DST grammar

DiBA PLC users can create text files with DST extension to manipulate what E5A will do. The DST file can be modified using various general editors such as Notepad. In the DST file, case sensitive is valid only for string content. Both CONFIGURATION and conFiGuratIon are recognized as CONFIGURATION structure indicators. The DST example uses case-sensitive characters only to make it easier for users to read.

```
CONFIGURATION nameOfConf
END_CONFIGURATION
```

[DST 1] conf0.dst

[DST 1] is the smallest example of a valid DST. CONFIGURATION must be defined and only one can exist. The name of the CONFIGURATION is not used elsewhere, but must be specified. The end is marked with END_CONFIGURATION.

1. Indicators of DST

We will list the predefined indicators in the DST. Indicators cannot be used for other purposes. When you create new indicators, such as variable names or function names, you should avoid predefined ones. Detailed descriptions of the indicators are divided into categories. The following table sorts the names of the indicators in alphabetical order, and shows the position of the body with a description for each indicator.

Note: Comments begin with "(*" and end with "*)". You cannot put comments inside comments and text strings, but you can put comments anywhere else in the DST.

Indicators	Classification	Position	Indicators	Classification	Position
+	Work	1.4.3.	-	Work	1.4.3.
*	Work	1.4.3.	/	Work	1.4.3.
&	Work	1.4.2.	|	Work	1.4.2.
!	Work	1.4.2.	^	Work	1.4.2.
ABS	Work	1.4.3.	ACOS	Work	1.4.3.
ADDROF	Work	1.4.5.	AND	Work	1.4.2.
ARRAY	Variable	1.3.	ASIN	Work	1.4.3.
AT	Variable	1.3	ATAN	Work	1.4.3.
BACKUP	Work	1.4.7.	BOOL	Type	1.1.
BY	Work	1.4.6.	BYTE, USINT	Type	1.1.
CASE	Work	1.4.6.	CHG_ENDIAN	Work	1.4.5.
CONCAT	Work	1.4.4.	CONFIGURATION	Structure	1.2.

Indicators	Classification	Position	Indicators	Classification	Position
COS	Work	1.4.3.	CPU	Structure	1.2
CTD	FB type	1.5.1.	CTU	FB type	1.5.1.
CTUD	FB type	1.5.1.	DATE	Type	1.1.
DATE_AND_TIME, DT	Type	1.1.	DELETE	Work	1.4.4.
DINT	Type	1.1.	DO	Work	1.4.6.
DWORD, UDINT	Type	1.1.	EDC_CRC_16ARC	Work	1.4.5.
EDC_CRC_16DDS110	Work	1.4.5.	EDC_CRC_16DECTR	Work	1.4.5.
EDC_CRC_16DNP	Work	1.4.5.	EDC_CRC_16EN13757	Work	1.4.5.
EDC_CRC_16MODBUS	Work	1.4.5.	EDC_CRC_16UMTS	Work	1.4.5.
EDC_CRC_16USB	Work	1.4.5.	EDC_CRC_32ISOHDLC	Work	1.4.5.
EDC_CRC_8DARC	Work	1.4.5.	EDC_CRC_8I4321	Work	1.4.5.
EDC_CRC_8ICODE	Work	1.4.5.	EDC_CRC_8MAXIMDOW	Work	1.4.5.
EDC_CRC_8ROHC	Work	1.4.5.	EDC_CRC_8SMBUS	Work	1.4.5.
EDC_CRC_8WCDMA	Work	1.4.5.	EDC_SUM_BYTE	Work	1.4.5.
EDC_SUM_DWORD	Work	1.4.5.	EDC_SUM_WORD	Work	1.4.5.
EDC_XOR_BYTE	Work	1.4.5.	EDC_XOR_DWORD	Work	1.4.5.
EDC_XOR_WORD	Work	1.4.5.	ELSE	Work	1.4.6.
ELSIF	Work	1.4.6.	END_CASE	Work	1.4.6.
END_CONFIGURATION	Structure	1.2	END_FOR	Work	1.4.6.
END_FUNCTION	Structure	1.2.	END_FUNCTION_BLOCK	Structure	1.2.
END_IF	Work	1.4.6.	END_PROGRAM	Structure	1.2.
END_REPEAT	Work	1.4.6.	END_RESOURCE	Structure	1.2.
END_STRUCT	Type	1.1.	END_TYPE	Type	1.1.
END_VAR	Variable	1.3.	END_WHILE	Work	1.4.6.
EQ, =	Work	1.4.2.	ETH_1	Structure	1.2.
ETH_2	Structure	1.2.	EXIT	Work	1.4.6.
EXP	Work	1.4.3.	EXPT, **	Work	1.4.3.
F_TRIG	FB type	1.5.1.	FALSE	Value	1.1.
FIND	Work	1.4.4.	FOR	Work	1.4.6.
FUNCTION	Structure	1.2.	FUNCTION_BLOCK	Structure	1.2.
GE, >=	Work	1.4.2.	GET_BOOL	Work	1.4.5.
GET_BYTE	Work	1.4.5.	GET_DATE	Work	1.4.5.
GET_DINT	Work	1.4.5.	GET_DT	Work	1.4.5.

Indicators	Classification	Position	Indicators	Classification	Position
GET_DWORD	Work	1.4.5.	GET_INT	Work	1.4.5.
GET_REAL	Work	1.4.5.	GET_SINT	Work	1.4.5.
GET_TIME	Work	1.4.5.	GET_TOD	Work	1.4.5.
GET_WORD	Work	1.4.5.	GT, >	Work	1.42.
IF	Work	1.4.6.	INSERT	Work	1.4.4.
INT	Type	1.1.	LE, <=	Work	1.42.
LEFT	Work	1.4.4.	LEN	Work	1.4.4.
LIMIT	Work	1.4.3.	LN	Work	1.4.3.
LOG	Work	1.4.3.	LT, <	Work	2.2
MAX	Work	1.4.3.	MID	Work	1.4.4.
MIN	Work	1.4.3.	MODULO	Work	1.4.3.
MUX	Work	1.4.6.	NE, <>	Work	1.4.2.
NOT	Work	1.4.2.	OF	Variable, Work	1.3, 1.4.6.
ON	Structure	1.2	OR	Work	1.4.2.
PID	FB type	1.5.2.	PROGRAM	Structure	1.2.
R_TRIG	FB type	1.5.1.	REAL	Type	1.1.
REPEAT	Work	1.4.6.	REPLACE	Work	1.4.4.
RESOURCE	Structure	1.2.	RETURN	Work	1.4.6.
RIGHT	Work	1.4.4.	ROL	Work	1.4.3.
ROR	Work	1.4.3.	RS	FB type	1.5.1.
RTC	FB type	1.5.1.	SEE_NW_DOMAIN	Work	1.4.7.
SEE_NW_IP	Work	1.4.7.	SEE_NW_MODE	Work	1.4.7.
SEE_NW_SSID	Work	1.4.7.	SEL	Work	1.4.6.
SEMA	FB type	1.5.1.	SER_1	Structure	1.2.
SER_2	Structure	1.2.	SER_3	Structure	1.2.
SER_4	Structure	1.2.	SER_5	Structure	1.2.
SER_6	Structure	1.2.	SET_BOOL	Work	1.4.5.
SET_BYTE	Work	1.4.5.	SET_DATE	Work	1.4.5.
SET_DINT	Work	1.4.5.	SET_DT	Work	1.4.5.
SET_DWORD	Work	1.4.5.	SET_INT	Work	1.4.5.
SET_REAL	Work	1.4.5.	SET_SINT	Work	1.4.5.
SET_TIME	Work	1.4.5.	SET_TOD	Work	1.4.5.
SET_WORD	Work	1.4.5.	SHL	Work	1.4.3.

Indicators	Classification	Position	Indicators	Classification	Position
SHR	Work	1.4.3.	SIN	Work	1.4.3.
SINT	Type	1.1.	SIZEOF	Work	1.4.5.
SQRT	Work	1.4.3.	SR	FB type	1.5.1.
STR_FROM_BOOL	Work	1.4.4.	STR_FROM_BYTE	Work	1.4.4.
STR_FROM_DATE	Work	1.4.4.	STR_FROM_DINT	Work	1.4.4.
STR_FROM_DT	Work	1.4.4.	STR_FROM_DWORD	Work	1.4.4.
STR_FROM_INT	Work	1.4.4.	STR_FROM_REAL	Work	1.4.4.
STR_FROM_SINT	Work	1.4.4.	STR_FROM_TIME	Work	1.4.4.
STR_FROM_TOD	Work	1.4.4.	STR_FROM_WORD	Work	1.4.4.
STR_TO_BOOL	Work	1.4.4.	STR_TO_BYTE	Work	1.4.4.
STR_TO_DATE	Work	2.4	STR_TO_DINT	Work	1.4.4.
STR_TO_DT	Work	2.4	STR_TO_DWORD	Work	1.4.4.
STR_TO_INT	Work	2.4	STR_TO_REAL	Work	1.4.4.
STR_TO_SINT	Work	1.4.4.	STR_TO_TIME	Work	1.4.4.
STR_TO_TOD	Work	1.4.4.	STR_TO_WORD	Work	2.4
STRING	Type	1.1.	STRUCT	Type	1.1.
TAN	Work	1.4.3.	TASK	Structure	1.2.
THEN	Work	1.4.6.	TIME	Type	1.1.
TIME_OF_DAY, TOD	Type	1.1.	TO	Work	1.4.6.
TOF	FB type	1.5.1.	TON	FB type	1.5.1.
TP	FB type	1.5.1.	TRUE	Value	1.1.
TYPE	Type	1.1.	UINT, WORD	Type	1.1
UNTIL	Work	1.4.6.	VAR	Variable	1.3.
VAR_GLOBAL	Variable	1.3.	VAR_INPUT	Variable	1.3.
WA_ABLE_GEN	Work	1.4.7.	WA_ABLE_OUT	Work	1.4.7.
WA_DISABLE_GEN	Work	1.4.7.	WA_DISABLE_OUT	Work	1.4.7.
WA_ENABLE_GEN	Work	1.4.7.	WA_ENABLE_OUT	Work	1.4.7.
WHILE	Work	1.4.6.	XOR	Work	1.4.2.

The indicators in the following table are defined in the standard (IEC 61131-3) but not used in the DST, which restrict users from defining them in any other sense.

Indicators	Classify	Indicators	Classify	Indicators	Classifiy	Indicators	Classify
LINT	Type	LREAL	Type	LWORD	Type	ULINT	Type
VAR_ACCESS	Variable	VAR_EXTERNAL	Variable	VAR_IN_OUT	Variable	VAR_OUTPUT	Variable

You should input DST files in row units. Most content must use a row delimiter, except for some structural and variable indicators. The following are indicators that are used in the same sense throughout the DST file.

;	A row delimiter Show end of action row
,	Item delimiter Displays between items and items when multiple items of the same level are used within a task row
:=	Copy Value Place the expression with the result of the operation on the right side of the indicator, and place the variable with the result value on the left side

1.1. Type/Value Indicators

The following is a summary of the type/value indicators used by DST. If the type indicator and the value indicator are different, it shows additional value indicators.

Indicators	Explanation
FALSE	Specify a value of 0
TRUE	Specify a value of 1
BOOL	Specify variable type as BOOL
BYTE, USINT	Specify variable type as BYTE
WORD, UINT	Specify variable type as WORD
DWORD, UDINT	Specify variable type as DWORD
SINT	Specify variable type as SINT
INT	Specify variable type as INT
DINT	Specify variable type as DINT
REAL	Specify variable type as REAL
DATE (Value indicator: D)	Specify variable type as DATE, Specify the type of value as DATE (An example of a value) d#2019-1-2
DT, DATE_AND_TIME (Value indicator: DT)	Specify variable type as DT, Specify the type of value as DT (An example of a value 1) dt#2019-1-2-3:4:5.678 (An example of a value 2) dt#2019-1-2-3:4:5
TIME (Value indicator: T)	Specify variable type as TIME, Specify the type of value as TIME (An example of a value 1) t#1d2h3m4s567ms (An example of a value 2) t#800ms (An example of a value 3) t#9h

Indicators	Explanation
TOD, TIME_OF_DAY (Value indicator: TOD)	Specify variable type as TOD, Specify the type of value as TOD (An example of a value 1) tod#1:2:3.456 (An example of a value 2) tod#7:8:9
STRING	Specify variable type as STRING (An example of a value) ‘AbCd? EF ! ^&*@lr’ (Special Characters) ’ = ‘$’’, $ = ‘$$’, Line Feed = ‘$L’ or ‘$l’, Carriage Return = ‘$R’ or ‘$r’
TYPE	Initiate user-type declaration
END_TYPE	End user-type declaration
STRUCT	Initiate structure declaration
END_STRUCT	End of structure declaration

The values of the type representing numbers from BOOL to REAL can be expressed as follows: You can insert '_' to make it easier for the user to read the digits of the number, etc. Number representations are not case sensitive.

Number representation	Expression example
Integer (recognize as decimal if no integer is specified)	0, 1, -2, 3_456(= 3456)
Real number	0.0, 1.2, -3.456_789, 1.2E3, 4.5E-6, -7.8E-9
Binary number	2#1111_0000(= 240), 2#10(= 2)
Octal number	8#73(= 59), 8#10_00(= 512)
Decimal number	10#12_345(= 12345)
Hexadecimal number	16#FEDC(= 65244), 16#89A(= 2202)

There are 13 basic variable types supported by E5A.

Variable type	Byte size	Bit size	Minimum	Maximum
BOOL	-	1	0	1
BYTE USINT	1	8	0	255
SINT	1	8	-128	127
WORD UINT	2	16	0	65,535
INT	2	16	-32,768	32,767
DWORD UDINT	4	32	0	4,294,967,295
DINT	4	32	-2,147,483,648	2,147,483,647
REAL	4	32	-3.4e38	3.4e38
TIME	8	64	-	-

Variable type	Byte size	Bit size	Minimum	Maximum
TOD	8	64	tod#0:0:0.0	tod#23:59:59.999
DATE	8	64	d#1900-1-1	-
DT	8	64	dt#1900-1-1-0:0:0.0	-
STRING	64(Can be specified)	-	-	-

We will explain how the user declares a new type. TYPE can only be used within CONFIGURATION, and STRUCT can only be used within TYPE. Variables in TYPE are overlapped at the same address. With STRUCT, variables are placed at consecutive addresses so that they do not overlap each other. By combining TYPE and STRUCT, a variable can be placed where the user wants it. The name of the new type specified while declaring TYPE can be used as the variable type when declaring a variable later.

User type declaration form	TYPE {name of new type}:
	{variable declaration};
	{variable declaration};
	{struct declaration}
	{struct declaration}
	END_TYPE
	{name of new type} is user-specified. You must use a unique name throughout the DST file.

ENUM declaration form	TYPE {name of new type}: ({name of ENUM}, {name of ENUM},); END_TYPE When declared as ENUM, the new type has the same size and scope as DINT. {ENUM name} is specified by the user. You must use a unique name throughout the DST file. If no value is specified, the first value is 0, and then the value is added by 1. You can specify a value in the form of {name of ENUM} := {assigned value}, Subsequent ENUMs have a value added by 1.

struct declaration form	STRUCT
	{variable declaration};
	{variable declaration};
	END_STRUCT

Example of declaring a type that combines a length of 1 byte and a buffer of 10 bytes	TYPE T_UDF:
	  STRUCT
	    length : BYTE;
	    buffer : ARRAY [10] OF BYTE;
	  END_STRUCT
	END_TYPE

An example of declaring a type that bundles IP V4 addresses in various forms	TYPE T_IPv4:
	  ip_value : UDINT;
	  ip_byte : ARRAY [4] OF BYTE;
	  ip_class_a : USINT;
	  ip_class_b : UINT;
	END_TYPE

Example using T_IPv4	PROGRAM aTestPgm
	  VAR
	    vIpAddress : T_IPv4;
	    vClassA : USINT;
	  END_VAR
	  vIpAddress.ip_value := 16#f11f001a;
	  vClassA := vIpAddress.ip_class_a;
	END_PROGRAM

Users can create new variable types by combining variables and work contents. As a characteristic of PLC of E5A, it is necessary to overcome limitations such as step-by-step operation progress and separation of operation and execution details while remembering settings. The utility can be seen in the example of handling tasks easily and simply using the built-in FB (Function Block) made for the operation of communication functions.

FB variables are created by designating the FB constructed using the structure indicator FUNCTION_BLOCK as the type of the global variable.

1.2. Structure Indicators

DST file is built by putting the structures of CONFIGURATION, PROGRAM, FUNCTION_BLOCK, and FUNCTION on the basic text file. One CONFIGURATION structure must exist in the entire DST file. Each of the PROGRAM structure, FUNCTION_BLOCK structure, and FUNCTION structure is constructed according to the needs, so it may not exist or may exist in plurality. Here is an example of a structure format that contains all the structure elements of a DST file one by one.

```
CONFIGURATION {name of CONF}
  RESOURCE {name of resource area} ON {name of resource}
    TASK {name of task} (SINGLE := TRUE, PRIORITY := 1);
    PROGRAM {nickname for the program} WITH {name of task} : {name of PROG or name of FB variable} ();
  END_RESOURCE
END_CONFIGURATION

PROGRAM {name of PROG}
END_PROGRAM

FUNCTION_BLOCK {name of FB}
END_FUNCTION_BLOCK

FUNCTION {name of FUN} : {output variable type}
END_FUNCTION
```

In the CONFIGURATION structure, user type definition, global variable declaration, and RESOURCE assignment are possible. A work object to be executed is defined through resource assignment. Individual work objects are independent of each other and occupy E5A exclusively during execution. The following is a summary of what is needed to build the CONFIGURATION structure.

{name of CONF} is user-specified. It is not used anywhere, but must be entered.
  CONFIGURATION must be built once within the DST file.

{name of the resource area} is user-specified. It is later used to delimit a resource in the work area. You must use a unique name throughout the DST file.

{name of resource} takes one of the following structure indicators to select a resource. A resource cannot be assigned to multiple RESOURCEs.

- CPU = Specifies E5A.
- ETH_1 = Specifies the server port.
- ETH_2 = Specifies the client port.
- SER_1 = Designates RS485 port.
- SER_2 = Nothing specified. This is a spare.
- SER_3 = Nothing specified. This is a spare.
- SER_4 = Nothing specified. This is a spare.
- SER_5 = Nothing specified. This is a spare.
- SER_6 = Nothing specified. This is a spare.

Multiple TASKs can be declared within RESOURCE. Traits are one-time and periodic. Specify SINGLE as TRUE to declare one-time, or INTERVAL to declare periodic. SINGLE is BOOL type, and INTERVAL is TIME type.

When multiple TASKs are declared, the execution condition may be satisfied at the same time. PRIORITY is the value that determines the execution priority in this case. The settable range is 1 to 9.

{name of task} is user-specified. It is only valid within that RESOURCE.

PROGRAM builds a work object by binding the execution condition and the execution priority created with TASK with {name of PROG or name of FB variable}.

{nickname for program} is user-specified. It is not used anywhere and does not need to be specified.

{name of PROG or name of FB variable} specifies {name of PROG} of PROGRAM structure or {name of FB variable} with FB type assigned to global variable.

The following parentheses only mean that PROG or FB is called, not for inputting parameters. Therefore, the user should set the parameter given to the FB called in the PROGRAM line in advance in another operation.

In the PROGRAM structure, FUNCTION_BLOCK structure, and FUNCTION structure, you can declare local variables and write work details. The following is a summary of what is needed to build each structure.

In the PROGRAM structure, {name of PROG} is user-specified. You must use a unique name throughout the DST file.

PROGRAM has no parameters (input variables) or output variables. The built task is defined as PROG.

The calling format of PROG is {name of PROG}();.

In the FUNCTION_BLOCK structure, {name of FB} is user-specified. You must use a unique name throughout the DST file.

A constructed task is defined as FB, and created type is defined as FB types. FB type can only be declared as a global variable, and a variable created in this way is defined as an FB variable. All local (internal) variables of FB variables can be used as parameters (input variables) or output variables.

The calling format of FB is {name of FB variable}({name of FB local variable} := {parameter value});. You can enter no parameters, or you can enter multiple parameters. Also, before calling FB, you can separate the line of work and enter {name of FB variable}.{name of FB local variable} := {parameter value};.

In the FUNCTION structure, {name of FUN} is specified by the user. You must use a unique name throughout the DST file.

For {output variable type}, you must select one of the default variable types. The name of the output variable is the same as {name of FUN}.

A constructed task is defined as a FUN. FUN has parameters (input variables), local (internal) variables, and output variables. Input variables and internal variables may not be declared, but one output variable must be declared.

The calling format of FUN is {variable name} := {name of FUN}({parameter value});. Unlike PROG or FB, the return value must be passed to a variable accessible by the task. The parameters must match the number and order of the input variables declared in the FUNCTION structure.

Constructed works form a hierarchical relationship. At the top level, there is a work object, followed by PROG, FB, and FUN. You can call tasks of siblings and children of the task you want to call. However, the invocation of a sibling is prohibited from calling itself (recursive invocation).

1.3. Variable Indicators

The types of variables that can be defined in the DST file are global variables, local variables, and input variables. The type of variable is determined by the area in which the variable is declared. The variable area is explained first, followed by the variable declaration.

Format of global variable area	VAR_GLOBAL
	{variable declaration};
	END_VAR

Global variable area description	The global variable area can be declared only in the CONFIGURATION structure and RESOURCE structure. If declared within the CONFIGURATION structure, it can be referenced as {variable name} in all work areas of the DST file. If declared within the RESOURCE structure, it can be referenced as {name of resource area}.{name of variable} in all work areas of the DST file.

Format of local variable area	VAR
	{variable declaration};
	END_VAR

Local variable area description	Local variable area can be declared only in the PROGRAM structure, FUNCTION_BLOCK structure, and FUNCTION structure. In general, a local variable can be referenced as {name of variable} only in the operation within the structure in which the local variable is declared. Exceptionally, local variables of FB variables can be referenced as {name of FB variable}.{name of variable} in all work areas of the DST file. If the FB variable is declared in the RESOURCE structure, it must be referenced as {name of resource area}.{name of FB variable}.{name of variable}.

Format of input variable area	VAR_INPUT
	{variable declaration};
	END_VAR

Input variable area description	The input variable area can be declared only in the FUNCTION structure. An input variable can be referenced as {name of variable} only in the operation within the structure in which the input variable is declared. When calling FUN from another PROG or FB, all input variables must be entered as parameters in the order in which they were declared.

{variable declaration} can be declared on zero or more lines. The basic format is {name of variable} : {variable type}; and has various functions for convenience. Restrictions on each function may exist in addition to those described below in order to reliably use the limited resources of E5A.

Variable declaration format	{name of variable} : {variable type} ; {name of variable} is specified by the user. You must use a name that is unique within the reference scope of the variable. For {variable type}, one of the basic variable type, user type, and FB type can be used. However, FB type is available only when the variable declaration is included in the global variable area. If {variable type} is STRING, you can also specify the maximum length as STRING(10). If the length is not specified, 64 bytes are specified as the maximum length by default. (usage example) vInt1 : INT; (usage example) vStr1 : STRING; (usage example) vStr2 : STRING(30);
Multiple variable declaration function	{name of variable 1}, {name of variable 2}, {name of variable 3} : {variable type} ; Multiple variable names can be declared with the same variable type. (usage example) vInt1, vInt2 : INT;

Array declaration function	{variable type} : ARRAY[{array size}] OF {variable type} ;
	This is a type with an array declaration added in type 1. It is available for all variable types except STRING. Arrays can be declared from 1st to 10th. Use "," to separate degrees.
	(usage example) vInt1 : ARRAY[6] OF INT;
	(usage example) vInt2 : ARRAY[2,3,4] OF INT;

Initial value designation function	{name of variable} : {variable type} := {initial value} ; You can specify initial values for basic variable types. The initial value cannot be specified for user type and FB type. (usage example) vInt1 : INT := 0;
Scoping function	{name of variable} : {variable type} ({minimum value} .. {maximum value}) ; You can specify a range for basic variable types except STRING. (usage example) vInt1 : INT(-2000..8000);
Addressing function	{name of variable} AT {variable address} : {variable type} ; Any variable can be addressed in the global variable area. The memory area available as {variable address} can be input, output, or internal. The size of the address must have the same size as the variable type to be specified. However, STRING, user type, and FB type use byte address to designate the address. (usage example) gInt1 AT %IW16 : INT; (usage example) gStr1 AT %MB1024 : STRING;

1.4. Work Indicators

In the PROGRAM, FUNCTION_BLOCK, and FUNCTION structures, except for the variable areas, the rest are work areas.

1.4.1. Call

The work area is divided into rows, and you can perform operations by expressions or call PROG, FB, FUN, etc.

{name of PROG}();	It is called with the name of PROG. Parameters cannot be specified. The call operation alone constitutes one row.
{name of FB variable}({name of FB local variable} := {value});	It is called with the FB variable name. Parameters can be entered in the form of designating values for local variables declared within FB. There are no special restrictions on the number or order of parameters. If there are multiple parameters, separate them with ",". The call operation alone constitutes one row.
{name of FUN}({parameter value})	It is called with the name of FUN. The parameters must match the order and number of input variables declared within the FUN. If there are multiple parameters, separate them with ",".

1.4.2. Logical Operation

The following describes the work indicators for logical operations that can be used in expressions.

&	1 Bit AND. Gets the result of AND with BOOL operation. (usage example) TRUE & FALSE, result: FALSE
|	1 Bit OR. Gets the result of OR with BOOL operation. (usage example) TRUE | FALSE, result: TRUE
!	1 Bit NOT. Gets the result of performing NOT with BOOL operation. (usage example) ! TRUE, result: FALSE
^	1 Bit XOR. Returns the result of XORing with a BOOL operation. (usage example) TRUE ^ FALSE, result: TRUE
AND	32 Bit AND. Gets the result of AND with DWORD operation. (usage example) 2#1010 AND 2#1100, result: 16#8

OR	32 Bit OR. Gets the result of OR with DWORD operation. (usage example) 2#1010 OR 2#1100, result: 16#E
NOT	32 Bit NOT. Gets the result of NOT with DWORD operation. (usage example) NOT 2#1010, result: 16#FFFF_FFF5
XOR	32 Bit XOR. Gets the result of XOR with DWORD operation. (usage example) 2#1010 XOR 2#1100, result: 2#0110
>=, GE	Greater than or equal to. Returns TRUE if the value on the left side of the indicator is greater than or equal to the value on the right side, and FALSE otherwise. (usage example) 1 >= 2, result: FALSE
>, GT	Greater than. Returns TRUE if the value on the left side of the indicator is greater than on the right side, and FALSE otherwise. (usage example) 1 GT 2, result: FALSE
<=, LE	Less than or equal to. Returns TRUE if the value on the left side of the indicator is less than or equal to the value on the right side, and FALSE otherwise. (usage example) 1 <= 2, result: TRUE
<, LT	Less than. Returns TRUE if the value on the left side of the indicator is less than on the right side, and FALSE otherwise. (usage example) 1 LT 2, result: TRUE
=, EQ	Equal to. Returns TRUE if the right and left values of the indicator are the same, and FALSE otherwise. (usage example) 1 = 2, result: FALSE
<>, NE	Not equal to. Returns TRUE if the values on the right and left sides of the indicator are different, and FALSE otherwise. (usage example) 1 NE 2, result: TRUE

1.4.3. Arithmetic Operation

The following describes the work indicators for arithmetic operations that can be used in expressions.

+	Addition. (usage example) 1 + 2, result: 3

-	Subtract. (usage example) 1 - 2, result: -1
*	Multiplication. (usage example) 1 * 2, result: 2
/	Division. (usage example) 1 / 2, result: 0.5
MODULO	Remainder. (usage example) 5 MODULO 4, result: 1
**, EXPT	Square. (usage example) 5 ** 2, result: 25
ABS({parameter 1})	Absolute value. (usage example) ABS(-5), result: 5
SQRT({parameter 1})	Square root. (usage example) SQRT(4), result: 2
EXP({parameter 1})	Power of e. (usage example) EXP(1), result(approximation): 2.71828
LN({parameter 1})	Natural logarithm. (usage example) LN(EXP(1)), result: 1
LOG({parameter 1})	Common logarithm. (usage example) LOG(10), result: 1
MAX({parameter 1}, ...)	Maximum. Two or more parameters must be entered in numeric type. There is no limit on the number. Returns the largest of the entered values. (usage example) MAX(2, 3, 4), result: 4 (usage example) MAX(2, 1, 4), result: 4
MIN({parameter 1}, ...)	Minimum. Two or more parameters must be entered in numeric type. There is no limit on the number. Returns the smallest of the entered values. (usage example) MIN(2, 3, 4), result: 2 (usage example) MIN(2, 1, 4), result: 1
LIMIT({parameter 1}, {parameter 2}, {parameter 3})	Value limit. {parameter 1} is the minimum value, {parameter 2} is the input value, and {parameter 3} is the maximum value. Constrains the input values between the minimum and maximum values and returns them as the resulting values. (usage example) LIMIT(2, 3, 4), result: 3 (usage example) LIMIT(2, 1, 4), result: 2

ROL({parameter 1}, {parameter 2})	Rotate Left. Rotates the input value to the left by a specified number with DWORD operation. {parameter 1} is the input value, {parameter 2} is the number of shift bits. (usage example) ROL(1, 2), result: 4
ROR({parameter 1}, {parameter 2})	Rotate Right. Rotates the input value to the right by a specified number with DWORD operation. {parameter 1} is the input value, {parameter 2} is the number of shift bits. (usage example) ROR(1, 2), result: 16#4000_0000
SHL({parameter 1}, {parameter 2})	Shift Left. Moves the input value to the left by the specified number with DWORD operation. {parameter 1} is the input value, {parameter 2} is the number of shift bits. (usage example) SHL(1, 2), result: 4
SHR({parameter 1}, {parameter 2})	Shift Right. Moves the input value to the right by the specified number with DWORD operation. {parameter 1} is the input value, {parameter 2} is the number of shift bits. (usage example) SHR(1, 2), result: 0
ACOS({parameter 1})	Arc-cosine. {parameter 1} must be entered between –1 and 1. The result is obtained in the range of 0 to 180 [°]. (usage example) ACOS(0.5), result: 60
ASIN({parameter 1})	Arc-sine. {parameter 1} must be entered between –1 and 1. The result is obtained in the range of -90 to 90 [°]. (usage example) ASIN(0.5), result: 30
ATAN({parameter 1})	Arc-tangent. The result is obtained in the range of –90 to 90 [°]. (usage example) ATAN(0.5), result(approximation): 26.565
COS({parameter 1})	Cosine. {parameter 1} must be entered in [°] units. The resulting value is obtained in the range of –1 to 1. (usage example) COS(50), result(approximation): 0.643
SIN({parameter 1})	Sine. {parameter 1} must be entered in [°] units. The resulting value is obtained in the range of –1 to 1. (usage example) SIN(50), result(approximation): 0.766

TAN({parameter 1})	Tangent. {parameter 1} must be entered in [°] units. (usage example) TAN(50), result(approximation): 1.192

1.4.4. String Operation

CONCAT({parameter 1}, ...)	Concatenate. Two or more parameters must be entered in a string type. There is no limit on the number. Returns a string concatenated in input order. (usage example) CONCAT('Ab', ' Cd'), result: 'Ab Cd'
DELETE({parameter 1}, {parameter 2}, {parameter 3})	Delete. Returns a string deleted by the specified length from the specified position in the input string. {parameter 1} is the input string, {parameter 2} is the cut position, and {parameter 3} is the length to cut. The position must be within the string. If the length is negative or the position+length is outside the string, it is deleted to the end of the string. (usage example) DELETE('Ab Cd', 2, -1), result: 'Ab'
FIND({parameter 1}, {parameter 2})	Find. Returns the first position in the input string that matches the search string. {parameter 1} is the input string, {parameter 2} is the search string. (usage example) FIND('Ab Bb ', 'b '), result: 1
INSERT({parameter 1}, {parameter 2}, {parameter 3})	Insert. Returns a string in which the insert string is inserted at the specified position in the input string. {parameter 1} is the input string, {parameter 2} is the insert string, and {parameter 3} is the position. (usage example) INSERT('Ab Cd', ' x', 2), result: 'Ab x Cd'
LEFT({parameter 1}, {parameter 2})	Take from the left. Returns a string taken as long as the specified length from the left of the input string. {parameter 1} is the input string, {parameter 2} is the length. If the length is negative or outside the string, it is truncated to the end of the string. (usage example) LEFT('Ab Cd', 2), result: 'Ab'
LEN({parameter 1})	Find the length [byte]. Returns the length of the input string. (usage example) LEN('Ab Cd'), result: 5

MID({parameter 1}, {parameter 2}, {parameter 3})	Take in the middle. Returns a string taken as long as the specified length from the specified position in the input string. {parameter 1} is the input string, {parameter 2} is the position, and {parameter 3} is the length. If length is negative or position+length is out of string, end of string is taken. (usage example) MID(‘Ab Cd’, 2, 2), result: ‘ C’
REPLACE({parameter 1}, {parameter 2}, {parameter 3}, {parameter 4})	Replace. Returns a string in which the specified length from the specified position of the input string is replaced with the insert string. {parameter 1} is the input string, {parameter 2} is the insert string, {parameter 3} is the position, and {parameter 4} is the length. If the length is negative or the position+length is outside the string, it is truncated to the end of the string. (usage example) REPLACE(‘Ab Cd’, ‘x’, 1, 3), result: ‘Axd’
RIGHT({parameter 1}, {parameter 2})	Take from the right. Returns a string taken as long as the specified length from the right of the input string. {parameter 1} is the input string, {parameter 2} is the length. If the length is negative or out of the string, it is taken up to the beginning of the string. (usage example) RIGHT(‘Ab Cd’, 2), result: ‘Cd’
STR_FROM_BOOL ({parameter 1})	Convert BOOL to string. Converts the input BOOL value to a string. (usage example) STR_FROM_BOOL(1), result: ‘TRUE’
STR_FROM_BYTE ({parameter 1})	Convert BYTE to string. Converts the input BYTE value to a string. (usage example) STR_FROM_BYTE(1), result: ‘1’
STR_FROM_SINT ({parameter 1})	Convert SINT to string. Converts the input SINT value to a string. (usage example) STR_FROM_SINT(1), result: ‘1’
STR_FROM_WORD ({parameter 1})	Convert WORD to string. Converts the input WORD value to a string. (usage example) STR_FROM_WORD(1), result: ‘1’
STR_FROM_INT ({parameter 1})	Convert INT to string. Converts the input INT value to a string. (usage example) STR_FROM_INT(1), result: ‘1’
STR_FROM_DWORD ({parameter 1})	Convert DWORD to string. Converts the input DWORD value to a string. (usage example) STR_FROM_DWORD(1), result: ‘1’

STR_FROM_DINT ({parameter 1})	Convert DINT to string. Converts the input DINT value to a string. (usage example) STR_FROM_DINT(1), result: ‘1’
STR_FROM_REAL ({parameter 1})	Convert REAL to string. Converts the input REAL value to a string. (usage example) STR_FROM_REAL(1), result: ‘1E000’
STR_FROM_TIME ({parameter 1})	Convert TIME to string. Converts the input TIME value to a string. (usage example) STR_FROM_TIME(t#1s), result: ‘T#0d0h0m1s0ms’
STR_FROM_TOD ({parameter 1})	Convert TOD to string. Converts the input TOD value to a string. (usage example) STR_FROM_TOD(tod#1:1:1), result: ‘TOD#1:1:1.000’
STR_FROM_DATE ({parameter 1})	Convert DATE to string. Converts the input DATE value to a string. (usage example) STR_FROM_DATE(d#2019-1-2), result: ‘D#2019-1-2’
STR_FROM_DT ({parameter 1})	Convert DT to string. Converts the input DT value to a string. (usage example) STR_FROM_DT(dt#2019-1-2-3:4:5), result: ‘DT#2019-1-2-3:4:5.000’
STR_TO_BOOL ({parameter 1})	Convert string to BOOL. Converts the input string to a BOOL value. (usage example) STR_TO_BOOL(‘TRUE’), result: 1
STR_TO_BYTE ({parameter 1})	Convert string to BYTE. Converts the input string to a BYTE value. (usage example) STR_TO_BYTE(‘1’), result: 1
STR_TO_SINT ({parameter 1})	Convert string to SINT. Converts the input string to a SINT value. (usage example) STR_TO_SINT(‘1’), result: 1
STR_TO_WORD ({parameter 1})	Converts string to WORD. Converts the input string to a WORD value. (usage example) STR_TO_WORD(‘1’), result: 1
STR_TO_INT ({parameter 1})	Convert string to INT. Converts the input string to an INT value. (usage example) STR_TO_INT(‘1’), result: 1
STR_TO_DWORD ({parameter 1})	Convert string to DWORD. Converts the input string to a DWORD value. (usage example) STR_TO_DWORD(‘1’), result: 1

STR_TO_DINT ({parameter 1})	Convert string to DINT. Converts the input string to a DINT value. (usage example) STR_TO_DINT('1'), result: 1
STR_TO_REAL ({parameter 1})	Convert string to REAL. Converts the input string to a REAL value. (usage example) STR_TO_REAL('1'), result: 1E000
STR_TO_TIME ({parameter 1})	Convert string to TIME. Converts the input string to a TIME value. (usage example) STR_TO_TIME('t#1s'), result: T#0d0h0m1s0ms
STR_TO_TOD ({parameter 1})	Convert string to TOD. Converts the input string to a TOD value. (usage example) STR_TO_TOD('tod#1:1:1'), result: TOD#1:1:1.000
STR_TO_DATE ({parameter 1})	Convert string to DATE. Converts the input string to a DATE value. (usage example) STR_TO_DATE('d#2019-1-2'), result: D#2019-1-2
STR_TO_DT ({parameter 1})	Convert string to DT. Converts the input string to a DT value. (usage example) STR_TO_DT('dt#2019-1-2-3:4:5'), result: DT#2019-1-2-3:4:5.000

1.4.5. Memory Operation

ADDROF({parameter 1})	Get the address [bit] of a variable. Only internal memory and output memory are applicable. (usage example) ADDROF(%QX10), result: 10
SIZEOF({parameter 1})	Find the size [bit] of a variable. (usage example) SIZEOF(%QX10), result: 1
CHG_ENDIAN ({parameter 1})	Endian change. (usage example) CHG_ENDIAN(16#1122), result: 16#2211
EDC_SUM_BYTE ({parameter 1}, {parameter 2})	Get the BYTE SUM value. To check communication error, obtain the SUM of all BYTEs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_SUM_WORD ({parameter 1}, {parameter 2})	Get the WORD SUM value. To check communication error, obtain the SUM of all WORDs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the word count.

EDC_SUM_DWORD ({parameter 1}, {parameter 2})	Get the DWORD SUM value. To check communication error, obtain the SUM of all DWORDs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the dword count.
EDC_XOR_BYTE ({parameter 1}, {parameter 2})	Get the BYTE XOR value. To check communication error, obtain the XOR of all BYTEs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_XOR_WORD ({parameter 1}, {parameter 2})	Get the WORD XOR value. To check communication error, obtain the XOR of all WORDs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the word count.
EDC_XOR_DWORD ({parameter 1}, {parameter 2})	Get the DWORD XOR value. To check communication error, obtain the XOR of all DWORDs in the data. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the dword count.
EDC_CRC_8DARC ({parameter 1}, {parameter 2})	Get the 8bit DARC CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_8I4321 ({parameter 1}, {parameter 2})	Get the 8bit I-432-1(ITU) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_8ICODE ({parameter 1}, {parameter 2})	Get the 8bit I-CODE CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_8MAXIMDOW ({parameter 1}, {parameter 2})	Get the 8bit MAXIM(DOW-CRC) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_8ROHC ({parameter 1}, {parameter 2})	Get the 8bit ROHC CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.

EDC_CRC_8SMBUS ({parameter 1}, {parameter 2})	Get the 8bit SMBus CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_8WCDMA ({parameter 1}, {parameter 2})	Get the 8bit WCDMA CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16ARC ({parameter 1}, {parameter 2})	Get the 16bit ARC(CRC-16, CRC-16/LHA, CRC-IBM) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16DDS110 ({parameter 1}, {parameter 2})	Get the 16bit DDS-110 CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16DECTR ({parameter 1}, {parameter 2})	Get the 16bit DECT-R(R-CRC-16) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16DNP ({parameter 1}, {parameter 2})	Get the 16bit DNP CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16EN13757 ({parameter 1}, {parameter 2})	Get the 16bit EN-13757 CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16MODBUS ({parameter 1}, {parameter 2})	Get the 16bit MODBUS CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_16UMTS ({parameter 1}, {parameter 2})	Get the 16bit UMTS(CRC-16/BUYPASS, CRC-16/VERIFONE) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.

EDC_CRC_16USB ({parameter 1}, {parameter 2})	Get the 16bit USB CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
EDC_CRC_32ISOHDLC ({parameter 1}, {parameter 2})	Get the 32bit ISO-HDLC(CRC-32, CRC-32/ADCCP, PKZIP, CRC-32/V-42) CRC value. Obtain CRC value for communication error check. The data to be calculated must be stored in internal memory. {parameter 1} is the address, {parameter 2} is the byte count.
GET_BOOL ({parameter 1})	Read the BOOL value. {parameter 1} is the address of internal memory.
GET_BYTE ({parameter 1})	Read the BYTE value. {parameter 1} is the address of internal memory.
GET_SINT ({parameter 1})	Read the SINT value. {parameter 1} is the address of internal memory.
GET_WORD ({parameter 1})	Read the WORD value. {parameter 1} is the address of internal memory.
GET_INT ({parameter 1})	Read the INT value. {parameter 1} is the address of internal memory.
GET_DWORD ({parameter 1})	Read the DWORD value. {parameter 1} is the address of internal memory.
GET_DINT ({parameter 1})	Read the DINT value. {parameter 1} is the address of internal memory.
GET_REAL ({parameter 1})	Read the REAL value. {parameter 1} is the address of internal memory.
GET_TIME ({parameter 1})	Read the TIME value. {parameter 1} is the address of internal memory.
GET_TOD ({parameter 1})	Read the TOD value. {parameter 1} is the address of internal memory.
GET_DATE ({parameter 1})	Read the DATE value. {parameter 1} is the address of internal memory.
GET_DT ({parameter 1})	Read the DT value. {parameter 1} is the address of internal memory.
SET_BOOL ({parameter 1}, {parameter 2})	Write a BOOL value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_BYTE ({parameter 1}, {parameter 2})	Write a BYTE value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.

SET_SINT ({parameter 1}, {parameter 2})	Write a SINT value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_WORD ({parameter 1}, {parameter 2})	Write a WORD value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_INT ({parameter 1}, {parameter 2})	Write a INT value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_DWORD ({parameter 1}, {parameter 2})	Write a DWORD value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_DINT ({parameter 1}, {parameter 2})	Write a DINT value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_REAL ({parameter 1}, {parameter 2})	Write a REAL value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_TIME ({parameter 1}, {parameter 2})	Write a TIME value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_TOD ({parameter 1}, {parameter 2})	Write a TOD value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_DATE ({parameter 1}, {parameter 2})	Write a DATE value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.
SET_DT ({parameter 1}, {parameter 2})	Write a DT value. {parameter 1} is the address of internal memory. {parameter 2} is the value to write.

1.4.6. Flow Control

SEL({parameter 1}, {parameter 2}, {parameter 3})	Choose one of two. If {parameter 1} is 0, it returns {parameter 2}, otherwise it returns {parameter 3}. {parameter 2} and {parameter 3} can be of any type.
MUX({parameter 1}, ...)	Choose one of several. {parameter 1} selects {parameter 2} when 0 and returns the next parameter every +1.

RETURN;		Exit from the work structure. Exits the current operation structure (PROG, FB, FUN). (Returns to the place where the current operation was called.) The call operation alone constitutes one row.
EXIT;		Exit from iteration. Breaks out of the currently executing iteration such as FOR, REPEAT, WHILE, etc.

IF	form	IF {selection condition 1} THEN
		{work row 1};
		ELSIF {selection condition 2} THEN
		{work row 2};
		ELSE
		{work row e};
		END_IF;

	explanation	IF lines start with IF and end with END_IF;. ELSIF and ELSE are used when necessary. Multiple ELSIFs can be used, and up to one ELSE can be used. Each {work row} can contain zero or more lines. If the result of {selection condition 1} is TRUE, {work row 1} is executed. If the result of {selection condition 1} is FALSE and the result of {selection condition 2} is TRUE, {work row 2} is executed. If all of the preceding {selection condition}s are FALSE, {work row e} is executed.

CASE	form	CASE {selection condition} OF
		{selection value 1}:
		{work row 1};
		{selection value 2}:
		{work row 2};
		ELSE
		{work row e};
		END_CASE;

	explanation	CASE lines start with CASE and end with END_CASE;. ELSE is used when necessary. You can specify more than one {selection value}. Multiple can be specified at the same time even within {selection value}. If the value of {selection condition} matches {selection value 1}, {work row 1} is executed. The rest is the same as in IF.

FOR	form	FOR {variable assignment} TO {end value} BY {increase value} DO
		{work row};
		END_FOR;

	explanation	FOR lines start with FOR and end with END_FOR;. In {variable assignment}, specify the variable and initial value of the repeat execution condition. (e.g. vInt1 := 1) {end value} is a condition to stop repeating. {increase value} is the value to add to the iteration variable at the end of each iteration. {work line} can contain zero or more lines. Iteration is performed while the iteration condition is maintained. If {increase value} is positive, the iteration condition is maintained as long as the value of the variable is less than or equal to {end value}. If {incremental value} is negative, the iteration condition is maintained as long as the value of the variable is greater than or equal to {end value}. {Increase value} defaults to 1. It is set to 1 if BY {increment value} is not used.

REPEAT	form	REPEAT
		{work row};
		UNTIL {repeat condition}
		END_REPEAT;

	explanation	REPEAT lines start with REPEAT and end with END_REPEAT;. {work row} can contain zero or more lines. It is repeated while {repeat condition} is TRUE.

WHILE	form	WHILE {repeat condition} DO
		{work row};
		END_WHILE;

	explanation	WHILE lines start with WHILE and end with END_WHILE;. {work row} can contain zero or more lines. It is repeated while {repeat condition} is TRUE.

1.4.7. System Command

BACKUP ({parameter 1}, {parameter 2})	Backs up the specified internal memory. When restarting, E5A initializes all values to 0 if there are no backup values in the internal memory, otherwise it initializes to the backup values. The return value is the number of bytes written. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
SEE_NW_DOMAIN()	Get the current hostname of E5A as a string.
SEE_NW_IP()	Get the current IP address of E5A as a string.

SEE_NW_MODE()	Get the current WiFi mode (AP, Station) of E5A as a string.
SEE_NW_SSID()	Get the current SSID of E5A as a string.
WA_ABLE_GEN ({parameter 1}, {parameter 2})	Check whether E5A can be written to internal memory from outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
WA_ABLE_OUT ({parameter 1}, {parameter 2})	Check whether it is possible to write to the output memory from outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
WA_DISABLE_GEN ({parameter 1}, {parameter 2})	Change the internal memory to be non-writable from the outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
WA_DISABLE_OUT ({parameter 1}, {parameter 2})	Change the outpt memory to be non-writable from the outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
WA_ENABLE_GEN ({parameter 1}, {parameter 2})	Change the internal memory to be writable from the outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.
WA_ENABLE_OUT ({parameter 1}, {parameter 2})	Change the output memory to be writable from the outside the E5A. {parameter 1} is the address of memory. {parameter 2} is the length in bits.

1.5. FB Type

In addition to the FB defined in the standard, the E5A adds several FB types for user convenience.

1.5.1. Standards Based

The IEC 61131-3 standard was first published in 1993, and the 2nd edition on which the DST was based was published in 2003. The most recent release was in 2013 as the 3rd edition.

E5A is described as a small computer running the programming language DST, but when the standard was created, there was a strong tendency to see PLC as a bundle of logic gates. This can be seen from the original expression of PLC as Programmable Logic Controller. So, standard FBs are designed as if there is a logic gate inside.

From a hardware point of view, a digital signal is a binary signal delimited by 0[V] (= Low, L for short) or Vcc[V] (= High, H for short). From the software point of view, among variable types, BOOL is most often used for purposes corresponding to digital signals. When matching digital signals to BOOL, it is generally processed as (L = FALSE, H = TRUE), but there are cases where it is processed as (L = TRUE, H = FALSE). The falling edge (↘) is the moment when the signal changes from H to L, and the rising edge (↗) is the moment when the signal changes from L to H.

The criterion for changing the state of the logic gate can be a level trigger or an edge trigger. Level trigger means to perform the specified action when the signal is H or L, and is called H active or L active in order. Edge trigger means that the signal performs the action specified by ↘ or ↗, and is called falling edge trigger or rising edge trigger in order.

Standard FBs of E5A recognize FALSE as L and TRUE as H. In the following description, H active is indicated by (H), L active is indicated by (L), falling edge trigger is indicated by (↘), and rising edge trigger is indicated by (↗). A standard FB handles the state each time it is called, so the call cycle must be chosen appropriately.

CTD	name	Down Counter.		

	local variable	input	CD : BOOL;
			LD : BOOL;
			PV : INT;
				count down input signal. (↗)
				initial value setting signal. (H)
				Initial value.

		output	Q : BOOL;
			CV : INT;
				status output signal.
				present value.

	explanation	When LD is H, copy PV to CV. When CD is ↗, add -1 to CV. When CV <= 0, Q is 1. Otherwise, Q is 0.		
CTU	name	Up Counter.		

	local variable	input	CU : BOOL;
			R : BOOL;
			PV : INT;
				count up input signal. (↗)
				initial value setting signal. (H)
				reference value.

		output	Q : BOOL;
			CV : INT;
				status output signal.
				present value.

	explanation	When R is H, set CV to 0. When CU is ↗, add +1 to CV. When CV >= PV, Q is 1. Otherwise, Q is 0.		

CTUD	name	Up/Down Counter.		
	local variable	input	CU : BOOL;	count up input signal. (↗)
			CD : BOOL;	count down input signal. (↗)
			R : BOOL;	initial value setting signal. (H)
			LD : BOOL	initial value setting signal. (H)
			PV : INT;	reference value.
		output	QU : BOOL;	status output signal.
			QD : BOOL;	status output signal.
			CV : INT;	present value.
	explanation	When R is H, set CV to 0. When LD is H, copy PV to CV. When CU is ↗, add +1 to CV. When CD is ↗, add -1 to CV. When CV >= PV, QU is 1. Otherwise, QU is 0. When CV <= 0, QD is 1. Otherwise, QD is 0.		
F_TRIG	name	Falling Edge Detector.		
	local variable	input	CLK : BOOL;	input signal. (↘)
		output	Q : BOOL;	status output signal.
	explanation	When CLK is ↘, Q is 1. Otherwise, Q is 0.		
R_TRIG	name	Rising Edge Detector.		
	local variable	input	CLK : BOOL;	input signal. (↗)
		output	Q : BOOL;	status output signal.
	explanation	When CLK is ↗, Q is 1. Otherwise, Q is 0.		
RS	name	RS Flipflop.		
	local variable	input	S : BOOL;	input signal. (H)
			R1 : BOOL;	input signal. (H)
		output	Q : BOOL;	status output signal.
	explanation	When R1 is H, Q is 0. When R1 is L and S is H, Q is 1. Otherwise, Q retains its value.		
RTC	name	Real Time Clock.		
	local variable	input	EN : BOOL;	input signal. (↗)
			PDT : DT;	set time.
		output	Q : BOOL;	status output signal.
			CDT : DT;	current time.
	explanation	When EN is ↗, time reset to PDT. Copy EN to Q. CDT is the time of the call.		

SEMA	name	Semaphore.		

	local variable	input	CLAIM : BOOL;
			RELEASE:BOOL;
				input signal. (H)
				input signal. (H)

		output	BUSY : BOOL;	status output signal.
	explanation	If CLAIM is H and the request is successful, BUSY is 0. If CLAIM is H and the request fails, BUSY is 1. If RELEASE is H and the request is successful, BUSY is 0. This function is used to control access to shared resources and is assigned to the work object. Release must be performed on the work object that has acquired the privilege.		
SR	name	SR Flipflop.		

	local variable	input	S1 : BOOL;
			R : BOOL;
				input signal. (H)
				input signal. (H)

		output	Q : BOOL;	status output signal.
	explanation	When S1 is H, Q is 1. When S1 is L and R is H, Q is 0. In all other cases, Q remains its value.		
TOF	name	Timer OFF Delay.		

	local variable	input	IN : BOOL;
			PT : TIME;
				input signal. (H)
				delay time.

		output	Q : BOOL;
			ET : TIME;
				status output signal.
				elapsed time.

	explanation	When IN is H, Q is 1, ET is 0. When IN is ↘, ET measurement starts. When IN is L and ET < PT, ET continues to measure. When IN is L and ET >= PT, stop measuring ET, Q is 0. Operates a timer that keeps Q ON for a delay as long as PT according to the IN signal.		

TON	name	Timer ON Delay.		

	local
	variable
		input	IN : BOOL;
			PT : TIME;
				input signal. (H)
				delay time.

		output	Q : BOOL;
			ET : TIME;
				status output signal.
				elapsed time.

	explanation	When IN is L, Q is 0, ET is 0. When IN is ↗, ET measurement starts. When IN is H and ET < PT, ET continues to measure. When IN is H and ET >= PT, stop measuring ET, Q is 1. Operates a timer that turns ON Q after a delay as long as PT according to the IN signal.		
TP	name	Pulse Generator.		

	local
	variable
		input	IN : BOOL;
			PT : TIME;
				input signal. (H)
				delay time.

		output	Q : BOOL;
			ET : TIME;
				status output signal.
				elapsed time.

	explanation	When Q is 0 and IN is ↗, ET measurement starts. Q is 1. When ET < PT, ET continues to measure. When ET >= PT, stop measuring ET, Q is 0. When IN is L and ET >= PT, ET is 0. Operates a timer that turns on Q only for a delay as long as PT according to the IN signal.		

1.5.2. PID Control

PID control was first introduced in 1922 among automatic control technologies (officially named Control Theory) that have been developed since the late 19th century, and is the most well-known and widely used automatic control technology.

More information on PID control can be found in the English version of Wikipedia (https://en.wikipedia.org/wiki/PID_controller).

PID	name	PID controller.		

	local
	variable
		input	KP : USINT;
			KI : USINT;
			KD : USINT;
			HEND : INT;
			LEND : INT;
			HIW : INT;
				proportional constant. range: 1~100
				integral coefficient. range: 0~100
				differential coefficient. range: 0~100
				upper limit of control range.
				lower limit of control range.
				upper limit of integral term application.

			LIW : INT;
			SP : INT;
			PV : INT;
				lower limit of integral term application.
				target value.
				present value.

		output	EC : USINT;
			CV : INT;
				error code.
				control value[%]. range: -100~100

The PID FB, like the standard FB, processes the state each time it is called. It is common usage to periodically update only PV among input values. The call cycle determines the rate of response, so the user should select the call cycle appropriately.

Since CV is expressed as a percentage, the user must perform scaling according to the actual control range when applying to the control target. If CV is 100, it controls as much as the maximum output, if it is 0, it turns off the output, if it is −100, it controls as much as the maximum output in the reverse direction.

The influence of KP, KI, and KD is determined by their ratio to each other. Those set to (KP=2, KI=1, KD=0) and those set to (KP=100, KI=50, KD=0) have the same characteristics.

The control range specified by HEND and LEDN is the limit of the values that SP and PV can have. If the SP or PV is out of the control range, an error occurs.

HIW and LIW are values used for Integral Windup function applied to prevent problems caused by excessive integration of the integral term during the PID control process. If the PV is outside the range of LIW to HIW, the integral term is ignored in the calculation.

If an error occurs, the work object is aborted, so the task after the FB call is not executed. Since the error code is stored in the EC, the problem analysis is possible only when the error check is performed in the next execution cycle of the work object or in another work object. However, in the test phase using E5aLoader, the error content and location are displayed on the screen, so the user can refer to this and correct the DST.

1.5.3. Serial Communication

RS485 is linked to SER_1 in RESOURCE. Since the initial operation mode of SER_1 is closed, it must be set as CONF_SER1 FB for communication.

CONF_SER1	name	SER_1 communication settings.		

	local variable	input	GET_SET : BOOL;
			RATE : UDINT;
			PARITY : USINT;
			DATABITS : USINT;
			STOPBIT : BOOL;
			MODE : USINT;
			SLAVE_ID : USINT;
				setting direction.
				Bit Rate[bps]. range: 300~3M
				Parity. range: 0~4
				data Bits[bit]. range: 5~8
				stop Bit.
				operating mode. range: 0~3
				Slave ID. range: 1~247

		output	EC : UINT;	error code.

When FB is called, if GET_SET is 0, set values are read from the current set state, and if it is 1, new set values are written. When reading the set values, RATE, PARITY, DATABITS, STOPBIT, MODE, SLAVE_ID, etc. are updated. When writing set values, an error occurs if the settings are not as requested.

RATE is the bit rate setting of SER_1, and the unit is [bps].

The meaning of PARITY values is 0(=no parity), 1(=even parity), 2(=odd parity), 3(=zero parity), 4(=one parity).

DATABITS is the data bit size of serial communication.

The meaning of the STOPBIT value is 0(=no stop bit), 1(=1 stop bit).

The operation mode of SER_1 is one of closed, MODBUS RTU master, MODBUS RTU slave, and user defined protocol serial. You can change the operating mode only when the current operating mode is closed. This includes cases where you only want to change the RATE.

The meaning of MODE value is 0(=closed), 1(=MODBUS RTU master), 2(=MODBUS RTU slave), 3(=user defined protocol serial).

SLAVE_ID is a valid value only when the operation mode is MODBUS RTU slave. In other operating modes, no error occurs even if its value is out of range.

COMM_SER1	name	MODBUS RTU master communication.		

	local variable	input	READ_WRITE : BOOL;
			SLAVE_ID : USINT;
			SLAVE_AREA : USINT;
			SLAVE_ADDR : UINT;
			DATA_COUNT : UINT;
			MEM_ADDR : UINT;
				Data direction.
				Slave ID. range: 1~247
				access area. range: 0~4
				Slave's memory address.
				number of data.
				address of internal memory.

		output	EC : UINT;	error code.

If the operation mode is MODBUS RTU master, E5A can read or write the value of the counterpart device that is a MODBUS RTU slave. In this case, COMM_SER1 FB is used.

The meaning of the READ_WRITE value is 0(=read), 1(=write). If the data direction is read, E5A gets data from the other device, and if the data direction is write, E5A sends data to the other device.

SLAVE_ID is a value that identifies the counterpart device on RS485.

The meaning of SLAVE_AREA value is 0(=general register), 1(=coil status), 2(=input status), 3(=holding register), 4(=input register). If the counterpart device is E5A, it is a value that separates the memory area.

SLAVE_ADDR is the address in the access area. In coil status and input status, bit unit is used, and word (16bit) unit is used in general register, holding register, and input register.

DATA_COUNT is the number of data to read or write.

MEM_ADDR is the address of internal memory.

An error occurs if processing as requested by the input local variable is not possible. However, if a problem occurs in the communication process with the counterpart device, it is reflected only in the EC and does not affect the execution of the work object.

COMM_SERUSR	name	User defined protocol serial communication.		
	local variable	input	RECV_SEND : USINT;	communication type.
			SEND_ADDR : UINT;	transmission block address.
			SEND_LEN : UINT;	transmission length [byte]. range: 1~255
			RECV_ADDR : UINT;	receive block address.
			RECV_LEN : UINT;	receive maximum length [byte]. range: 1~255
		output	EC : UINT;	error code.

If the operation mode is the user defined protocol serial, E5A can send and receive data with the counterpart device operated by the user defined protocol serial. In this case, COMM_SERUSR FB is used.

The meaning of the RECV_SEND value is 0(=receive), 1(=send), 2(=send & receive). For receive, only RECV_ADDR and RECV_LEN are valid, and for send, only SEND_ADDR and SEND_LEN are valid. In case of send & receive, send first and then receive.

The number of bytes actually transmitted and received is stored in the first byte of the transmit block and receive block. The data sent and received starts from the 2nd byte. All must be located in the internal memory area.

SEND_LEN is the number of bytes of data to send, and RECV_LEN is the size (number of bytes) of the data buffer to receive.

1.5.4. Ethernet Server Communication

The TCP server built in E5A is linked with ETH_1 of RESOURCE. The initial operating mode is closed.

CONF_ETH1	name	ETH_1 communication settings.		
	local variable	input	GET_SET : BOOL;	setting direction.
			IPV4_6 : BOOL;	IP type.
			PORT : UINT;	Port number.
			MODE : USINT;	operating mode. Range: 0~3
			E5D_HOST : STRING;	(spare).
		output	EC : UINT;	error code.

When FB is called, if GET_SET is 0, set values are read from the current set state, and if GET_SET is 1, new set values are written. When reading the set values, IPV4_6, PORT, MODE, E5D_HOST, etc. are updated. When writing set values, an error occurs if the settings are not as requested.

The meaning of the IPV4_6 value is 0(=IPv4), 1(=IPv6).

PORT is the port number of the TCP server. In the well-known port area less than 1024, only ports assigned to operation mode can be opened (MODBUS TCP server = 502, user defined protocol TCP server = 23). To designate a port number arbitrarily, use a value between 1024 and 49151.

The meaning of the MODE value is 0(=closed), 1(=MODBUS TCP server), 2 (=user defined protocol TCP server), 3(=E5D server). The E5D server is a spare, not yet available.

COMM_ETHUSR	name	User defined protocol TCP server communication.		
	local variable	input	RECV_SEND : USINT;	communication type.
			SEND_ADDR : UINT;	transmission block address.
			SEND_LEN : UINT;	transmission length [byte]. range: 1~255
			RECV_ADDR : UINT;	receive block address.
			RECV_LEN : UINT;	receive maximum length [byte]. range: 1~255
		output	EC : UINT;	error code.

If the operation mode is user defined protocol TCP server, E5A can send and receive data with the counterpart device operated as user defined protocol TCP client. In this case, COMM_ETHUSR FB is used.

The meaning of the RECV_SEND value is 0(=receive), 1(=send), 2(=send & receive). For receive, only RECV_ADDR and RECV_LEN are valid, and for send, only SEND_ADDR and SEND_LEN are valid. In case of send & receive, send first and then receive.

The number of bytes actually transmitted and received is stored in the first byte of the transmit block and receive block. The data sent and received starts from the 2nd byte. All must be located in the internal memory area.

SEND_LEN is the number of bytes of data to send, and RECV_LEN is the size (number of bytes) of the data buffer to receive.

1.5.5. Ethernet Client Communication

The TCP client built in E5A is linked with ETH_2 of RESOURCE. The initial operating mode is closed.

CONF_ETH2	name	ETH_2 communication settings.		

	local variable	input	GET_SET : BOOL;
			IPV4_6 : BOOL;
			HOST : STRING;
			PORT : UINT;
			MODE : USINT;
			UNIT_ID : USINT;
				setting direction.
				IP type.
				Server address.
				Port number.
				operating mode. range: 0~2
				Unit ID.

		output	EC : UINT;	error code.

When FB is called, if GET_SET is 0, set values are read from the current set state, and if GET_SET is 1, new set values are written. When reading the set values, IPV4_6, PORT, MODE, UNIT_ID, HOST, etc. are updated. When writing set values, an error occurs if the settings are not as requested.

The meaning of the IPV4_6 value is 0(=IPv4), 1(=IPv6).

HOST is the IP address or domain name of the counterpart device.

PORT is the port number of the TCP server. You must enter the port number opened by the counterpart device.

The meaning of the MODE value is 0(=closed), 1(=MODBUS TCP client), 2(=user defined protocol TCP client).

UNIT_ID is used when the counterpart device is a MODBUS TCP server and responds only by specifying a specific unit ID. If the counterpart device is E5A or responds regardless of unit ID, no value is specified.

COMM_ETH2	name	MODBUS TCP client communication.		

	local variable	input	READ_WRITE : BOOL;
			SERVER_AREA : USINT;
			SERVER_ADDR : UINT;
			DATA_COUNT : UINT;
			MEM_ADDR : UINT;
				Data direction.
				access area. range: 0~4
				Slave's memory address.
				number of data.
				address of internal memory.

		output	EC : UINT;	error code.

If the operation mode is MODBUS TCP client, E5A can read or write the value of the counterpart device, which is a MODBUS TCP server. In this case, COMM_ETH2 FB is used.

The meaning of the READ_WRITE value is 0(=read), 1(=write). If the data direction is read, E5A gets data from the other device, and if the data direction is write, E5A sends data to the other device.

The meaning of the SERVER_AREA value is 0(=general register), 1(=coil status), 2(=input status), 3(=holding register), 4(=input register). If the counterpart device is E5A, it is a value that separates the memory area.

SERVER_ADDR is the address in the access area. In coil status and input status, bit unit is used, and word (16bit) unit is used in general register, holding register, and input register.

DATA_COUNT is the number of data to read or write.

MEM_ADDR is the address of internal memory.

An error occurs if processing as requested by the input local variable is not possible. However, if a problem occurs in the communication process with the counterpart device, it is reflected only in the EC and does not affect the execution of the work object.

COMM_ETHUSR	name	User defined protocol TCP client communication.		
	local variable	input	RECV_SEND : USINT;	communication type.
			SEND_ADDR : UINT;	transmission block address.
			SEND_LEN : UINT;	transmission length [byte]. range: 1~255
			RECV_ADDR : UINT;	receive block address.
			RECV_LEN : UINT;	receive maximum length [byte]. range: 1~255
		output	EC : UINT;	error code.

If the operation mode is user defined protocol TCP client, E5A can send and receive data with the counterpart device operated as user defined protocol TCP server. In this case, COMM_ETHUSR FB is used.

The meaning of the RECV_SEND value is 0 (=receive), 1(=send), 2(=send & receive). For receive, only RECV_ADDR and RECV_LEN are valid, and for send, only SEND_ADDR and SEND_LEN are valid. In case of send & receive, send first and then receive.

The number of bytes actually transmitted and received is stored in the first byte of the transmit block and receive block. The data sent and received starts from the 2nd byte. All must be located in the internal memory area.

SEND_LEN is the number of bytes of data to send, and RECV_LEN is the size (number of bytes) of the data buffer to receive.

2. Usage Example

Create basic0.dst by collecting usage examples for work directives. If you run it row by row in E5aLoader, you can check the result.

```
CONFIGURATION nameOfConf
  VAR_GLOBAL
    gTest : STRING := '123456789';
  END_VAR

  RESOURCE myDevice ON CPU
    TASK taskInit (SINGLE := TRUE, PRIORITY := 3);
    TASK taskSync (INTERVAL := t#3s, PRIORITY := 7);
    PROGRAM WITH taskInit : ProgInit();
    PROGRAM WITH taskSync : ProgMain();
  END_RESOURCE
END_CONFIGURATION

PROGRAM ProgInit
  VAR
    vLen : INT;
    vAddr, vDword1 : DWORD;
    vDint1 : DINT;
    vReal1 : REAL;
    vStr1 : STRING;
    vTime1 : TIME;
    vTod1 : TOD;
    vDate1 : DATE;
    vDt1 : DT;
  END_VAR

  vDword1 := TRUE & FALSE; (* FALSE *)
  vDword1 := TRUE | FALSE; (* TRUE *)
  vDword1 := !TRUE; (* FALSE *)
  vDword1 := TRUE ^ FALSE; (* TRUE *)
  vDword1 := 2#1010 AND 2#1100; (* 16#8 *)
  vDword1 := 2#1010 OR 2#1100; (* 16#E *)
  vDword1 := NOT 2#1010; (* 16#FFFF_FFF5 *)
  vDword1 := 2#1010 XOR 2#1100; (* 2#0110 *)
  vDword1 := 1 >= 2; (* FALSE *)
  vDword1 := 1 GT 2; (* FALSE *)
  vDword1 := 1 <= 2; (* TRUE *)
```

```
vDword1 := 1 LT 2; (* TRUE *)
vDword1 := 1 = 2; (* FALSE *)
vDword1 := 1 NE 2; (* TRUE *)

vReal1 := 1 + 2; (* 3 *)
vReal1 := 1 - 2; (* -1 *)
vReal1 := 1 * 2; (* 2 *)
vReal1 := 1 / 2; (* 0.5 *)
vReal1 := 5 MODULO 4; (* 1 *)
vReal1 := 5 ** 2; (* 25 *)
vReal1 := ABS(-5); (* 5 *)
vReal1 := SQRT(4); (* 2 *)
vReal1 := EXP(1); (* ~ 2.71828 *)
vReal1 := LN(EXP(1)); (* 1 *)
vReal1 := LOG(10); (* 1 *)
vReal1 := MAX(2,3,4); (* 4 *)
vReal1 := MAX(2,1,4); (* 4 *)
vReal1 := MIN(2,3,4); (* 2 *)
vReal1 := MIN(2,1,4); (* 1 *)
vReal1 := LIMIT(2,3,4); (* 3 *)
vReal1 := LIMIT(2,1,4); (* 2 *)
vDword1 := ROL(1,2); (* 4 *)
vDword1 := ROR(1,2); (* 16#4000_0000 *)
vDword1 := SHL(1,2); (* 4 *)
vDword1 := SHR(1,2); (* 0 *)

vReal1 := ACOS(0.5); (* 60 *)
vReal1 := ASIN(0.5); (* 30 *)
vReal1 := ATAN(0.5); (* ~ 26.565 *)
vReal1 := COS(50); (* ~ 0.643 *)
vReal1 := SIN(50); (* ~ 0.766 *)
vReal1 := TAN(50); (* ~ 1.192 *)

vStr1 := CONCAT('Ab', ' Cd'); (* 'Ab Cd' *)
vStr1 := DELETE('Ab Cd', 2, -1); (* 'Ab' *)
vDword1 := FIND('Ab Bb ', 'b '); (* 1 *)
vStr1 := INSERT('Ab Cd', ' x', 2); (* 'Ab x Cd' *)
vStr1 := LEFT('Ab Cd', 2); (* 'Ab' *)
vDword1 := LEN('Ab Cd'); (* 5 *)
vStr1 := MID('Ab Cd', 2, 2); (* ' C' *)
vStr1 := REPLACE('Ab Cd', 'x', 1, 3); (* 'Axd' *)
vStr1 := RIGHT('Ab Cd', 2); (* 'Cd' *)
```

```
vStr1 := STR_FROM_BOOL(1); (* 'TRUE' *)
vStr1 := STR_FROM_BYTE(1); (* '1' *)
vStr1 := STR_FROM_SINT(1); (* '1' *)
vStr1 := STR_FROM_WORD(1); (* '1' *)
vStr1 := STR_FROM_INT(1); (* '1' *)
vStr1 := STR_FROM_DWORD(1); (* '1' *)
vStr1 := STR_FROM_DINT(1); (* '1' *)
vStr1 := STR_FROM_REAL(1); (* '1E000' *)
vStr1 := STR_FROM_TIME(t#1s); (* 'T#0d0h0m1s0ms' *)
vStr1 := STR_FROM_TOD(tod#1:1:1); (* 'TOD#1:1:1.000' *)
vStr1 := STR_FROM_DATE(d#2019-1-2); (* 'D#2019-1-2' *)
vStr1 := STR_FROM_DT(dt#2019-1-2-3:4:5); (* 'DT#2019-1-2-3:4:5.000' *)
vDword1 := STR_TO_BOOL('TRUE'); (* 1 *)
vDword1 := STR_TO_BYTE('1'); (* 1 *)
vDword1 := STR_TO_SINT('1'); (* 1 *)
vDword1 := STR_TO_WORD('1'); (* 1 *)
vDword1 := STR_TO_INT('1'); (* 1 *)
vDword1 := STR_TO_DWORD('1'); (* 1 *)
vDword1 := STR_TO_DINT('1'); (* 1 *)
vReal1 := STR_TO_REAL('1'); (* 1.000 *)
vTime1 := STR_TO_TIME('t#1s'); (* T#0d0h0m1s0ms *)
vDt1 := STR_TO_TOD('tod#1:1:1'); (* TOD#1:1:1.000 *)
vDt1 := STR_TO_DATE('d#2019-1-2'); (* D#2019-1-2 *)
vDt1 := STR_TO_DT('dt#2019-1-2-3:4:5'); (* DT#2019-1-2-3:4:5.000 *)

vDword1 := ADDROF(%QX10); (* 10 *)
vDword1 := SIZEOF(%QX10); (* 1 *)
vDword1 := CHG_ENDIAN(16#1122); (* 16#2211 *)
vAddr := ADDROF(gTest);
vLen := LEN(gTest);
vDword1 := EDC_SUM_BYTE(vAddr, vLen); (* 16#DD *)
vDword1 := EDC_SUM_WORD(vAddr, SHR(vLen, 1)); (* 16#D4D0 *)
vDword1 := EDC_SUM_DWORD(vAddr, SHR(vLen, 2)); (* 16#6C6A6866 *)
vDword1 := EDC_XOR_BYTE(vAddr, vLen); (* 16#31 *)
vDword1 := EDC_XOR_WORD(vAddr, SHR(vLen, 1)); (* 16#800 *)
vDword1 := EDC_XOR_DWORD(vAddr, SHR(vLen, 2)); (* 16#C040404 *)
vDword1 := EDC_CRC_8DARC(vAddr, vLen); (* 16#15 *)
vDword1 := EDC_CRC_8I4321(vAddr, vLen); (* 16#A1 *)
vDword1 := EDC_CRC_8ICODE(vAddr, vLen); (* 16#7E *)
vDword1 := EDC_CRC_8MAXIMDOW(vAddr, vLen); (* 16#A1 *)
vDword1 := EDC_CRC_8ROHC(vAddr, vLen); (* 16#D0 *)
vDword1 := EDC_CRC_8SMBUS(vAddr, vLen); (* 16#F4 *)
```

```
  vDword1 := EDC_CRC_8WCDMA(vAddr, vLen); (* 16#25 *)
  vDword1 := EDC_CRC_16ARC(vAddr, vLen); (* 16#BB3D *)
  vDword1 := EDC_CRC_16DDS110(vAddr, vLen); (* 16#9ECF *)
  vDword1 := EDC_CRC_16DECTR(vAddr, vLen); (* 16#7E *)
  vDword1 := EDC_CRC_16DNP(vAddr, vLen); (* 16#EA82 *)
  vDword1 := EDC_CRC_16EN13757(vAddr, vLen); (* 16#C2B7 *)
  vDword1 := EDC_CRC_16MODBUS(vAddr, vLen); (* 16#4B37 *)
  vDword1 := EDC_CRC_16UMTS(vAddr, vLen); (* 16#FEE8 *)
  vDword1 := EDC_CRC_32ISOHDLC(vAddr, vLen); (* 16#CBF43926 *)
  vAddr := 64;
  vDword1 := GET_BOOL(vAddr); (* 1 *)
  vDword1 := GET_BYTE(vAddr); (* 57 *)
  vDint1 := GET_SINT(vAddr); (* 57 *)
  vDword1 := GET_WORD(vAddr); (* 57 *)
  vDint1 := GET_INT(vAddr); (* 57 *)
  vDword1 := GET_DWORD(vAddr); (* 57 *)
  vDint1 := GET_DINT(vAddr); (* 57 *)
  vReal1 := GET_REAL(vAddr); (* 0.000 *)
  vTime1 := GET_TIME(vAddr); (* T#0d0h0m0s57ms *)
  vTod1 := GET_TOD(vAddr); (* TOD#0:0:0.057 *)
  vDate1 := GET_DATE(vAddr); (* D#1900-1-1 *)
  vDt1 := GET_DT(vAddr); (* DT#1900-1-1-0:0:0.057 *)
  vDword1 := SET_BOOL(vAddr, 20190102); (* 1 *)
  vDword1 := SET_BYTE(vAddr, 20190102); (* 150 *)
  vDint1 := SET_SINT(vAddr, 20190102); (* -106 *)
  vDword1 := SET_WORD(vAddr, 20190102); (* 5014 *)
  vDint1 := SET_INT(vAddr, 20190102); (* 5014 *)
  vDword1 := SET_DWORD(vAddr, 20190102); (* 20190102 *)
  vDint1 := SET_DINT(vAddr, 20190102); (* 20190102 *)
  vReal1 := SET_REAL(vAddr, 20190102); (* 20190102.000 *)
  vTime1 := SET_TIME(vAddr, t#1d2h3m4s); (* T#1d2h3m4s0ms *)
  vTod1 := SET_TOD(vAddr, dt#2019-1-2-3:4:5.678); (* TOD#3:4:5.678 *)
  vDate1 := SET_DATE(vAddr, dt#2019-1-2-3:4:5.678); (* D#2019-1-2 *)
  vDt1 := SET_DT(vAddr, dt#2019-1-2-3:4:5.678); (* DT#2019-1-2-3:4:5.678 *)

  vDword1 := 1;
END_PROGRAM

PROGRAM ProgMain
  VAR
    vInt1, vInt2, vInt3 : INT := 12;
    vInt4 : ARRAY[3] OF INT := 3;
  vDword1 : DWORD := 34;
  vStr1 : STRING;
END_VAR

vInt1 := SEL(1, vInt2, vDword1); (* 34 *)
vDword1 := MUX(1, vInt1, vInt2, vInt3); (* 12 *)
vStr1 := FunAct(vInt1); (* '34' *)
vStr1 := FunAct(-1); (* '34' *)
vStr1 := FunAct(1); (* 'Act number is 1.$l$r' *)
vStr1 := FunAct(6); (* '6' *)
vStr1 := FunAct(12); (* 'Just match!$l$r' *)

(* The following is for understanding the flow of iteration tasks. *)
FOR vInt4[0] := 0 TO 3 DO
  vInt3 := vInt3 + 3;
  REPEAT
    vInt4[1] := vInt4[1] - 2;
    WHILE vInt4[2] <= 100 DO
      vInt4[2] := vInt4[2] + 1;
      EXIT;
    END_WHILE;
  UNTIL vInt4[1] > 0
  END_REPEAT;
END_FOR;

(* The following is an example of using a system command. *)
vStr1 := SEE_NW_IP();
vStr1 := SEE_NW_SSID();
vStr1 := SEE_NW_MODE();
vStr1 := SEE_NW_DOMAIN();
vDword1 := WA_ABLE_GEN(8, 8); (* FALSE *)
vDword1 := WA_ENABLE_GEN(8, 8); (* TRUE *)
vDword1 := WA_ABLE_GEN(8, 8); (* TRUE *)
vDword1 := WA_ABLE_GEN(7, 1); (* FALSE *)
vDword1 := WA_ABLE_GEN(16, 1); (* FALSE *)
vDword1 := WA_DISABLE_GEN(9, 3); (* TRUE *)
vDword1 := WA_ABLE_GEN(8, 8); (* FALSE *)
vDword1 := WA_ABLE_OUT(0, 2); (* FALSE *)
vDword1 := WA_ENABLE_OUT(0, 2); (* TRUE *)
vDword1 := WA_ABLE_OUT(0, 2); (* TRUE *)
vDword1 := WA_DISABLE_OUT(1, 1); (* TRUE *)
vDword1 := WA_ABLE_OUT(0, 2); (* FALSE *)
```

```
   vDword1 := WA_ABLE_OUT(0, 1); (* TRUE *)

   vDword1 := 0;
END_PROGRAM

FUNCTION FunAct : STRING
   VAR_INPUT
      iAct1 : INT;
   END_VAR
   VAR
      vStr1 : STRING;
   END_VAR

   IF iAct1 > 100 THEN
      RETURN;
   ELSIF iAct1 < 0 THEN
      RETURN;
   ELSE
      CASE iAct1 OF
         0, 1, 2, 3, 4, 5:
            vStr1 := CONCAT('Act number is ', STR_FROM_INT(iAct1), '.');
         12:
            vStr1 := 'Just match!';
      ELSE
         FunAct := STR_FROM_BYTE(iAct1);
         RETURN;
      END_CASE;
   END_IF;
   FunAct := CONCAT(vStr1, '$l$r');
END_FUNCTION
```

An example of using a standard FB and a PID FB is written in pid0.dst. Set the target temperature by connecting a variable resistor to UI0, and measure the current temperature with PT1000 in UI1. It controls PID in 1:1:1 ratio, and the temperature range is -200~800[℃]. Integral windup function operates outside -100~400[℃]. A heater with controllable intensity is connected to AO0. PID control is performed every 5 seconds.

```
CONFIGURATION nameOfConf
   VAR_GLOBAL
      gPid : PID;
      gTsp AT %IW16 : INT; (* UI0 = target value *)
      gTpv AT %IW17 : INT; (* UI1 = present value *)
```

```
  END_VAR

  RESOURCE myDevice ON CPU
    TASK taskInit (SINGLE := TRUE, PRIORITY := 1);
    TASK taskSync (INTERVAL := t#5s, PRIORITY := 2);
    PROGRAM WITH taskInit : ProgInit();
    PROGRAM WITH taskSync : ProgMain();
  END_RESOURCE
END_CONFIGURATION

PROGRAM ProgInit
  gPid.Kp := 1;
  gPid.Ki := 1;
  gPid.Kd := 1;
  gPid.Hend := 8000;
  gPid.Lend := -2000;
  gPid.Hiw := 4000;
  gPid.Liw := -1000;
  gPid.Sp := 1000;
  gPid.Pv := 1000;
END_PROGRAM

PROGRAM ProgMain
  IF (%IX0 = 0) | (%IX1 = 0) THEN (* Temperature sensor not detected. *)
    %QW16 := 0;
    RETURN;
  END_IF;

  gPid(Sp := gTsp, Pv := gTpv);
  IF (gPid.EC <> 0) | (gPid.Cv <= 0) THEN (* An error occurred, or the measured value exceeded the set value. *)
    %QW16 := 0;
    RETURN;
  END_IF;

  %QW16 := gPid.Cv * 100; (* AO0 is the control value *)
END_PROGRAM
```

[DST 2] pid0.dst

An example of using MODBUS RTU communication is written as rtu0.dst(slave) and rtu1.dst(master). Two E5As are required for testing. Connect two E5As by RS485. Communication setting is 9600[bps], N/8/1.

E5A acting as a slave changes some of its internal memory and output memory to writable and waits. Set the device's slave ID to 11.

E5A acting as Master tries the MODBUS protocol available to slave E5A one by one in order.

```
CONFIGURATION nameOfConf
  RESOURCE extLine1 ON SER_1
    VAR_GLOBAL
      gConf : CONF_SER1;
    END_VAR

    TASK taskInit (SINGLE := TRUE, PRIORITY := 1);
    TASK taskSync (INTERVAL := t#5s, PRIORITY := 2);
    PROGRAM pgm1Init WITH taskInit : Prog1Init();
    PROGRAM WITH taskSync : Prog1Sync();
  END_RESOURCE
END_CONFIGURATION

PROGRAM Prog1Init
  VAR
    loBool : BOOL;
  END_VAR

  (* Communication settings: SET, MODBUS RTU slave mode, slave ID is 11, 9600/N/8/1 *)
  extLine1.gConf(GET_SET := 1, MODE := 2, SLAVE_ID := 11, RATE := 9600, PARITY := 0,
DATABITS := 8, STOPBIT := 1);

  (* Disable write protection for the workspace *)
  loBool := WA_ENABLE_GEN(0, 16);
  loBool := WA_ENABLE_GEN(2048 * 16, 16);
  loBool := WA_ENABLE_GEN(4095 * 16, 16);

  loBool := WA_ENABLE_OUT(0, 1);
  loBool := WA_ENABLE_OUT(16384, 1);
  loBool := WA_ENABLE_OUT(32767, 1);
  loBool := WA_ENABLE_OUT(32768, 1);

  loBool := WA_ENABLE_OUT(16383, 2);
  IF loBool = FALSE THEN
    loBool := WA_ENABLE_OUT(16383, 1);
  END_IF;
  loBool := WA_ENABLE_OUT(16384, 2);
  loBool := WA_ENABLE_OUT(32766, 2);
  loBool := WA_ENABLE_OUT(32767, 2);
  loBool := WA_ENABLE_OUT(0, 16);
  loBool := WA_ENABLE_OUT(1024 * 16, 16);
  loBool := WA_ENABLE_OUT(2047 * 16, 16);
  loBool := WA_ENABLE_OUT(4095 * 16, 16);

  loBool := WA_ENABLE_OUT(1023 * 16, 2 * 16);
  IF loBool = FALSE THEN
    loBool := WA_ENABLE_OUT(1023 * 16, 16);
  END_IF;
  loBool := WA_ENABLE_OUT(1024 * 16, 2 * 16);
  loBool := WA_ENABLE_OUT(2046 * 16, 2 * 16);
  loBool := WA_ENABLE_OUT(2047 * 16, 2 * 16);
END_PROGRAM

PROGRAM Prog1Sync
  VAR
    vDw : DWORD;
  END_VAR

  vDw := 0;
END_PROGRAM
```

[DST 3] rtu0.dst

```
CONFIGURATION nameOfConf
  TYPE T_CMD:
    STRUCT
      rw : BOOL; (* read/write *)
      sar : USINT; (* area *)
      sad : UINT; (* address *)
    END_STRUCT
  END_TYPE

  VAR_GLOBAL
    gWait : BOOL;
    gSar, gSad, gRw : SINT;
    gWord0, gWord1 : WORD;
    gRtc : RTC;
    gCmd : ARRAY[5,3,2] OF T_CMD; (* area, addr, r/w *)
  END_VAR

  RESOURCE extLine1 ON SER_1
```

```
    VAR_GLOBAL
      gConf : CONF_SER1;
      gComm : COMM_SER1;
    END_VAR

    TASK taskInit (SINGLE := TRUE, PRIORITY := 1);
    TASK taskSync (INTERVAL := t#5s, PRIORITY := 2);
    PROGRAM pgm1Init WITH taskInit : Prog1Init();
    PROGRAM WITH taskSync : Prog1Sync();
  END_RESOURCE
END_CONFIGURATION

PROGRAM Prog1Init
  gWait := FALSE; (* Initialize communication status *)
  (* Communication settings: SET, MODBUS RTU master mode, slave ID is 1, 9600/N/8/1 *)
  extLine1.gConf(GET_SET := 1, MODE := 1, SLAVE_ID := 1, RATE := 9600, PARITY := 0,
DATABITS := 8, STOPBIT := 1);
  (* Initialize the work *)
  gSar := 0;
  gSad := 0;
  gRw := 1;
  extLine1.gComm.EC := 0; (* Initialize communication result *)
  extLine1.gComm.SLAVE_ID := 11;
  extLine1.gComm.DATA_COUNT := 1;
  extLine1.gComm.MEM_ADDR := ADDROF(gWord0);

  (* Initialize the command structure *)
  gCmd[0,0,0].sar := 1; (* coil status, begin of range, read *)
  gCmd[0,0,0].sad := 0;
  gCmd[0,0,0].rw := 0;
  gCmd[0,0,1].sar := 1; (* coil status, begin of range, write *)
  gCmd[0,0,1].sad := 0;
  gCmd[0,0,1].rw := 1;
  gCmd[0,1,0].sar := 1; (* coil status, end of range, read *)
  gCmd[0,1,0].sad := 16383;
  gCmd[0,1,0].rw := 0;
  gCmd[0,1,1].sar := 1; (* coil status, end of range, write *)
  gCmd[0,1,1].sad := 16383;
  gCmd[0,1,1].rw := 1;
  gCmd[0,2,0].sar := 1; (* coil status, out of range, read *)
  gCmd[0,2,0].sad := 16384;
  gCmd[0,2,0].rw := 0;
```

```
gCmd[0,2,1].sar := 1; (* coil status, out of range, write *)
gCmd[0,2,1].sad := 16384;
gCmd[0,2,1].rw := 1;
gCmd[1,0,0].sar := 2; (* input status, begin of range, read *)
gCmd[1,0,0].sad := 0;
gCmd[1,0,0].rw := 0;
gCmd[1,0,1].sar := 2; (* input status, begin of range, read *)
gCmd[1,0,1].sad := 1;
gCmd[1,0,1].rw := 0;
gCmd[1,1,0].sar := 2; (* input status, end of range, read *)
gCmd[1,1,0].sad := 16383;
gCmd[1,1,0].rw := 0;
gCmd[1,1,1].sar := 2; (* input status, end of range, read *)
gCmd[1,1,1].sad := 16382;
gCmd[1,1,1].rw := 0;
gCmd[1,2,0].sar := 2; (* input status, out of range, read *)
gCmd[1,2,0].sad := 16384;
gCmd[1,2,0].rw := 0;
gCmd[1,2,1].sar := 2; (* input status, out of range, read *)
gCmd[1,2,1].sad := 16385;
gCmd[1,2,1].rw := 0;
gCmd[2,0,0].sar := 3; (* holding register, begin of range, read *)
gCmd[2,0,0].sad := 0;
gCmd[2,0,0].rw := 0;
gCmd[2,0,1].sar := 3; (* holding register, begin of range, write *)
gCmd[2,0,1].sad := 0;
gCmd[2,0,1].rw := 1;
gCmd[2,1,0].sar := 3; (* holding register, end of range, read *)
gCmd[2,1,0].sad := 2047;
gCmd[2,1,0].rw := 0;
gCmd[2,1,1].sar := 3; (* holding register, end of range, write *)
gCmd[2,1,1].sad := 2047;
gCmd[2,1,1].rw := 1;
gCmd[2,2,0].sar := 3; (* holding register, out of range, read *)
gCmd[2,2,0].sad := 2048;
gCmd[2,2,0].rw := 0;
gCmd[2,2,1].sar := 3; (* holding register, out of range, write *)
gCmd[2,2,1].sad := 2048;
gCmd[2,2,1].rw := 1;
gCmd[3,0,0].sar := 4; (* input register, begin of range, read *)
gCmd[3,0,0].sad := 0;
gCmd[3,0,0].rw := 0;
```

```
   gCmd[3,0,1].sar := 4; (* input register, end of range, read *)
   gCmd[3,0,1].sad := 1023;
   gCmd[3,0,1].rw := 0;
   gCmd[3,1,0].sar := 4; (* input register, out of range, read *)
   gCmd[3,1,0].sad := 1024;
   gCmd[3,1,0].rw := 0;
   gCmd[3,1,1].sar := 4; (* input register, design year, read *)
   gCmd[3,1,1].sad := 9900;
   gCmd[3,1,1].rw := 0;
   gCmd[3,2,0].sar := 4; (* input register, MAC 5~6, read *)
   gCmd[3,2,0].sad := 9912;
   gCmd[3,2,0].rw := 0;
   gCmd[3,2,1].sar := 4; (* input register, lot, read *)
   gCmd[3,2,1].sad := 9991;
   gCmd[3,2,1].rw := 0;
   gCmd[4,0,0].sar := 0; (* general register, begin of range, read *)
   gCmd[4,0,0].sad := 0;
   gCmd[4,0,0].rw := 0;
   gCmd[4,0,1].sar := 0; (* general register, begin of range, write *)
   gCmd[4,0,1].sad := 0;
   gCmd[4,0,1].rw := 1;
   gCmd[4,1,0].sar := 0; (* general register, end of range, read *)
   gCmd[4,1,0].sad := 8191;
   gCmd[4,1,0].rw := 0;
   gCmd[4,1,1].sar := 0; (* general register, end of range, write *)
   gCmd[4,1,1].sad := 8191;
   gCmd[4,1,1].rw := 1;
   gCmd[4,2,0].sar := 0; (* general register, out of range, read *)
   gCmd[4,2,0].sad := 8192;
   gCmd[4,2,0].rw := 0;
   gCmd[4,2,1].sar := 0; (* general register, out of range, write *)
   gCmd[4,2,1].sad := 8192;
   gCmd[4,2,1].rw := 1;
END_PROGRAM

PROGRAM Prog1Sync
   VAR
      p1sDt : DT;
   END_VAR

   IF gWait = TRUE & extLine1.gComm.EC = 1 THEN
      (* Wait for communication result. *)
```

```
    RETURN;
  END_IF;

  IF gWait = FALSE THEN
    (* Communication is terminated *)
    extLine1.gComm.SLAVE_AREA := gCmd[gSar, gSad, gRw].sar;
    extLine1.gComm.SLAVE_ADDR := gCmd[gSar, gSad, gRw].sad;
    extLine1.gComm(READ_WRITE := gCmd[gSar, gSad, gRw].rw);
    gWait := TRUE; (* Switching to waiting state for communication result. *)
    RETURN;
  ELSE
    (* Waiting for communication result *)
    IF extLine1.gComm.EC = 0 THEN (* Communication ended normally. *)
      gRtc(EN := 0);
      p1sDt := gRtc.CDT; (* Save the communication normal completion time *)
    END_IF;
    gWait := FALSE; (* Switching to communication end state. *)

    gWord1 := gWord0 + 1;

    (* Command automatic switching *)
    gRw := gRw + 1;
    IF gRw >= 2 THEN
      gRw := 0;
      gSad := gSad + 1;
      IF gSad >= 3 THEN
        gSad := 0;
        gSar := gSar + 1;
        IF gSar >= 5 THEN
          gSar := 0;
        END_IF;
      END_IF;
    END_IF;

    (* In case the next operation is write, the value to be written is designated as 'current
value + 1'.. *)
    gWord0 := gWord1;
  END_IF;
END_PROGRAM
```

[DST 4] rtu1.dst

An example of using MODBUS TCP communication is written as tcp0.dst(server) and tcp1.dst(client). Two E5As are required for testing. Two E5As are connected via Wifi. For communication setting, operate one of E5A in AP mode or connect to a wireless router as stations.

E5A acting as a Server changes some of its internal memory and output memory to writable, and waits. It works on RESOURCE ETH_1. TCP port opens 502.

E5A acting as master assigns 4 addresses in one area of slave E5A and tries one by one in order. It works on RESOURCE ETH_2. The Server's IP address is 192.168.0.21.

```
CONFIGURATION nameOfConf
  RESOURCE extLine1 ON ETH_1
    VAR_GLOBAL
      gConf : CONF_ETH1;
    END_VAR

    TASK taskInit (SINGLE := TRUE, PRIORITY := 1);
    TASK taskSync (INTERVAL := t#5s, PRIORITY := 2);
    PROGRAM pgm1Init WITH taskInit : Prog1Init();
    PROGRAM WITH taskSync : Prog1Sync();
  END_RESOURCE
END_CONFIGURATION

PROGRAM Prog1Init
  VAR
    loBool : BOOL;
  END_VAR

  (* Communication settings: SET, MODBUS TCP server mode, IPv4, port 502 *)
  extLine1.gConf(GET_SET := 1, MODE := 1, IPV4_6 := 0, PORT := 502);

  (* Disable write protection for the workspace *)
  loBool := WA_ENABLE_GEN(0, 16);
  loBool := WA_ENABLE_GEN(2048 * 16, 16);
  loBool := WA_ENABLE_GEN(4095 * 16, 16);

  loBool := WA_ENABLE_OUT(0, 1);
  loBool := WA_ENABLE_OUT(16384, 1);
  loBool := WA_ENABLE_OUT(32767, 1);
  loBool := WA_ENABLE_OUT(32768, 1);

  loBool := WA_ENABLE_OUT(16383, 2);
  IF loBool = FALSE THEN
```

```
    loBool := WA_ENABLE_OUT(16383, 1);
  END_IF;
  loBool := WA_ENABLE_OUT(16384, 2);
  loBool := WA_ENABLE_OUT(32766, 2);
  loBool := WA_ENABLE_OUT(32767, 2);

  loBool := WA_ENABLE_OUT(0, 16);
  loBool := WA_ENABLE_OUT(1024 * 16, 16);
  loBool := WA_ENABLE_OUT(2047 * 16, 16);
  loBool := WA_ENABLE_OUT(4095 * 16, 16);

  loBool := WA_ENABLE_OUT(1023 * 16, 2 * 16);
  IF loBool = FALSE THEN
    loBool := WA_ENABLE_OUT(1023 * 16, 16);
  END_IF;
  loBool := WA_ENABLE_OUT(1024 * 16, 2 * 16);
  loBool := WA_ENABLE_OUT(2046 * 16, 2 * 16);
  loBool := WA_ENABLE_OUT(2047 * 16, 2 * 16);
END_PROGRAM

PROGRAM Prog1Sync
  VAR
    vDw : DWORD;
  END_VAR

  vDw := 0;
END_PROGRAM
```

[DST 5] tcp0.dst

```
CONFIGURATION nameOfConf
  TYPE T_CMD:
    STRUCT
      rw : BOOL; (* read/write *)
      sad : UINT; (* address *)
    END_STRUCT
  END_TYPE

  VAR_GLOBAL
    gWait : BOOL;
    gTest, gCntTest : SINT;
    gWord0, gWord1 : WORD;
    gRtc : RTC;
    gCmd : ARRAY[4] OF T_CMD;
  END_VAR

  RESOURCE extLine1 ON ETH_2
    VAR_GLOBAL
      gConf : CONF_ETH2;
      gComm : COMM_ETH2;
    END_VAR

    TASK taskInit (SINGLE := TRUE, PRIORITY := 1);
    TASK taskSync (INTERVAL := t#5s, PRIORITY := 2);
    PROGRAM pgm1Init WITH taskInit : Prog1Init();
    PROGRAM WITH taskSync : Prog1Sync();
  END_RESOURCE
END_CONFIGURATION

PROGRAM Prog1Init
  gWait := FALSE; (* Initialize communication status *)
  (* Communication settings: SET, MODBUS TCP client mode, unit ID is 1, IPv4, server address is 192.168.0.21:502 *)
  extLine1.gConf(GET_SET := 1, MODE := 1, UNIT_ID := 1, IPV4_6 := 0, HOST := '192.168.0.21', PORT := 502);
  (* Initialize the work *)
  gTest := 0;
  gCntTest := 4;
  extLine1.gComm.EC := 0; (* Initialize communication result *)
  (* Area 0=general reference, 1=coil status, 2=input status, 3=holding register, 4=input register *)
  extLine1.gComm.SERVER_AREA := 0;
  extLine1.gComm.DATA_COUNT := 1;
  extLine1.gComm.MEM_ADDR := ADDROF(gWord0);

  (* Initialize the command structure *)
  (* R/W 0=read, 1=write *)
  gCmd[0].rw := 1; (* step 0 *)
  gCmd[0].sad := 0;
  gCmd[1].rw := 1; (* step 1 *)
  gCmd[1].sad := 2048;
  gCmd[2].rw := 1; (* step 2 *)
  gCmd[2].sad := 4095;
  gCmd[3].rw := 1; (* step 3 *)
  gCmd[3].sad := 4096;
```

```
END_PROGRAM

PROGRAM Prog1Sync
  VAR
    p1sDt : DT;
    p1sDw : DWORD;
  END_VAR

  IF gWait = TRUE & extLine1.gComm.EC = 1 THEN (* Waiting for communication result *)
    RETURN;
  END_IF;

  IF gWait = FALSE THEN
    (* Communication is terminated *)
    extLine1.gComm.SERVER_ADDR := gCmd[gTest].sad;
    extLine1.gComm(READ_WRITE := gCmd[gTest].rw);
    gWait := TRUE; (* Switching to waiting state for communication result. *)
    RETURN;
  ELSE
    (* Communication result waiting state *)
    IF extLine1.gComm.EC = 0 THEN (* Communication ended normally. *)
      gRtc(EN := 0);
      p1sDt := gRtc.CDT; (* Save the communication normal completion time *)
    END_IF;
    gWait := FALSE; (* Switching to communication end state. *)

    gWord1 := gWord0 + 1;

    (* Command automatic switching *)
    gTest := gTest + 1;
    IF gTest >= gCntTest THEN
      gTest := 0;
    END_IF;

    (* In case the next operation is write, the value to be written is designated as 'current value + 1'. *)
    gWord0 := gWord1;
  END_IF;
END_PROGRAM
```

[DST 6] tcp1.dst