconds after it turned OFF, TOF will be initialized (TIMER_OK is 1).

(4) After T_OFF becomes 0, the elapsed time (ET_TIME) will be measured and shown.

TON

ON Delay Timer (function block)

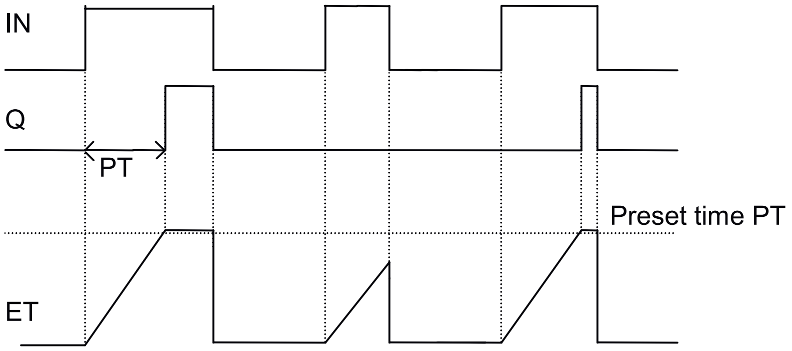
••••••••	•••••	•••••	•••••	•••••	•••••	•••••	•••••
••••••••••	•••••	••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: timer operation condition PT: preset time **Output** Q: timer output ET: elapsed Time

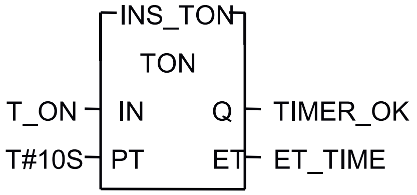
■ Function

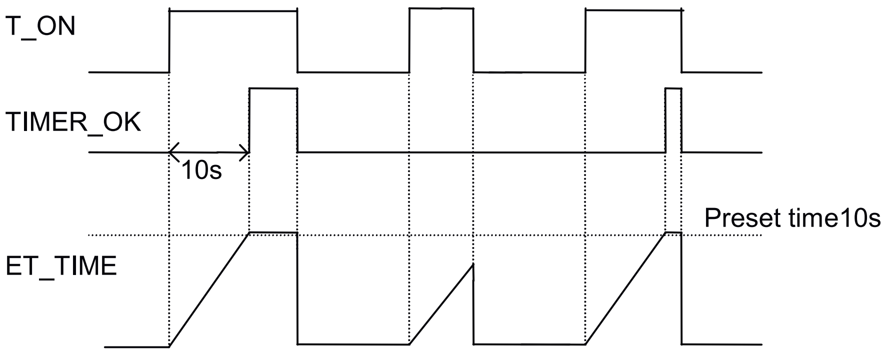
Elapsed time (ET) is measured and shown after IN becomes 1. When IN becomes 0 before ET reaches the preset time, ET will be 0. If IN becomes 0 after Q is 1, Q will be 0.

■ Time Chart



■ Program Example

LD	IL
 INS_TON TON T_ON IN Q TIMER_OK T#10S PT ET ET_TIME	CAL TON INS_TON IN: = T_ON PT: = T#10S LD INS_TON.Q ST TIMER_OK LD INS_TON.ET ST ET_TIME



(1) The output TIMER_OK = 1 ten seconds later after the input T_ON is asserted (T_ON = 1).

(2) Elapsed time ET_TIME is measured and shown after the input T_ON becomes 1.

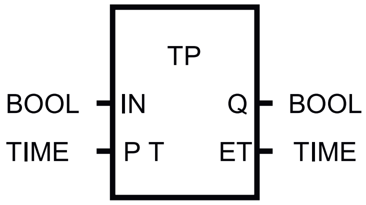
(3) When T_ON = 0 before ET_TIME reaches the preset time (10s), ET_TIME will be 0.

(4) If T_ON = 0 after TIMER_OK = 1, then TIMER_OK = 0 and ET_TIME = 0.

TP

Pulse timer (function block)

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** IN: timer operation condition PT: preset time **Output** Q: timer output ET: elapsed Time

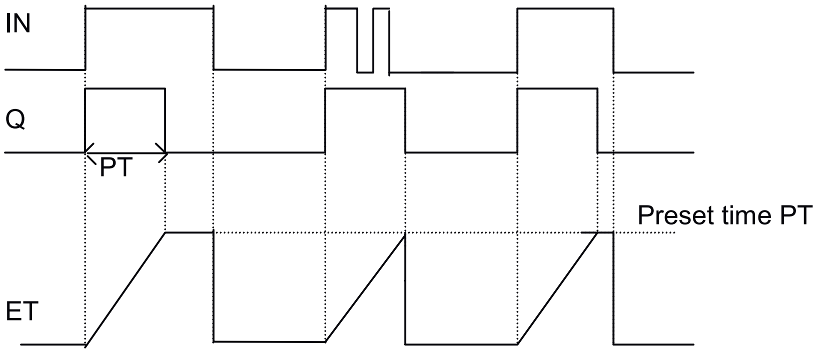
■ Function

If IN = 1, Q will be 1 only during the preset time PT; if ET reaches PT, Q will be 0.

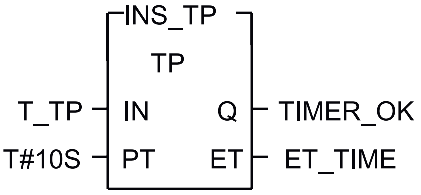
If IN = 1, elapsed time ET starts to be measured and maintains its value after when it reaches PT; if IN = 0 after ET reaches PT, ET = 0.

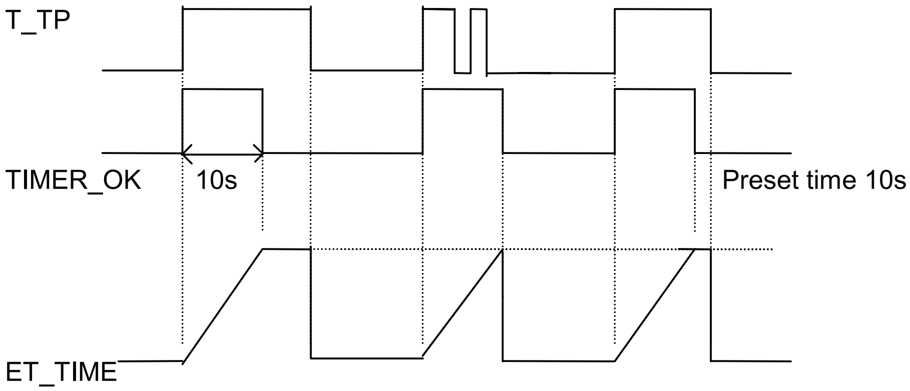
The state of IN doesn't matter while ET is measured (increased).

■ Time Chart



■ Program Example

LD	IL
	CAL TP INS_TP IN: = T_TP PT: = T#10S LD INS_TP.Q ST TIMER_OK LD INS_TP.ET ST ET_TIME



(1) TIMER_OK is 1 during 10 seconds after input T_TP was asserted (T_TP = 1). While ET_TIME increases during 10 seconds, the state of input T_TP doesn't affect TIMER_OK.

(2) ET_TIME increases when it reaches T#10S and then it becomes 0 when T_TP = 0.

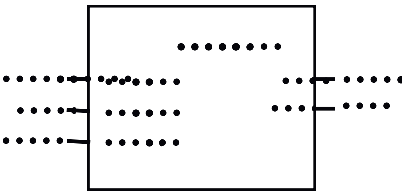
8.4 Application Function Block Library

1. This chapter describes each application function block library (MASTER-K and others).
2. It's much easier to apply function block library to your program after grasping the general of function blocks.

CTR

Ring Counter

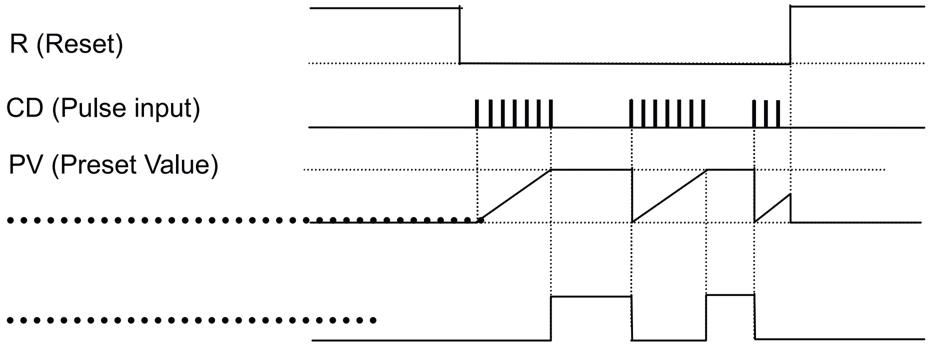
•••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	••••	••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** CD: pulse input of Ring Counter PV: preset value RST: reset **Output** Q: Ring Counter output CV: current value

■ Function

- • CTR function block (Ring Counter) functions: current value (CV) increases with the rising pulse input (CD) and if, after CV reaches PV, CD becomes 1, then CV is 1.
- • When CV reaches PV, output Q is 1.
- • If CV is less than PV or reset input (RST) is 1, output Q is 0.

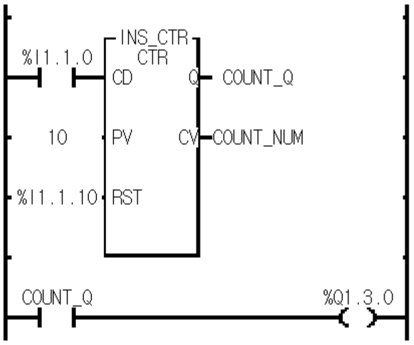
■ Timing Chart



■ **Program Example**

Output %Q1.3.1 is on with 10-time rising pulse input of %I1.1.0 is depicted as below.

LD

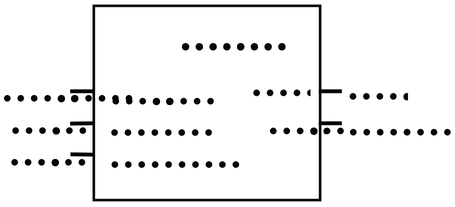


(1) Define CTR function block as INS_CTR.
(2) Set %I1.1.0 to the input contact of CD referring to the above.
(3) Set 10 to PV.
(4) Set %I1.1.10 to RST resetting CV.
(5) Set random variable COUNT_NUM to CV.
(6) Set random output variable COUNT_Q to Q.
(7) After a program is complete, compile and write it to PLC.
(8) When ‘Write’ is complete, do ‘Mode Change’ (Stop → Run).
(9) CV (COUNT_NUM) increases by 1 in number with the rising input pulse of %I1.1.0, CD
(10) With 10-time rising input pulse of input contact, CV is 10 which is the same as PV and output variable COUNT_Q is 1.
(11) If Q (COUNT_Q) is 1, output contact %Q1.3.0 is on.
(12) If the rising input pulse is loaded into input contact %I1.1.0, then Q (COUNT_Q) is 0 and output contact %Q1.3.0 is off.

DUTY

Scan setting On/Off

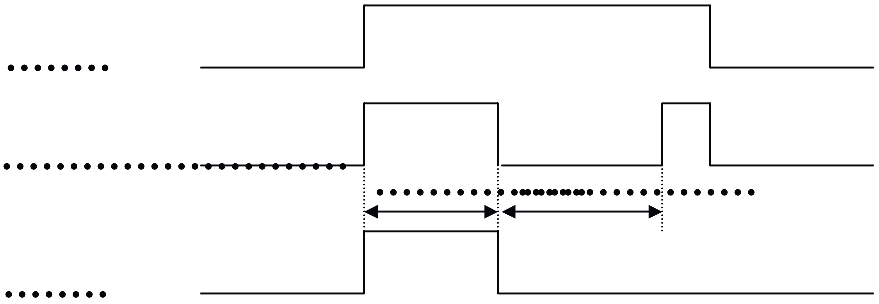
•••••••	••••••••	••••	••••••••	•••••	••••	•••••	••••••••
••••••••••••	••••••	••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** REQ: requires to execute the function block SON: scan number to turn on SOFF: scan number to turn off **Output** DONE: it is 1 when REQ is on and both input variables are not less than 0. OUT: output is 1 during on scan time

■ Function

- • DUTY function block produces a pulse which is on during the SON scan time and off during the SOFF scan time while REQ is on.
- • If SON = 0, OUT is always off.
- • If SON > 0 and SOFF = 0, OUT is always on.
- • If REQ is off, OUT is off.
- • If SON < 0 or SOFF < 0, then DONE is off and OUT is 0.

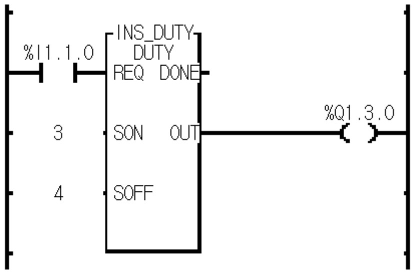
■ Timing Chart



■ **Program Example**

If input contact %I1.1.0 is set, output contact %Q1.3.0 is on during 3 scan times and off during 4 scan times.

LD

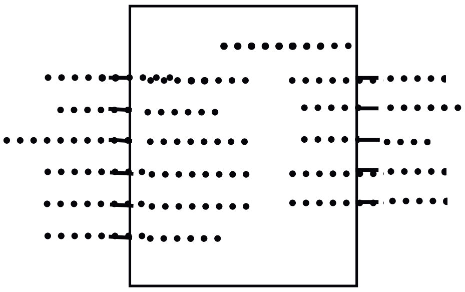


(1) Define DUTY function block as DUTY_C.
(2) Set %I1.1.0 to REQ (the input contact) of DUTY.
(3) Set 3 to SON.
(4) Set 4 to SOFF.
(5) Set %Q1.3.0 to output OUT.
(6) After a program is complete, compile and write it to PLC.
(7) When ‘Write’ is complete, do ‘Mode Change’ (Stop → Run).
(8) If input contact %I1.1.0 is on, output contact %Q1.3.0 is on during 3 scan times and off during 4 scan times.

FIFO_***

Load/Unload data to FIFO stack
(First In First Out)

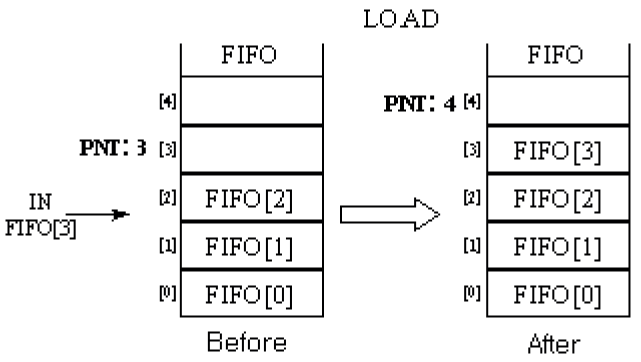
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

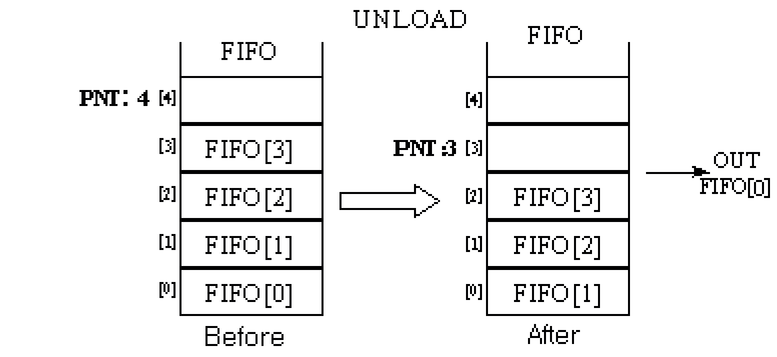
Function Block	Description
	**Input** REQ: requires to execute the function block IN: input data to be stored at FIFO stack LOAD: FB is on the input mode, if it’s on. UNLD: FB is on the output mode, if it’s on, RST: pointer value reset **Output** DONE: it’s 1 after first execution OUT: on output mode, it’s the data from FIFO stack PNT: pointer for input data of FIFO stack FULL: if FIFO stack is full, it’s 1 EMTY: if FIFO stack is empty, it’s 1 **In/Output** FIFO: array used as FIFO stack

■ Function

- • It loads IN to FIFO or unloads data from FIFO.
- • If Input and Output mode are set at the same time, it executes In/Output simultaneous.
- • If data is unloaded from FIFO, then the output is the lowest element of stack, the rest elements are shifts, PNT value is decreased by 1, and the element position of PNT is cleared (0).
- • If RST is loaded to FIFO, PNT is initialized as 0, EMTY is on and all the data of FIFO stack are cleared as 0.
- • The stack number is the input array number set by In/Output variable FIFO.
- • If you want to keep the data of FIFO array variables and FIFO function block instance in case that power is off or power failure occurs, set them as 'RETAIN'.
- • Reset functions without REQ input.
- • PNT shows the position of IN to be loaded next time, or the number of pointers to be loaded.
- • If it’s on the input mode, output OUT is 0.

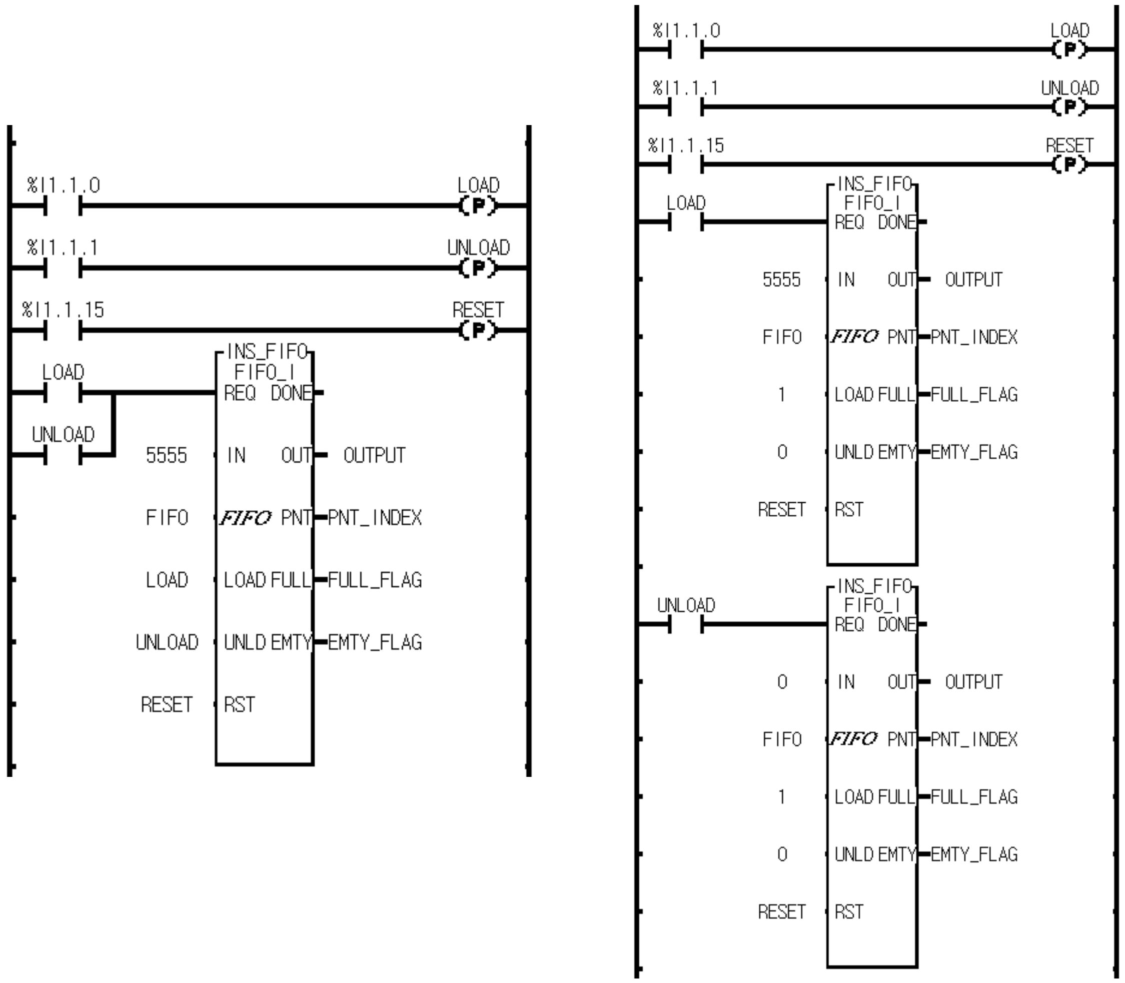
Function	FIFO variable type	Description
FIFO_Q	BOOL	It functions as FIFO for BOOL-type data
FIFO_B	BYTE	It functions as FIFO for BYTE-type data
FIFO_W	WORD	It functions as FIFO for WORD-type data
FIFO_DW	DWORD	It functions as FIFO for DWORD-type data
FIFO_LW	LWORD	It functions as FIFO for LWORD-type data
FIFO_SI	SINT	It functions as FIFO for SINT-type data
FIFO_I	INT	It functions as FIFO for INT-type data
FIFO_DI	DINT	It functions as FIFO for DINT-type data
FIFO_LI	LINT	It functions as FIFO for LINT-type data
FIFO_USI	USINT	It functions as FIFO for USINT-type data
FIFO_UI	UINT	It functions as FIFO for UINT-type data
FIFO_UDI	UDINT	It functions as FIFO for UDINT-type data
FIFO_ULI	ULINT	It functions as FIFO for ULINT-type data
FIFO_R	REAL	It functions as FIFO for REAL-type data
FIFO_LR	LREAL	It functions as FIFO for LREAL-type data
FIFO_TM	TIME	It functions as FIFO for TIME-type data
FIFO_DAT	DATE	It functions as FIFO for DATE-type data
FIFO_TOD	TOD	It functions as FIFO for TOD-type data
FIFO_DT	DT	It functions as FIFO for DT-type data





■ **Program Example**

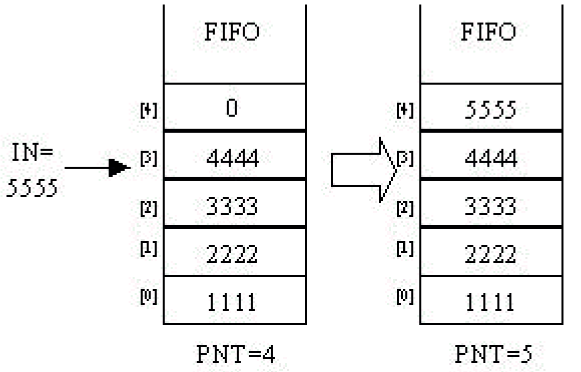
LD



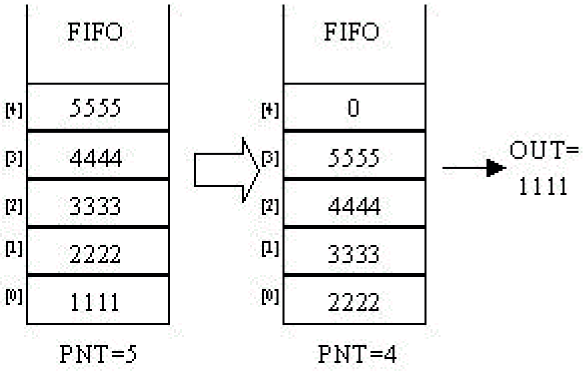
FIFO_*** function block is used as the above. The two examples of the above execute the same operation. The left one is a program which executes input and output functions at the same time to use only one function block while the right one is a program which executes input and output functions independently to use input function and output function respectively. Note that the instance name should be the same on the right program.

(1) If the input conditions (%I1.1.0, %I1.1.1, %I1.1.15) are on, FIFO_INT is executed.
(2) If input contact %I1.1.0 is on, load function is executed. 5555 is loaded to FIFO stack and PNT_INDEX increased by 1.
(3) If input contact %I1.1.1 is on, unload function is executed. 1111 is unloaded from FIFO stack and PNT_INDEX decreased by 1.
(4) If input contact %I1.1.15 is on, reset function is executed. All the stack of FIFO is cleared as 0, PNT_INDEX is initialized as 0 and EMTY_FLAG is on.

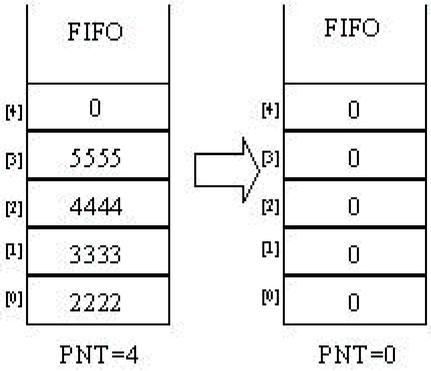
LOAD (%I1.1.0 is ON)



UNLOAD (%I1.1.1 is ON)



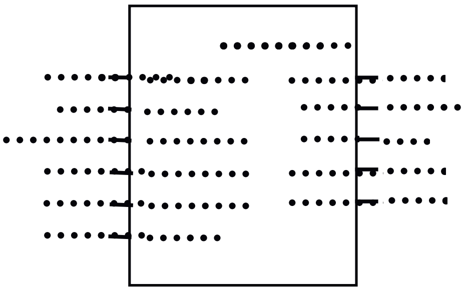
RESET (%I1.1.15 is ON)



LIFO_***

Load/Unload data to LIFO stack
(Last In First Out)

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

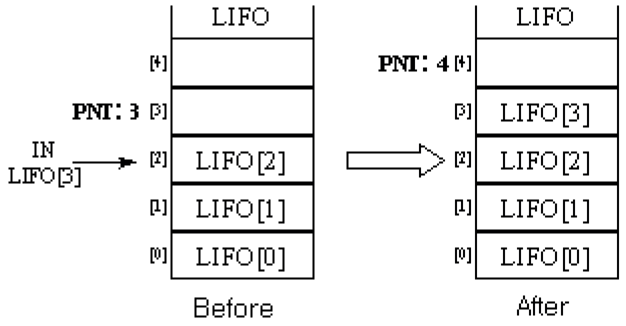
Function Block	Description
	**Input** REQ: requires to execute the function block IN: input data to be stored at LIFO stack LOAD: FB is on the input mode, if it’s on UNLD: FB is on the output mode, if it’s on RST: pointer value reset **Output** DONE: it’s 1 after first execution OUT: on output mode, it’s the data from LIFO stack PNT: pointer for input data of LIFO stack FULL: if LIFO stack is full, it’s 1 EMTY: if LIFO stack is empty, it’s 1 **In/Output** LIFO: array used as LIFO stack

■ Function

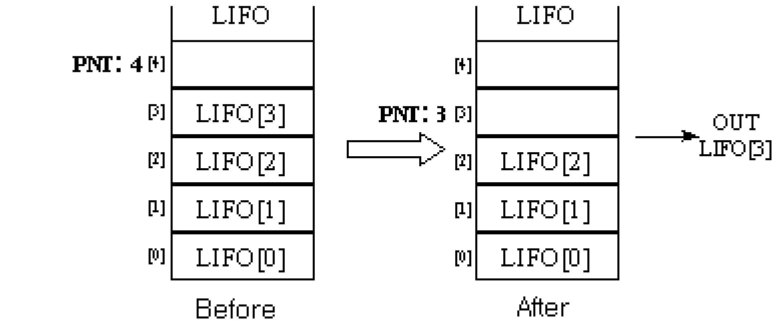
- • It loads IN to LIFO or unloads data from LIFO.
- • If LOAD and UNLD are on at the same time, input IN is produced as output OUT.
- • If data is unloaded from LIFO by unload function of LIFO_***, unloaded data is deleted in stack and initialized as 0.
- • If RST is loaded to LIFO, PNT is initialized as 0, EMTY is on and all the data of LIFO stack are cleared as 0.
- • The stack number is the array number set by In/Output variable LIFO.
- • If you want to keep the data of LIFO array variables and LIFO function block instance in case that power is off or power failure occurs, set them as 'RETAIN'.
- • Reset functions without REQ input.
- • PNT shows the position of IN to be loaded next time, or the number of pointers to be loaded.
- • If it's on the input mode, output OUT is 0.

Function	FIFO variable type	Description
LIFO_Q	BOOL	It functions as LIFO for BOOL-type data
LIFO_B	BYTE	It functions as LIFO for BYTE-type data
LIFO_W	WORD	It functions as LIFO for WORD-type data
LIFO_DW	DWORD	It functions as LIFO for DWORD-type data
LIFO_LW	LWORD	It functions as LIFO for LWORD-type data
LIFO_SI	SINT	It functions as LIFO for SINT-type data
LIFO_I	INT	It functions as LIFO for INT-type data
LIFO_DI	DINT	It functions as LIFO for DINT-type data
LIFO_LI	LINT	It functions as LIFO for LINT-type data
LIFO_USI	USINT	It functions as LIFO for USINT-type data
LIFO_UI	UINT	It functions as LIFO for UINT-type data
LIFO_UDI	UDINT	It functions as LIFO for UDINT-type data
LIFO_ULI	ULINT	It functions as LIFO for ULINT-type data
LIFO_R	REAL	It functions as LIFO for REAL-type data
LIFO_LR	LREAL	It functions as LIFO for LREAL-type data
LIFO_TM	TIME	It functions as LIFO for TIME-type data
LIFO_DAT	DATE	It functions as LIFO for DATE-type data
LIFO_TOD	TOD	It functions as LIFO for TOD-type data
LIFO_DT	DT	It functions as LIFO for DT-type data

LOAD

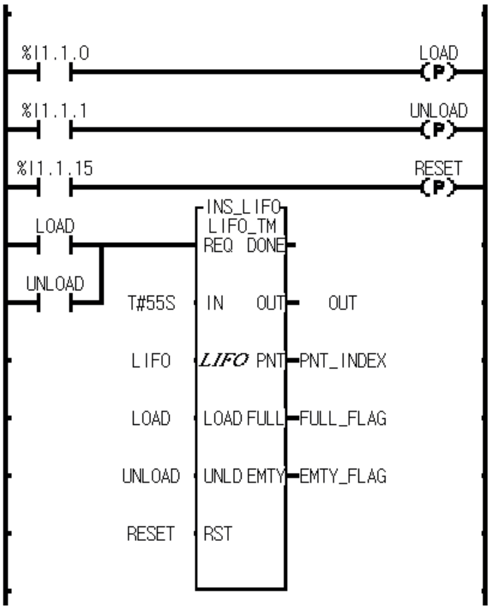


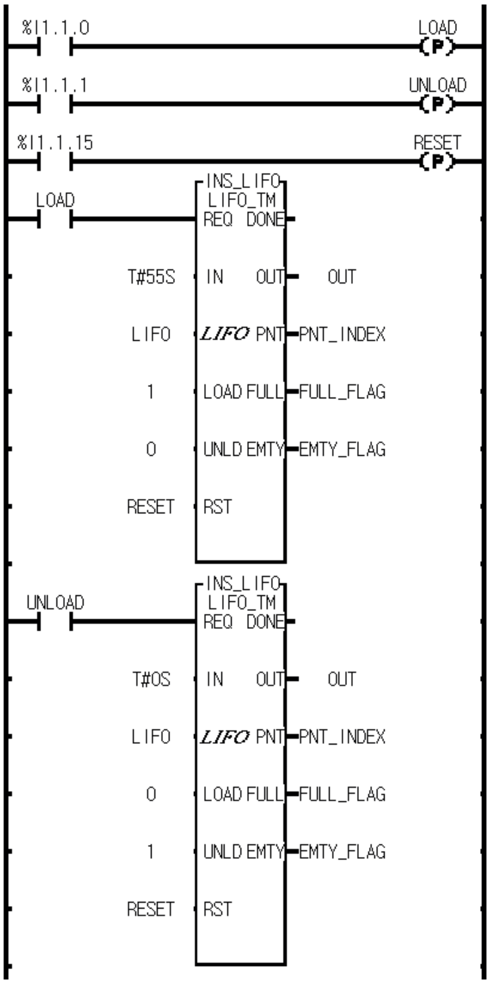
UNLOAD



■ **Program Example**

LD

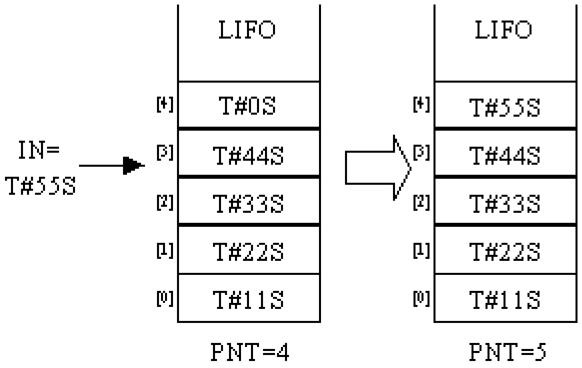




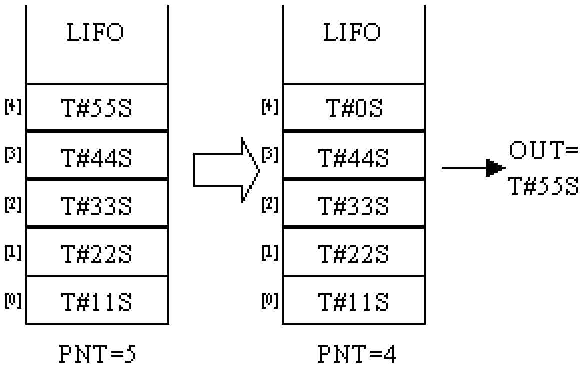
LIFO_*** function block is used as the above. The two examples of the above execute the same operation. The left one is a program which executes input and output functions at the same time to use only one function block while the right one is a program which executes input and output functions independently to use input function and output function respectively. Note that the instance name should be the same on the right program.

(1) If the input conditions (%I1.1.0, %I1.1.1, %I1.1.15) are on, LIFO_TM is executed.
(2) If input contact %I1.1.0 is on, load function is executed. T#55S is loaded to LIFO stack and PNT_INDEX increased by 1.
(3) If input contact %I1.1.1 is on, unload function is executed. T#55S is unloaded from LIFO stack and PNT_INDEX decreased by 1.
(4) If input contact %I1.1.15 is on, reset function is executed. All the stack of LIFO is cleared as T#0S, PNT_INDEX is initialized as 0 and EMTY_FLAG is on.

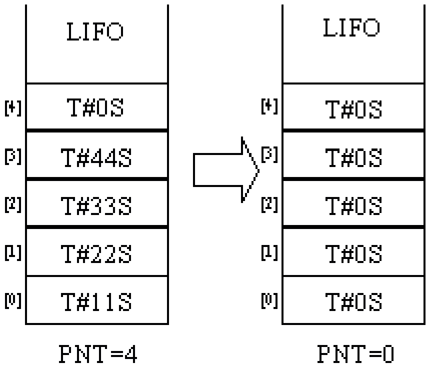
LOAD (%I1.1.0 is ON)



UNLOAD (%I1.1.1 is ON)



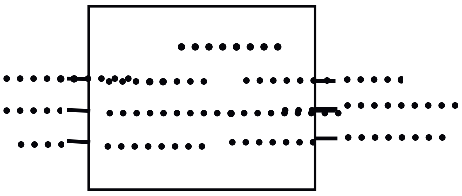
RESET (%I1.1.15 is ON)



SCON

Step Controller

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** REQ: if it's 1, the function block is executed S/O: if 0, SET function is enabled; if 1, OUT function is enabled. SET: step number (0 ~ 99) **Output** DONE: without an error, it will be 1 S: produces an set bit array CUR_S: produces a current step number

■ Function

• • Setting of step controller group
- The instance name of function block is the name of step controlling group.
  (Examples of FB declaration: S00, G01, Manu1
  Examples of step contacts: S00.S[1], G01.S[1], Manu1.S[1])

• • In case of SET function (ST_0/JP_1 = 0)
- In the same step controller group, the present step number can be on when the previous step number is on.
- If the present step number is on, it keeps its state even when the input is off.
- Only one step number is on even when several input conditions are on at the same time.
- If Sxx.S[0] is on, all the SET output is cleared.

• • In case of JUMP function (ST_0/JP_1 = 1)
- In the same step controller group, only one step number is on, even when several input conditions are on.
- If input conditions are on at the same time, last programmed one is produced.
- If the present step number is on, it keeps its state even when the input is off.
- If Sxx.S[0] is on, it returns to its first step.

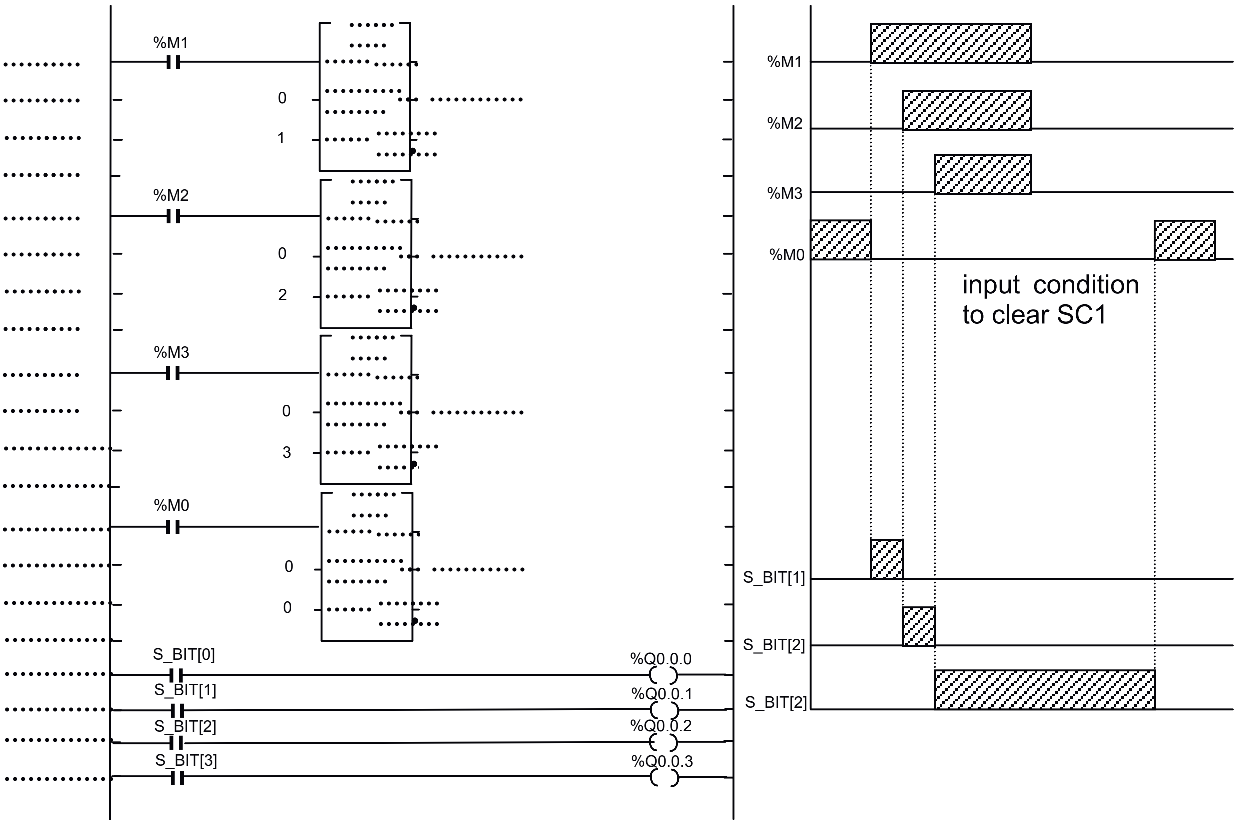
■ Error

• • An error occurs when step setting (SET) is out of its range (0 ~ 99).

• • If an error occurs, DONE is off and step output maintains its previous step.

■ Program Example

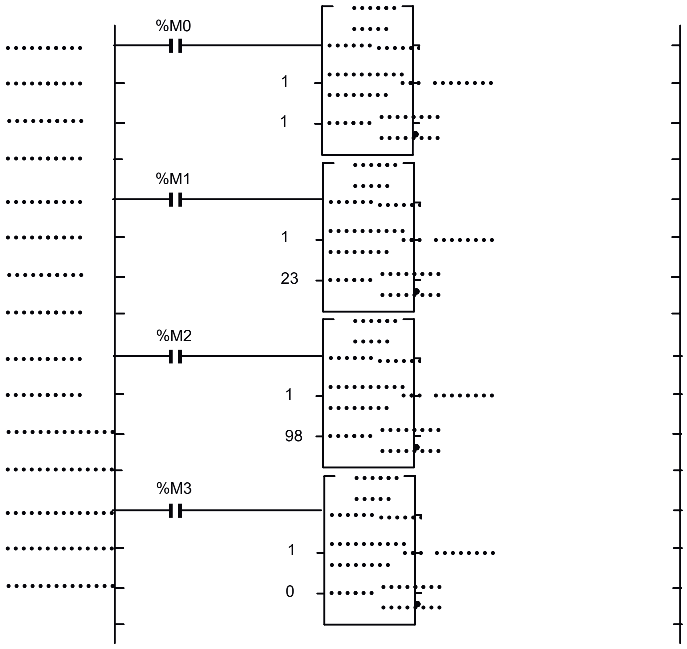
In case of SET function (ST_0/JP_1 = 0), using SC1 group



Step control produces an output when the previous step is on and its present condition is on.

■ **Program Example**

In case of JUMP function (ST_0/JP_1 = 1), using SC2 group (last input priority)

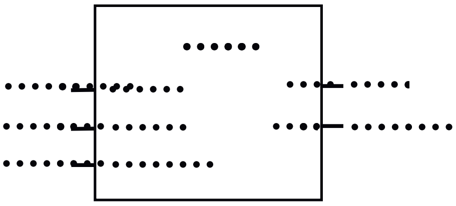


NO	%M1	%M2	%M3	%M4	S_O[1]	S_O[23]	S_O[98]	S_O[0]
1	On	Off	Off	Off	O			
2	On	On	Off	Off		O		
3	On	On	On	Off			O	
4	On	On	On	On				O

TMR

Integration Timer

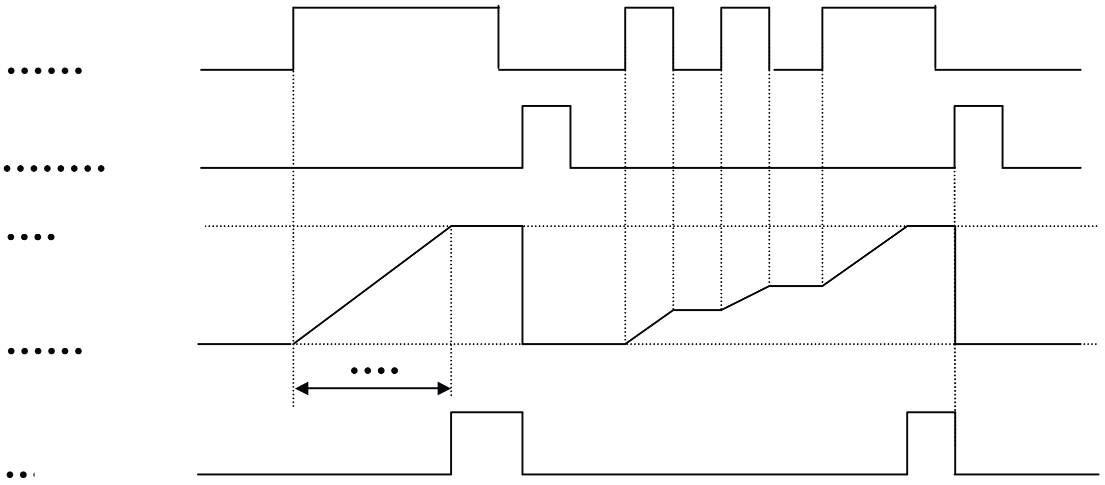
•••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	••• ••••	••• •••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time RST: reset **Output** Q: timer output ET: elapsed time

■ Function

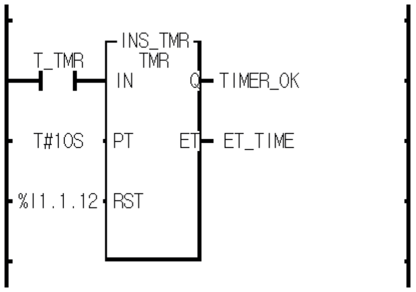
- • When IN is 1, elapsed time is produced at ET.
- • Even if IN is 0 before ET reaches PT, ET keeps its value. If IN is 1 again, elapsed time is produced at ET integrating its previous value.
- • If ET reaches PT, Q is 1..
- • If RST is 1, Q and ET are 0.

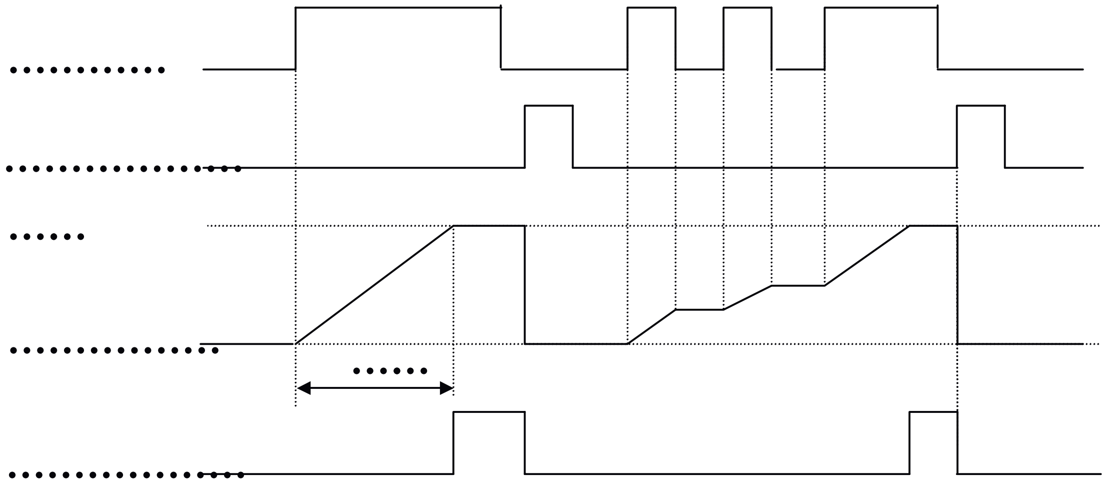
■ Timing Chart



■ **Program Example**

LD



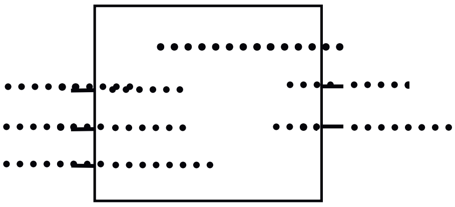


(1) If 10 seconds passes after input variable T_TMR is 1, output variable TIMER_OK is 1.
(2) Elapsed time is produced at ET_TIME after T_TMR is 1.
(3) ET_TIME keeps its value even if T_TMR is 0 before ET_TIME reaches its preset time 10 seconds.
(4) If T_TMR is 1, elapsed time is produced at ET_TIME integrating its previous value.
(5) If input contact %I1.1.12 is 1, elapsed time ET_TIME and output variable TIMER_OK are all cleared.

TMR_FLK

TMR with Flicker

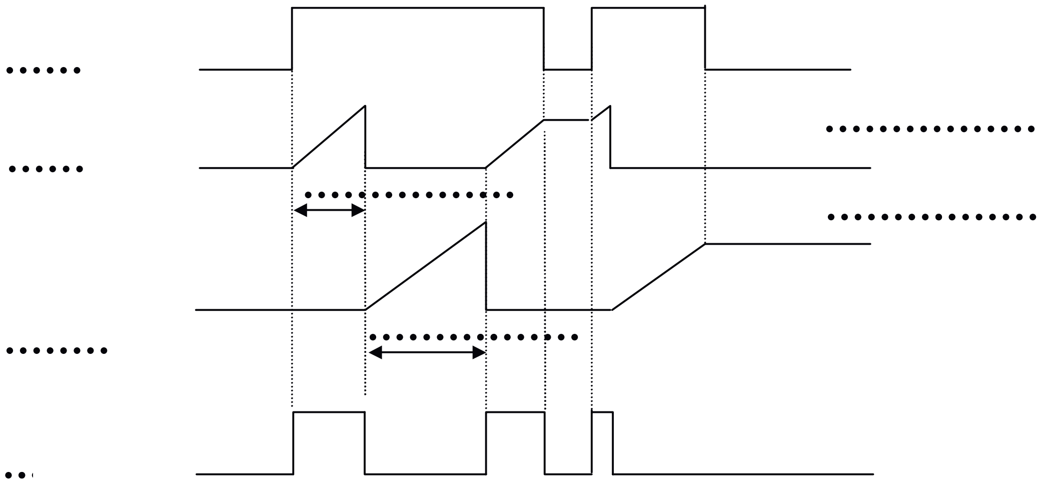
•••••••	•••••	•••••	•••••	•••••	•••••	•••••	•••••••
•••••••••••••	••••••	••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer ON: TON setting time OFF: TOF setting time **Output** Q: Timer output ET: elapsed time

■ Function

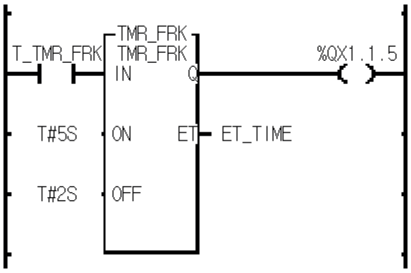
- • If IN is 1, Q is 1 and maintains its value during TON setting time.
- • After TON setting time set by ON, Q is 0 during TOF setting time.
- • If IN is 0, it stops its function of either on or off operation and keeps its time. If IN is 1 again, it is executed with its previous data.
- • Output Q is 0 while IN is 0.
- • If ON is 0, output Q is always 0.

■ Timing Chart



■ **Program Example**

LD

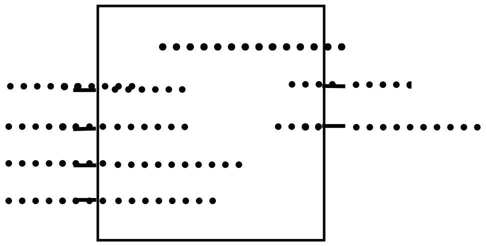


(1) If input variable T_TMR_FRK is 1, TMR_FRK function block is executed.
(2) Output contact %QX1.1.5 is 1 during 5 seconds set by ON after input variable T_TMR_FRK is 1.
(3) Output contact %QX1.1.5 is 0 during 2 seconds set by OFF after 5 seconds set by ON.
(3) TON time (ON) when Q is 1 and TOF time (OFF) when Q is 0 are produced at ET_TIME by turns while T_TMR_FRK is 1.
(4) If input variable T_TMR_FRK is 0, then it keeps its time and output contact %QX1.1.5 is 0. If T_TMR_FRK is 1, it is executed again.

TMR_UNIT

TMR with integer setting

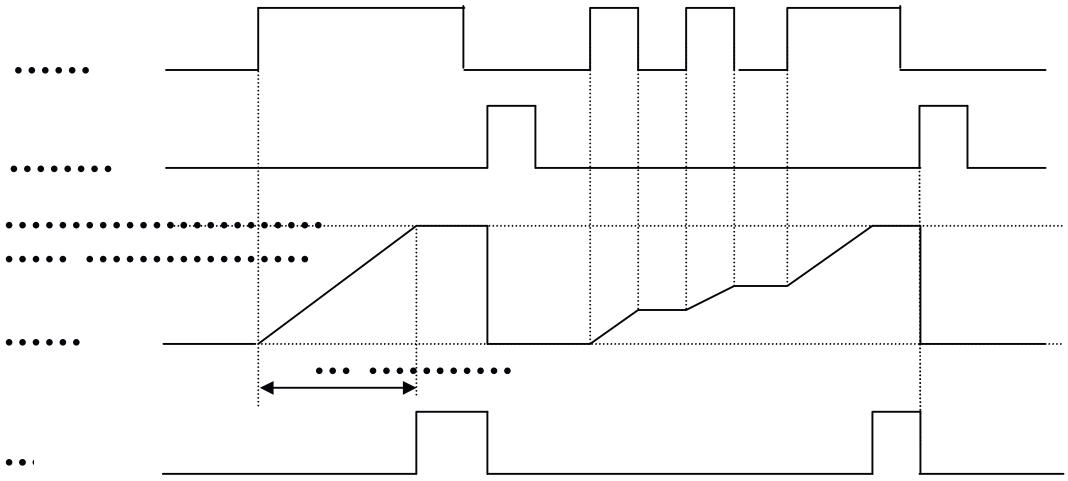
••••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••••	•••••••	••••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time UNIT: time unit of setting time RST: reset input **Output** Q: timer output ET: elapsed time

■ Function

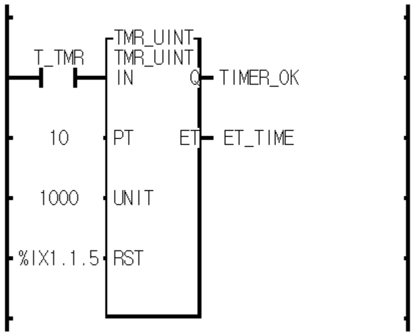
- • Elapsed time is produced at ET after IN is 1.
- • Even if IN is 0 before ET reaches PT, ET keeps its value. If IN is 1 again, elapsed time is produced at ET integrating its previous value.
- • Q is 1 when elapsed time reaches preset time.
- • If RST is 1, Q and ET are 0.
- • Setting time is PT x UNIT (ms).

■ Timing Chart

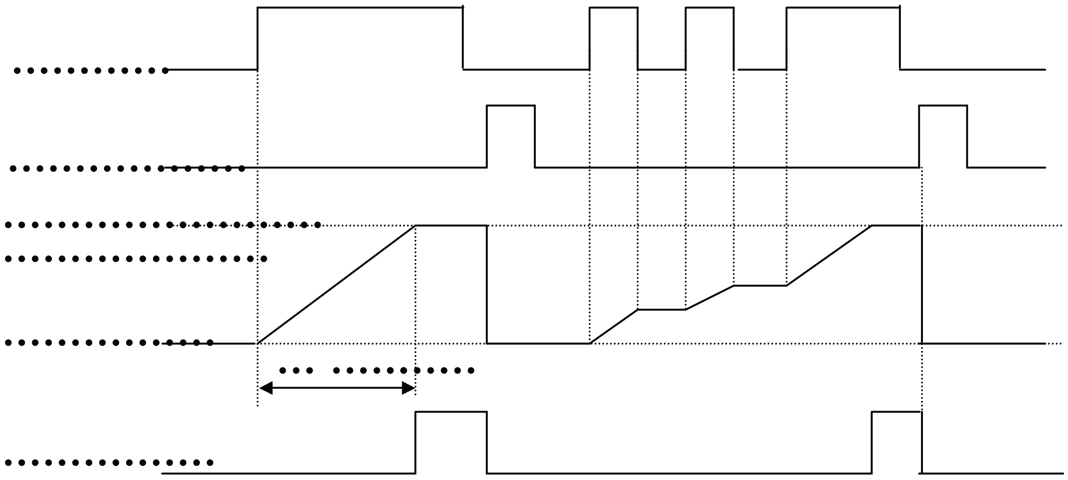


■ **Program Example**

LD



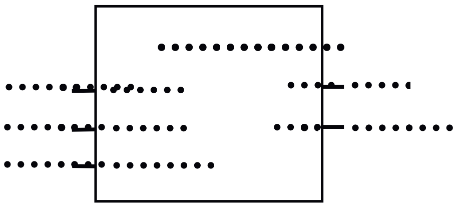
(1) Setting time is PT x UNIT[ms] = 10 x 1000[ms] = 10[s].
(2) Output variable TIMER_OK is 1, if 10 seconds passes after input variable T_TMR is 1.
(3) Elapsed time is produced at ET_TIME after input variable T_TMR is 1.
(4) Even if T_TMR is 0 before ET_TIME reaches preset time 10 seconds, ET_TIME keeps its value.
(5) If input variable T_TMR is 1 again, elapsed time is produced at ET integrating its previous value.
(6) If input contact %IX1.1.5 is 1, elapsed time ET_TIME and output contact TIMER_OK are all cleared.



TOF_RST

TOF with Reset

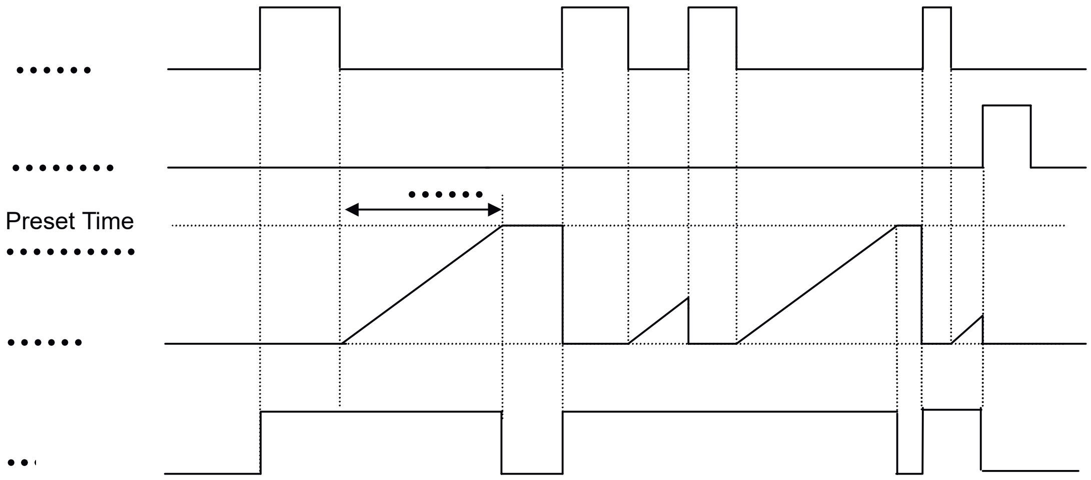
•••••••	••••••••••••••••••••••••••••••					
••••••••••••	••••••	••••••	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time RST: reset **Output** Q: Timer output ET: elapsed time

■ Function

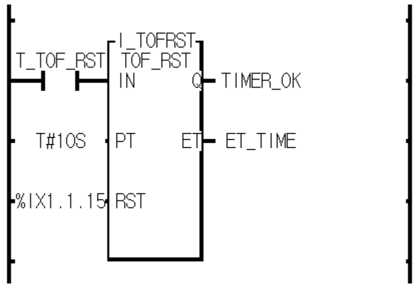
- • Q is 1 when IN is 1 and Q is 0 after preset time (PT) after IN is 0.
- • Elapsed time is produced at ET after IN is 0.
- • Elapsed time is 0 if IN is 1 before ET reaches PT.
- • If RST is 1, Q and ET are 0.

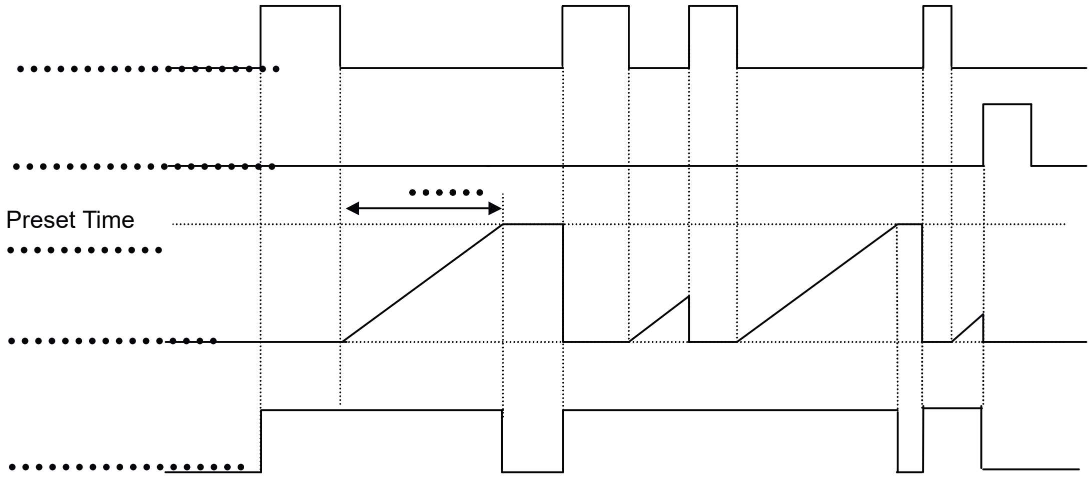
■ Timing Chart



■ **Program Example**

LD



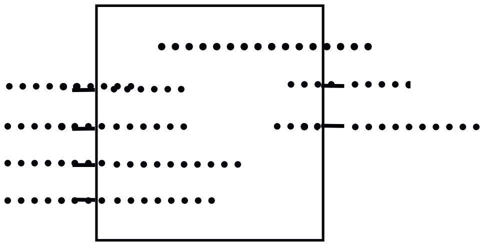


(1) If input variable T_TOF_RST is 1, output variable TIMER_OK is 1. And TIMER_OK is 0 after 10 seconds after T_TOF_RST is 0.
(2) If T_OF_RST is 1 within 10 seconds after it turns off, TOF_RST is initialized.
(3) Elapsed time is produced at ET_TIME.
(4) If input contact %IX1.1.15 is 1, elapsed time ET_TIME and output contact TIMER_OK are all cleared.

TOF_UINT

TOF with integer setting

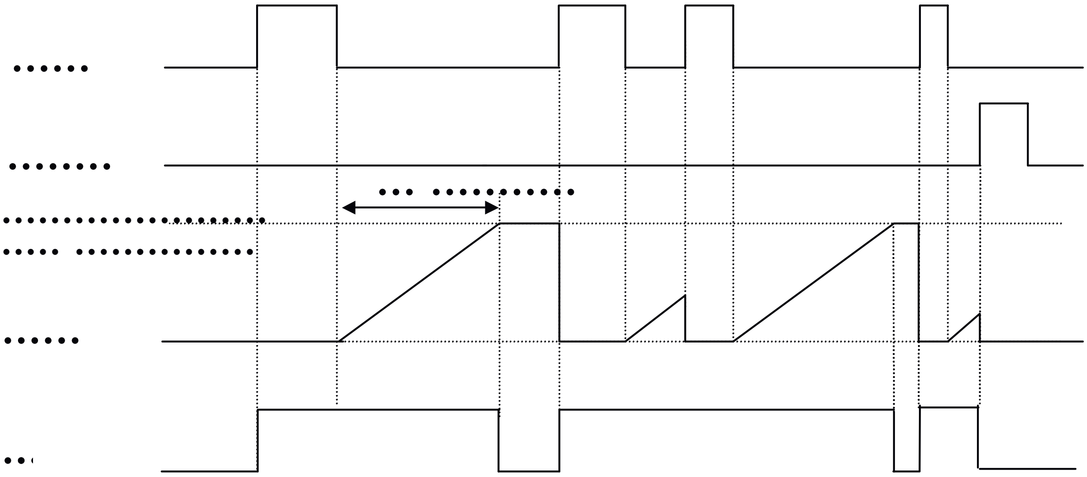
•••••••	••••	••••	••••	••••	••••	••••	••••
•••••••••••••	•• •••	••• •••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time UNIT: time unit of setting time RST: reset **Output** Q: Timer output ET: elapsed time

■ Function

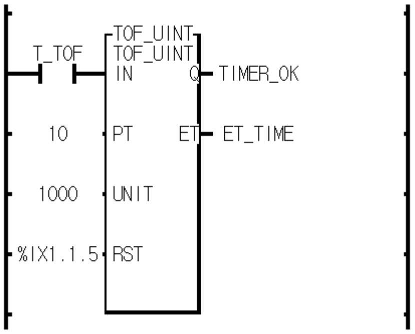
- • Q is 1 when IN is 1. And Q is 0, if setting time (PT) passes after IN is 0.
- • Elapsed time is produced at ET after IN is 0.
- • If IN is 1 before ET reaches PT, ET is 0.
- • If RST is 1, Q and ET are 0.
- • Setting time is PT x UNIT (ms).

■ Timing Chart



■ **Program Example**

LD



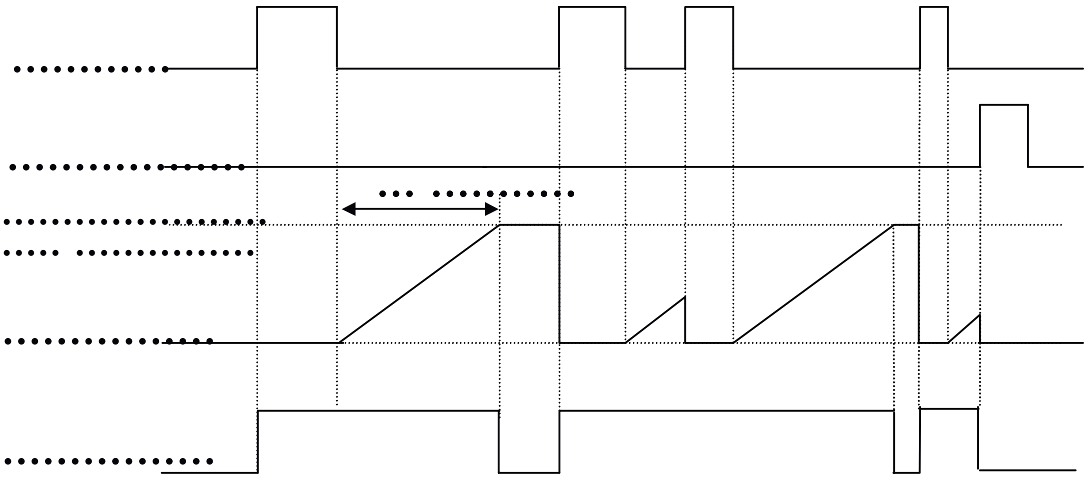
(1) Preset time PT x UNIT[ms] = 10 x 1000[ms] = 10[s].

(2) If input variable T_TOF is 1, output variable TIMER_OK is 1. TIMER_OK is 0, if 10 seconds passes after T_TOF is 0.

(3) If T_TOF is 1 within 10 seconds, TOF_UINT is initialized.

(4) Elapsed time is produced at ET_TIME.

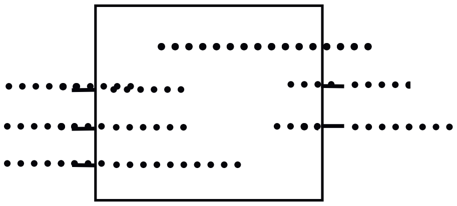
(5) If input contact %IX1.1.5 is 1, TIMER_OK and ET_TIME are all cleared.



TON_UINT

TON with integer setting

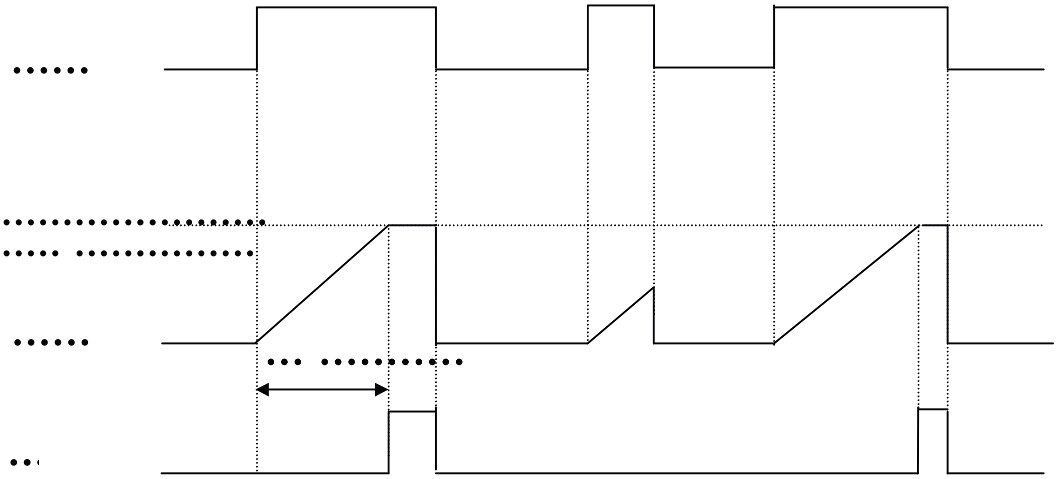
[illegible]	[illegible]						
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time UNIT: time unit of setting time **Output** Q: timer output ET: elapsed time

■ Function

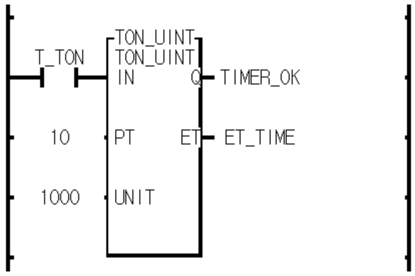
- • Elapsed time is produced at ET after IN is 1.
- • Elapsed time ET is 0, if IN is 0 before ET reaches PT.
- • Q is 0, if IN is 0 after Q is 1.
- • Preset time is PT x UNIT[ms].

■ Timing Chart

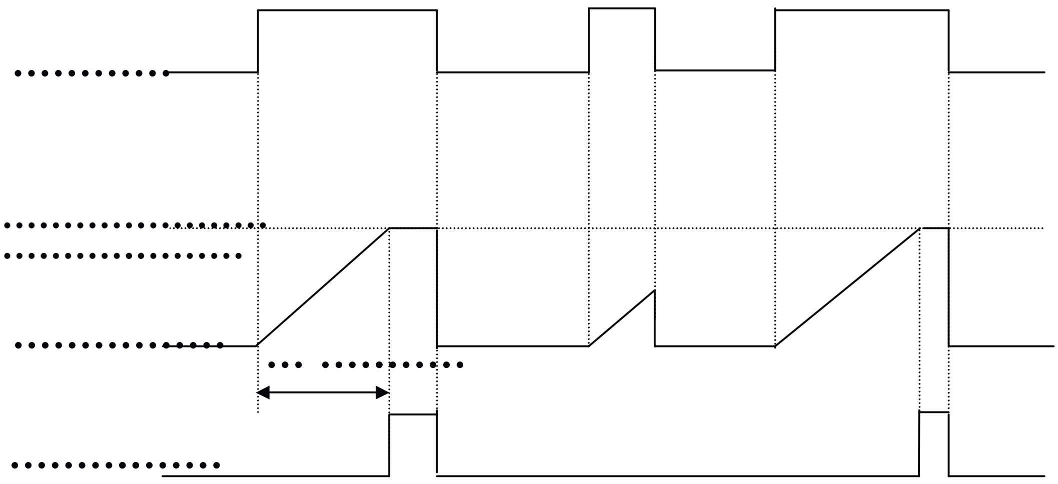


■ **Program Example**

LD



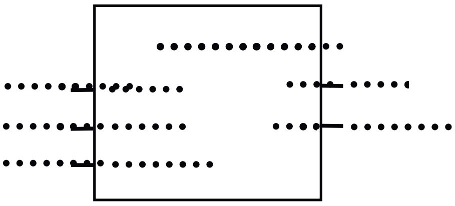
(1) Preset time is PT x UNIT[s] = 10 x 1000[s] = 10[s].
(2) If 10 seconds passes after input variable T_TON is on, output variable TIMER_OK is 1.
(3) Elapsed time is produced at ET_TIME after input variable T_TON is on.
(4) If T_TON is 0 before elapsed time ET_TIME reaches 10 seconds, ET_TIME is 0.
(5) If T_TON is 0 after TIMER_OK is 1, TIMER_OK and ET_TIME are 0.



TP_RST

TP with Reset

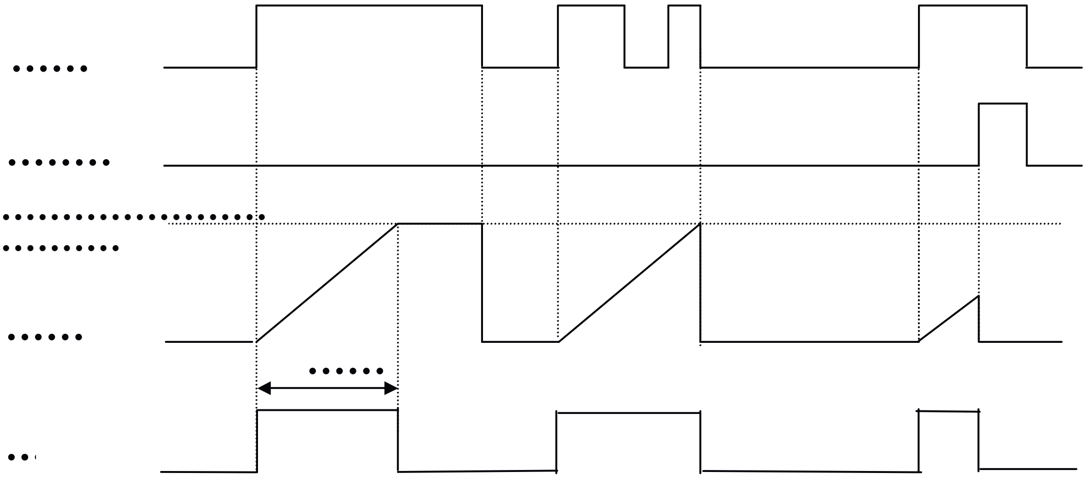
•••••••	••••••••••••••••••••••••••••••••••••						
••••••••••••	•••••	•••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time RST: reset **Output** Q: timer output ET: elapsed time

■ Function

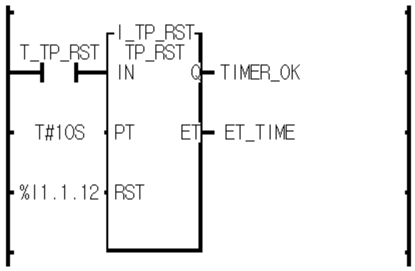
- • If IN is 1, Q is 1. And if elapsed time reaches preset time, timer output Q is 0.
- • ET increases its value from when IN is 1, keeps its value at PT and is cleared when IN is 0.
- • It doesn't matter whether IN changes its state or not while timer output Q is 1 (during a pulse output).
- • If RST is 1, output Q and ET are 0.

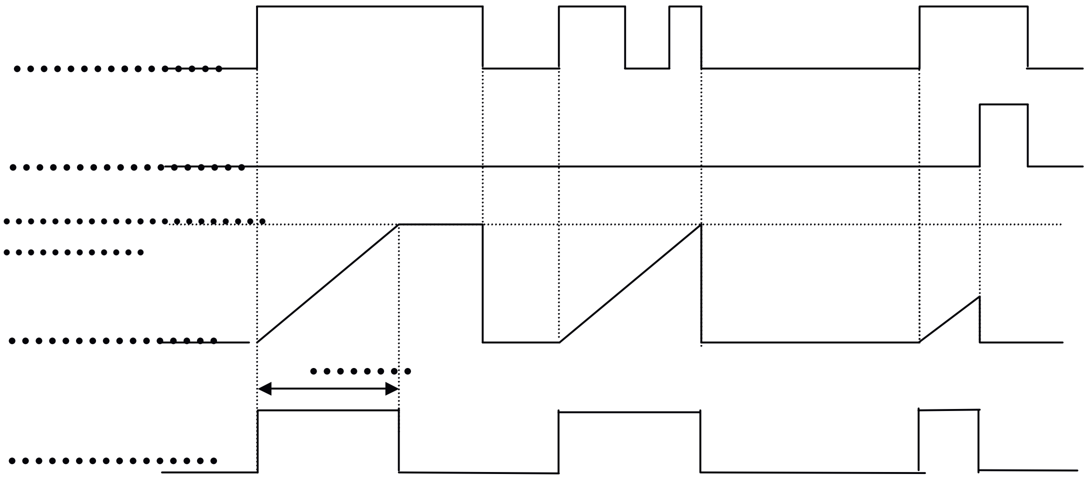
■ Timing Chart



■ **Program Example**

LD



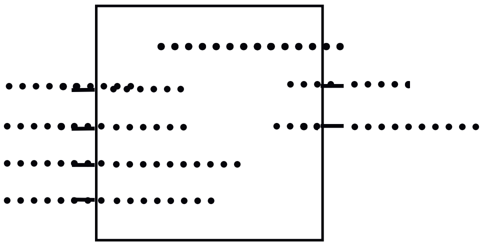


(1) If input variable T_TP_RST is 1, output variable TIMER_OK is 1. And 10 seconds later, TIMER_OK is 0.
Once TP_RST timer is executed, input T_TP_RST doesn't matter.
(2) ET_TIME value increases and stops at 10S. And if T_TP_RST is 0, it is 0.
(3) If input contact %I1.1.12 is 1, TIIMER_OK and ET_TIME are all cleared.

TP_UINT

TP with integer setting

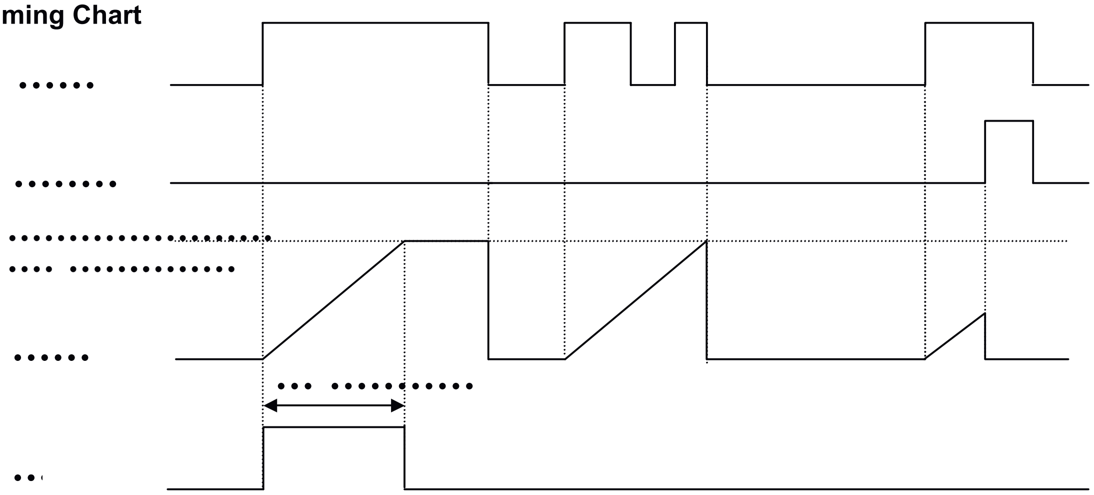
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time UNIT: time unit of setting time RST: reset **Output** Q: timer output ET: elapsed time

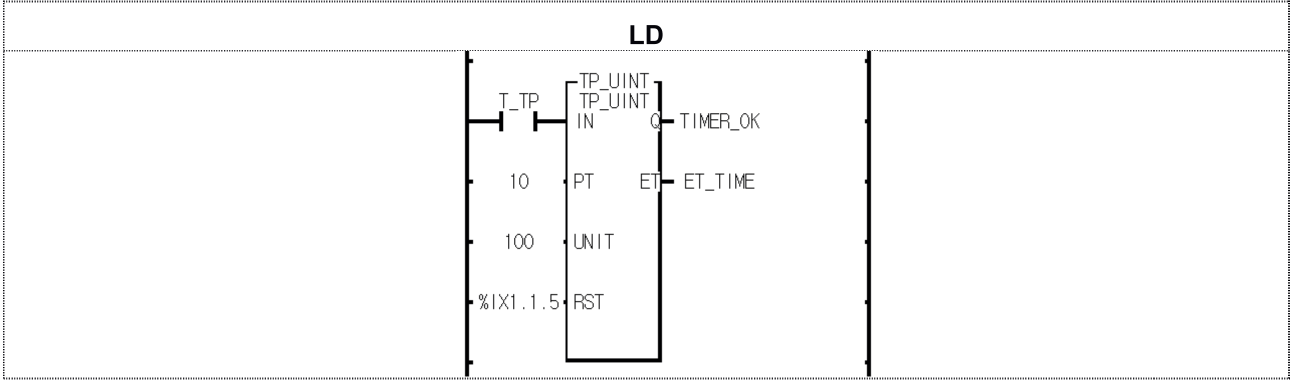
■ Function

- • If IN is 1, Q is 1. And if elapsed time reaches preset time, timer output Q is 0.
- • ET increases its value from when IN is 1, keeps its value at PT and is cleared when IN is 0.
- • It doesn't matter whether IN changes its state or not while timer output Q is 1 (during a pulse output).
- • If RST is 1, output Q and ET are 0.
- • Preset time is PT x UNIT[ms].

■ Timing Chart



■ **Program Example**

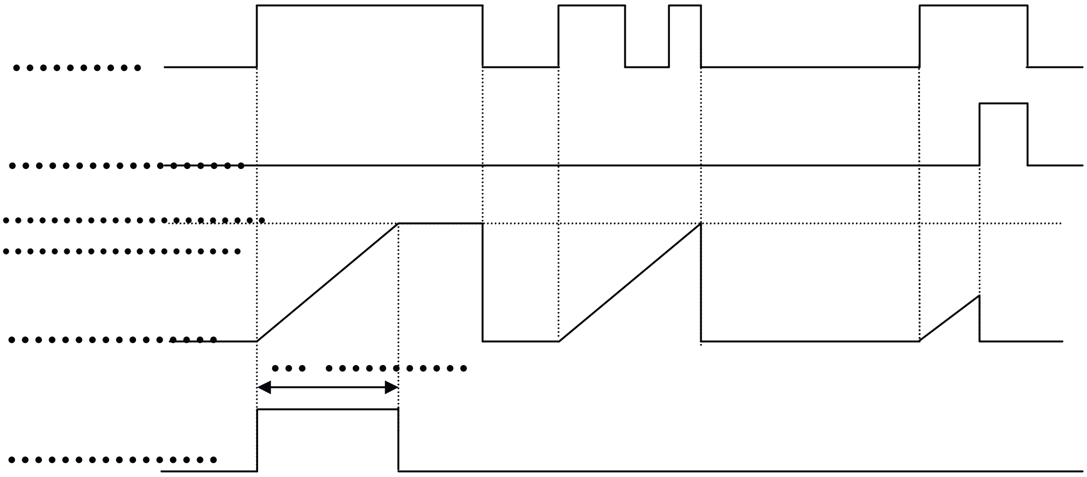


(1) Preset time is PT x UNIT[s] = 10 x 1000[s] = 10[s].

(2) If input variable T_TP is 1, output variable TIMER_OK is 1. And 10 seconds later, TIMER_OK is 0. Once TP_UINT timer is executed, input T_TP doesn't matter.

(3) ET_TIME value increases and stops at 10000. And if T_TP is 0, it is 0.

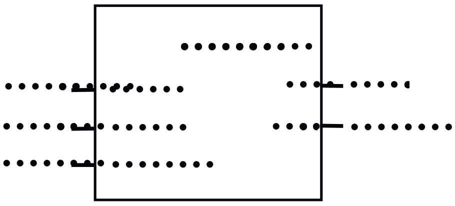
(4) If input contact %IX1.1.5 is 1, TIMER_OK and ET_TIME are all cleared.



TRTG

Retriggerable Timer

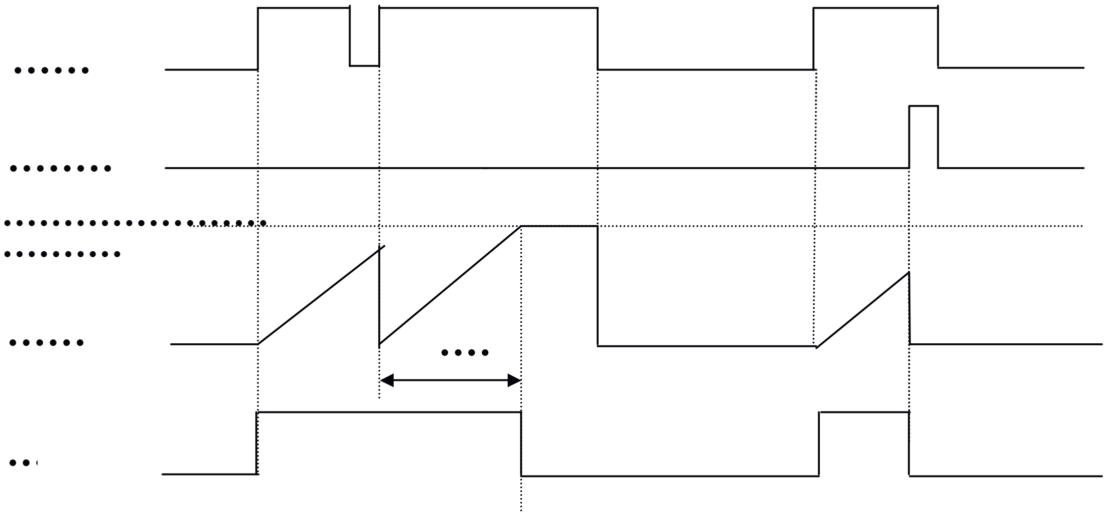
••••••••	•••						
••••••••••••	••••••	••••••	••	••	••	••	••

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time RST: reset **Output** Q: timer output ET: elapsed time

■ Function

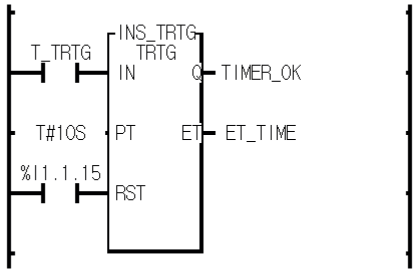
- • If IN is 1, Q is 1. And if elapsed time reaches preset time, timer output Q is 0.
- • If IN turns on again before elapsed time reaches preset time, then elapsed time is set as 0 and increased again. And if it reaches PT, Q is 0.
- • If RST is 1, timer output Q and elapsed time ET are 0.

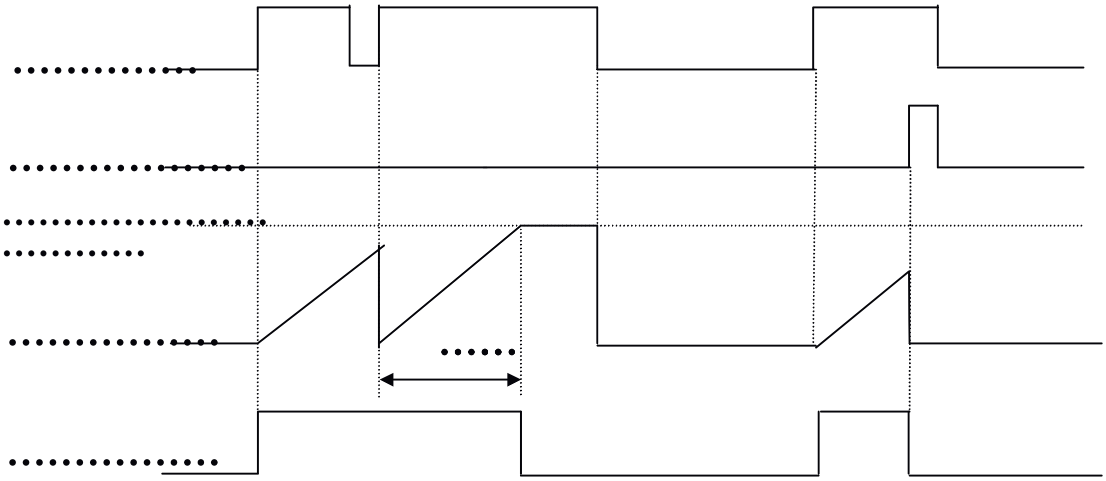
■ Timing Chart



■ **Program Example**

LD



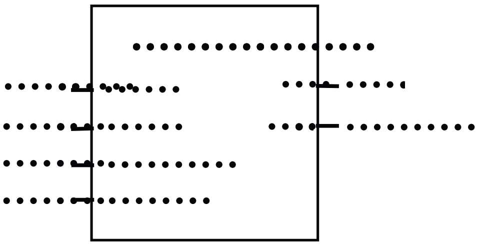


(1) TIMER_OK is 1 during 10 seconds after input variable T_TRTG becomes 1 from 0. If T_TRTG becomes 1 from 0 after timer is executed, ET_TIME is set as 0 and increased again.
(2) TIMER_OK is 1 during 10 seconds even when T_TRTG becomes 0 from 1.
(3) ET_TIME value increases and stops at T#10S. And it is 0 when T_TRTG is 0.
(4) If input contact %I1.1.15 is 1, TIMER_OK and ET_TIME are all cleared.

TRTG_UINT

TRTG with integer setting

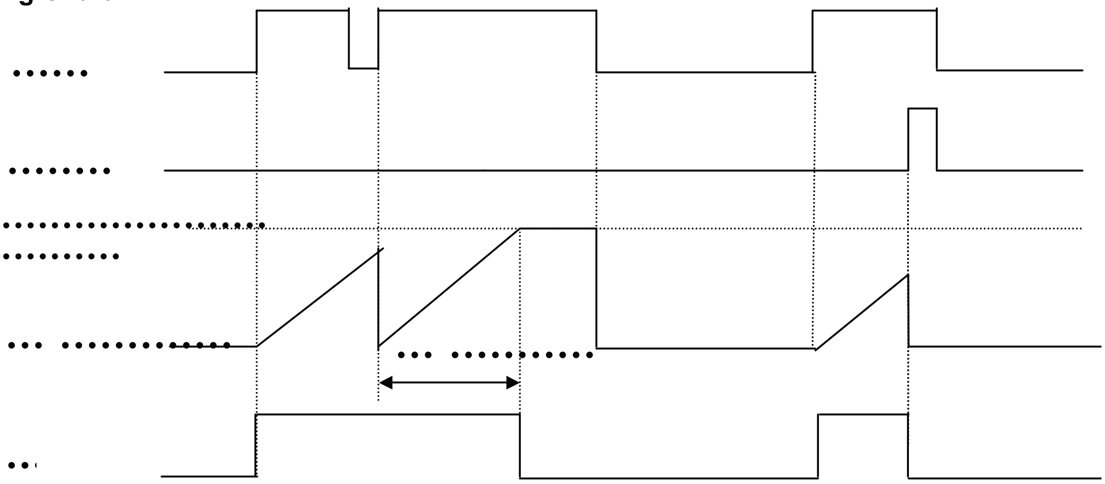
•••••••	••••	••••	••••	••••	••••	••••	••••••
••••••••••••	•••••	••••	• •	• •	• •	• •	• •

Function Block	Description
	**Input** IN: operation condition for Timer PT: preset time UNIT: time unit of setting time RST: reset **Output** Q: timer output ET: elapsed time

■ Function

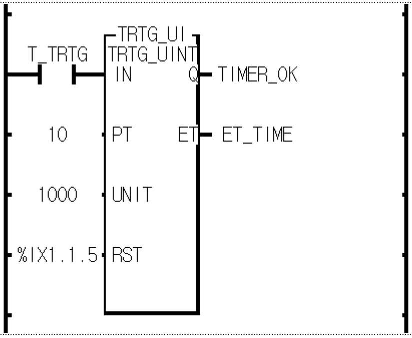
- • If IN is 1, Q is 1. And if elapsed time reaches preset time, timer output Q is 0.
- • If IN turns on again before elapsed time reaches preset time, then elapsed time is set as 0 and increased again. And if it reaches PT, Q is 0.
- • If RST is 1, timer output Q and elapsed time ET are 0.
- • Preset time is PT x UNIT[ms].

■ Timing Chart



■ **Program Example**

LD



(1) Preset time is PT x UNIT[ms] = 10 x 1000[ms] = 10[s].
(2) TIMER_OK is 1 during 10 seconds after input variable T_TRTG becomes 1 from 0. If T_TRTG becomes 1 from 0 after timer is executed, ET_TIME is set as 0 and increased again.
(3) TIMER_OK is 1 during 10 seconds even when T_TRTG becomes 0 from 1.
(4) ET_TIME value increases and stops at 10000. And it is 0 when T_TRTG is 0.
(5) If input contact %IX1.1.5 is 1, TIMER_OK and ET_TIME are all cleared.