SYS TEC-specific Extensions for OpenPCS / IEC 61131-3

User Manual

Version 4.0

Edition July 2011

Document no.: L-1054d_04

SYSTEC electronic GmbH   August-Bebel-Str. 29   D-07973 Greiz
Phone: +49 (3661) 6279-0   Fax: +49 (3661) 6279-99
Web: http://www.systec-electronic.com   Email: info@systec-electronic.com

Status/Changes

Status: Released

Date/Version	Section	Change	By
2004/11/17 1.0	all	Creation	R. Sieber
2007/03/02 2.0	3	Section 3 new, therefore all the following sections have been shifted	R. Sieber
2007/07/18 3.0	1.4	Section 1.4 new	R. Sieber
2009/02/10 3.1	5.4	Description of return value rectified	R. Sieber
2011/07/15 4.0	2.2.6, 2.2.7, 3.12, 3.13, 4.5, 5.8, 5.9	Description of all FB/FUN with Parameter of Type POINTER new	R. Sieber

Inform yourselves:

Contact	**Direct**	**Your local distributor**
Address:	SYS TEC electronic GmbH August-Bebel-Str. 29 D-07973 Greiz GERMANY	Please find a list of our distributors under: http://www.systec-electronic.com/distributors
Ordering Information:	+49 (0) 36 61 / 62 79-0 info@systec-electronic.com	
Technical Support:	+49 (0) 36 61 / 62 79-0 support@systec-electronic.com	
Fax:	+49 (0) 36 61 / 6 79 99	
Web Site:	http://www.systec-electronic.com	

4th Edition July 2011

Table of Contents

List of Tables

List of Illustrations

1 Event Tasks

1.1 Application of Event Tasks

PLC programs which are only executed in case of a certain event (aka "interrupts") are called event tasks. For example, starts and stops of a PLC or a run-time error during program execution (division by zero or access to data field elements outside the defined field boundaries).

The start task is responsible for the one-time configuration and initialization of the control or system components. This includes, for example, the parameterization of decentralized field nodes at the start of program execution (e.g. parameterization of CANopen field bus devices via corresponding SDO accesses to the objects' device directories). Additionally, the stop task enables defined deactivation of the field nodes at the end of PLC program execution. In case of an error it is possible to set the local PLC outputs as well as the field node outputs to an uncritical state via the error task.

Start Task: Execution of the start task occurs during the stop to run state change of the PLC. This can be triggered on the hardware-side by switching the RUN/STOP switch to RUN and on the software-side by pressing a start button in the OpenPCS programming environment. The actual main PLC program is not executed until the start task has been fully executed.

Stop Task: Execution of the stop task occurs during the run to stop state change of the PLC. This can be triggered on the hardware-side by switching the RUN/STOP switch to STOP and on the software-side by pressing the stop button in the OpenPCS programming environment. The stop task is executed after termination of the actual main PLC program. Only then is the PLC in stop state.

Error Task: Error task execution is coupled to the occurrence of various error states (e.g. division by zero) which can occur during PLC program execution. Similar to the stop task, error task execution occurs after termination of the actual main PLC program. Only then is the PLC is in stop state.

1.2 Creation and Configuration of Event Tasks

In OpenPCS event tasks are only PLC programs with specific properties. Therefore, an event task is only created similar to "normal" programs via the menu item "File ➲ New ➲ Program". The entry "interrupt" has to be selected in the select field "task type" during assignment of the task to the resource (see Figure 1).

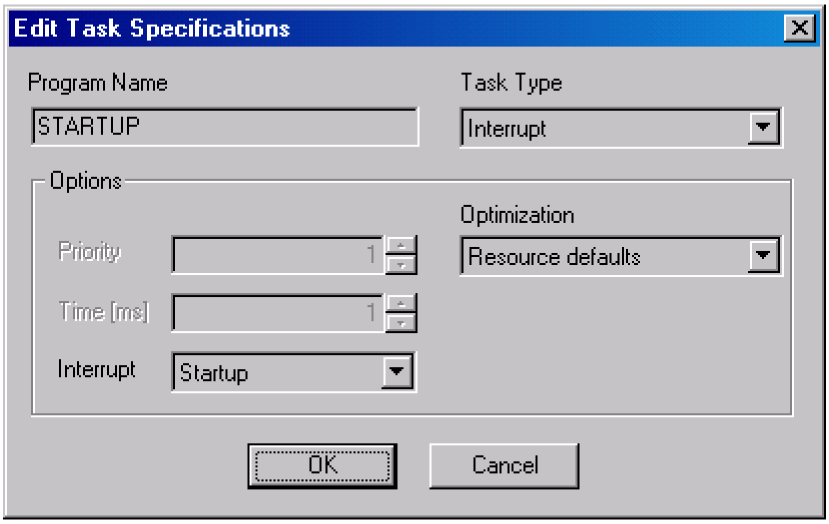

Figure 1: Dialog box "Processing Task Properties"

As standard, an event task is executed once when the allocated event occurs. The function block *ETRC,* described in section 1.3, enables an expansion of program execution to numerous successive cycles.

1.3 Function Block ETRC

As standard, an event task is executed once when the allocated event occurs. However, it may be necessary, especially when using decentralized field nodes, to expand the execution of an event task to numerous successive cycles. For example, the SDO accesses necessary for parameterization of CANopen field bus devices require the continuous calling of the SDO block for several PLC program cycles until successful completion.

Via the firmware function block *ETRC* (**E**vent **T**ask **R**un **C**ontrol), an event task can expand its own execution by a further program cycle. Information available at the block's outputs about the previous runtime and the number of the executed cycles can be used as a stop criterion to avoid getting caught in an infinite loop during event task execution in case of an error.

<u>Prototype of the Function Block</u>

```
             +----------+
             |   ETRC   |
             |          |
     BOOL ---|IN       Q|--- BOOL
             |          |
             |       EVC|--- USINT
             |       ERT|--- TIME
             |       CCV|--- UDINT
             |          |
             |     ERROR|--- USINT
             |          |
             +----------+
```

Definition of Operands

IN: TRUE: The event task requests execution for a further cycle.
FALSE: The event task intends to terminate its execution, or only asks for current status information without simultaneously requesting expansion for a further cycle.

Q: TRUE: The event task is processed by the runtime system for a further cycle.
FALSE: Execution of the event task is terminated after the current cycle.

EVC: The event code describes the internal system reason for the event task call. The event codes are defined in Table 1.

ERT: The elapsed run time states the event task time which has already been executed.

CCV: The cycle counter value defines the number of event task cycles which have already been executed.

ERROR: The error code provides information about the execution result of the function block. Possible error codes are defined in Table 2.

Table 1 Event Codes of the Function Block ETRC

Event Code	Event for the Task Call
0	Called task is unknown
1	PLC cold start executed
2	PLC warm start executed
3	PLC hot start executed
4	Single cycle start executed
5	PLC has been switched to STOP via the RUN/STOP switch
6	PLC has been switched to STOP on the software-side
7	PLC changes to STOP after executing a single cycle
8	General error during PLC program execution (e.g. invalid program code)
9	Division by zero
10	Access to an invalid data field index (ARRAY)
11	Error during the execution of a function block

Table 2 Error Codes of the Function Block ETRC

Error Code	Definition
0	The function block has been successfully executed
1	The function block has been called by an invalid event task. Therefore, execution of the function block is not possible.

Description

As standard an event task is only called for one single cycle. If the event task requires further cycles for its execution, it has to register this via the function block *ETRC*. The function block *ETRC* simultaneously states the reason for the event task call at the *EVC* output (see Table 1). Additionally, output *ERT* (Elapsed Run Time) and output C*CV* (Cycle Counter Value) state the elapsed event task runtime in milliseconds and the number of cycles during which the event task has already been executed respectively. Via the *ERT* and *CCV* information the event task can decide whether to execute a further cycle or not. Possible errors during function block execution are displayed at output *ERROR* and described in Table 2.

In case a runtime error such as Division by Zero occurred, the user can restart the nominal PLC program execution by using the function block *PTRC* (**P**rogram **T**ask **R**un **C**ontrol) described in Section 1.4.

The following sample program shows a simple start task which is processed for a total of 4 cycles. In order to achieve this, the task requests execution of a further cycle 3 times by calling the function block *ETRC*.

<u>Sample Program</u>

```
PROGRAM Startup

VAR
    Out8_15 AT %QB1.0 : BYTE;
    RunState : BOOL;
    EventCode : USINT;
    RunTime : TIME;
    CycleCounter : UDINT;
    Error : USINT;

        FB_ETRC : ETRC;
END_VAR


(* get the current state only, but don't request execution time for *)
(* the next cycle yet *)
CAL     FB_ETRC (
        IN := FALSE
        |
        RunState := Q,
        EventCode := EVC,
        RunTime := ERT,
        CycleCounter := CCV,
        Error := ERROR)

LD      CycleCounter
UDINT_TO_BYTE
ST      Out8_15

(* for 1.-3. cycle request execution time for the next cycle *)
LD      CycleCounter
LE      3
CALC    FB_ETRC (IN := TRUE)

RET

END_PROGRAM
```

1.4 Function Block PTRC

The *PTRC* (**P**rogram **T**ask **R**un **C**ontrol) function block provides methods to stop and restart the program execution from within the PLC program.

Prototype of the Function block

```
                +------------+
                |    PTRC    |
                |            |
      USINT  ---|MODE   ERROR|--- USINT
                |            |
                +------------+
```

Definition of Operand

MODE: Command to be executed, start mode of function block, see Table 3

ERROR: The error code states information about the execution result of the function block. Possible error codes are defined in Table 4.

Table 3: Start mode of PTRC function block

Start mode	Meaning
0	Stop execution of PLC program
1	Start execution of PLC program; Coldstart
2	Start execution of PLC program; Warmstart
3	Start execution of PLC program; Hotstart

Table 4: Error codes of the PTRC function block

Error code	Meaning
0	The function block has been successfully executed
4	Invalid mode (*MODE*) passed when calling the function block

Description

Using the *PTRC* function block it is possible to stop or restart the execution of a PLC program from within the PLC program and enable an automatic restart in case a runtime error (i.e. division by zero) occurred and thus, allows for a continuous operation without user-interaction. Normally this function block is called from within an Error Task (see also Section 1.1). The supported modes are listed in Table 3. Possible errors during function block execution are displayed at output *ERROR* and described in Table 4.

The following example shows the application of the *PTRC* function block within an user-specific Error Task (here the program is called "Resume" as used in Section 1.2).

Sample Program

```
PROGRAM Resume

VAR CONSTANT
    PTRC_MODE_STOP       : USINT    := 0;
    PTRC_MODE_COLDSTART  : USINT    := 1;
    PTRC_MODE_WARMSTART  : USINT    := 2;
    PTRC_MODE_HOTSTART   : USINT    := 3;
END_VAR

VAR
    FB_PTRC  : PTRC;
    usiError : USINT;
END_VAR

FB_PTRC (MODE := PTRC_MODE_COLDSTART | usiError := ERROR);

RETURN;

END_PROGRAM
```

2 String Processing

2.1 Standard String Functions according to IEC 61131-3

The string functions listed below are standard functions according to IEC 61131-3 and are described in detail in the OpenPCS online help. This list provides an overview of all the string functions available on SYSTEC controls.

2.1.1 Function LEN

The function *LEN* determines the length of a character string.

```
              +---------------+
              |      LEN      |
              |               |
STRING  ---|IN           OUT|--- INT
              |               |
              +---------------+
```

Description

This function determines the length of the character string IN.

Example

A := LEN('ABCDEF'); *(* Result: A := 6 *)*

2.1.2 Function LEFT

The function *LEFT* determines the left part of a character string.

```
              +---------------+
              |     LEFT      |
              |               |
STRING  ---|IN           OUT|--- STRING
ANY_INT ---|L                |
              |               |
              +---------------+
```

Description

This function determines the left part with the length L of the character string IN.

Sample Program

A := LEFT(IN:='ABCDEF', L:=3); *(* Result: A := 'ABC' *)*

2.1.3 Function RIGHT

The function *RIGHT* determines the right part of a character string.

```
           +---------------+
           |     RIGHT     |
           |               |
STRING  ---|IN          OUT|--- STRING
ANY_INT ---|L              |
           |               |
           +---------------+
```

Description

This function determines the right part with the length L of the character string IN.

Sample Program

```
A := RIGHT(IN:='ABCDEF', L:=3);              (* Result: A := 'DEF' *)
```

2.1.4 Function MID

The function *MID* determines the mid part of a character string.

```
           +---------------+
           |      MID      |
           |               |
STRING  ---|IN          OUT|--- STRING
ANY_INT ---|L              |
ANY_INT ---|P              |
           |               |
           +---------------+
```

Description

This function determines the mid part with the length L of the character string IN, starting at position P.

Sample Program

```
A := MID(IN:='ABCDEF', L:=3, P:=2);          (* Result: A := 'BCD' *)
```

2.1.5 Function CONCAT

The function *CONCAT* concatenates character strings.

```
               +---------------+
               |    CONCAT     |
               |               |
      STRING  ---|IN1         OUT|--- STRING
      STRING  ---|IN2            |
               |               |
               +---------------+
```

Description

This function determines the total character string concatenated from character strings IN1 and IN2.

Sample Program

```
A := CONCAT(IN1:='ABC', IN2:='xyz');         (* Result: A := 'ABCxyz' *)
```

2.1.6 Function INSERT

The function *INSERT* inserts a character string into another character string.

```
               +---------------+
               |    INSERT     |
               |               |
      STRING  ---|IN1         OUT|--- STRING
      STRING  ---|IN2            |
      ANY_INT ---|P              |
               |               |
               +---------------+
```

Description

This function inserts character string IN2 into character string IN1 after position P.

Sample Program

```
A := INSERT(IN1:='ABC', IN2:='xyz', P:=2);  (* Result: A := 'ABxyzC' *)
```

2.1.7 Function DELETE

The function *DELETE* deletes characters from a character string.

```
               +---------------+
               |    DELETE     |
               |               |
     STRING  ---|IN          OUT|--- STRING
     ANY_INT ---|L              |
     ANY_INT ---|P              |
               |               |
               +---------------+
```

Description

This function deletes L characters from character string IN, starting at position P.

Sample Program

```
A := DELETE(IN:='ABCDEF', L:=3, P:=2);      (* Result: A := 'AEF' *)
```

2.1.8 Function REPLACE

The function *REPLACE* replaces parts of a character string.

```
               +---------------+
               |    REPLACE    |
               |               |
     STRING  ---|IN1         OUT|--- STRING
     STRING  ---|IN2            |
     ANY_INT ---|L              |
     ANY_INT ---|P              |
               |               |
               +---------------+
```

Description

This function replaces L characters of character string IN1 with character string IN2, starting at position P.

Sample Program

```
A := REPLACE(IN1:='ABCDEF', IN2:='z',       (* Result: A := 'AzEF' *)
             L:=3, P:=2);
```

2.1.9 Function FIND

The function *FIND* finds a character string.

```
               +--------------+
               |     FIND     |
               |              |
     STRING ---|IN1        OUT|--- INT
     STRING ---|IN2           |
               |              |
               +--------------+
```

Description

This function determines the start position for the first appearance of character string IN2 in character string IN1. If character string IN2 is not contained in character string IN1, the function results in the value 0.

Sample Program

```
A := FIND(IN1:='ABCBCF', IN2:='BC');        (* Result: A := 2 *)
```

2.2 SYSTEC-Specific String Functions and Function Blocks

2.2.1 Function Block GETSTRINFO

The function block GETSTRINFO retrieves specific string information.

Prototype of the Function Block

```
               +---------------+
               |   GETSTRINFO  |
               |               |
     STRING ---|ISTR--------ISTR|--- STRING
               |               |
               |           SIZE|--- INT
               |           USED|--- INT
               |           FREE|--- INT
               |               |
               +---------------+
```

Definition of Operands

ISTR	String whose properties are to be determined
SIZE	Maximum string length (internal size of the available buffer for this string variable)
USED	Occupied string length (same as IEC 61131-3 standard function *LEN*, see section 2.1.1)
FREE	Unoccupied/Unused string length (same as *SIZE - USED*)

Description

This function block determines the size of the available internal buffer of a specified string (maximum string length) as well as the occupied and unoccupied string length. This block is especially important in connection with other function blocks which are used to read out or receive character strings, e.g. *NVDATA_STR* (see section 4.4), *SIO_READ_STR* (see section 5.6) or *CAN_SDO_READ_STR*.

Sample Program

```
VAR
    strText  : STRING(16) := 'ABCDEFGHIJ';
    iStrSize : INT;
    iStrUsed : INT;
    iStrFree : INT;
    FB_GetStrInfo : GETSTRINFO;
END_VAR

CAL     FB_GetStrInfo (
        ISTR := strText
        |
        iStrSize := SIZE,            (* iStrSize := 16 *)
        iStrUsed := USED,            (* iStrUsed := 10 *)
        iStrFree := FREE)            (* iStrFree := 6  *)

...

RET
```

2.2.2 Function CHR

The function *CHR* changes a numerical character code into the respective ASCII character.

Prototype of the Function

```
                +---------------+
                |      CHR      |
                |               |
     USINT   ---|CHRCODE  CHRSTR|--- STRING
                |               |
                +---------------+
```

Definition of Operands

CHRCODE Numerical character code to be changed into an ASCII character

CHRSTR String with ASCII character which corresponds to the numerical character code

Description

This function returns a string to output *CHRSTR* whose only character corresponds to the character code passed at input *CHRCODE*.

Sample Program

```
CHR(16#41)       (* Result: 'A' *)
CHR(97)          (* Result: 'a' *)
CHR(60)          (* Result: '<' *)
CHR(36)          (* Result: '$' *)
```

2.2.3 Function ASC

The function *ASC* changes an ASCII character into the corresponding numerical character code.

Prototype of the Function

```
                 +---------------+
                 |      ASC      |
                 |               |
      STRING  ---|CHRSTR  CHRCODE|--- USINT
                 |               |
                 +---------------+
```

Definition of Operands

CHRSTR	String whose first character is used to determine the numerical character code
CHRCODE	Numerical character code of the first ASCII character in the string

Description

This function returns the numerical character code of the first character of the string passed at input *CHRSTR* to output *CHRCODE*.

Sample Program

```
ASC('A')        (* Result: 65 / 16#41 *)
ASC('a')        (* Result: 97 / 16#61 *)
ASC('ABC')      (* Result: 65 / 16#41 *)
ASC(' 123')     (* Result: 32 / 16#20 *)
```

2.2.4 Function STR

The function *STR* changes a REAL value into a corresponding string.

Prototype of the Function

```
                 +---------------+
                 |      STR      |
                 |               |
      REAL    ---|VALNUM   VALSTR|--- STRING
                 |               |
                 +---------------+
```

Definition of Operands

VALNUM	Numerical REAL value to be changed into a string
VALSTR	String with a character string which corresponds to the numerical REAL value

Description

This function changes the numerical REAL value passed at input *VALNUM* into a corresponding string and returns it to output *VALSTR*. During conversion, a leading space is always reserved for the sign. No trailing zeros are given; the decimal point is also dropped for integers.

Sample Program

```
STR(123)           (* Result: ' 123'     *)
STR(123.45)        (* Result: ' 123.45'  *)
STR(-123.45)       (* Result: '-123.45'  *)
STR(98.7654)       (* Result: ' 98.7654' *)
```

2.2.5 Function VAL

The function *VAL* changes a string into a corresponding REAL value.

Prototype of the Function

```
               +---------------+
               |      VAL      |
               |               |
     STRING  ---|VALSTR   VALNUM|--- REAL
               |               |
               +---------------+
```

Definition of Operands

VALSTR String whose character string is to be changed into a numerical REAL value

VALNUM Numerical REAL value which corresponds to the passed character string

Description

This function changes the string passed at input *VALSTR* into a corresponding numerical REAL value and returns it to output *VALNUM*.

Sample Program

```
VAL('123')         (* Result: 123     *)
VAL('123.45')      (* Result: 123.45  *)
VAL('-123.45')     (* Result: -123.45 *)
VAL('98.7654')     (* Result: 98.7654 *)
```

2.2.6 Function BIN_TO_STR

The function *BIN_TO_STR* converts a numerical value into an appropriate string.

Prototype of the function

```
              +----------------------+
              |      BIN_TO_STR      |
              |                      |
    POINTER ---|PTR_DATA_IN   DATA_OUT|--- STRING
    STRING  ---|FORMAT                |
              |                      |
              +----------------------+
```

Definition of Operands

PTR_DATA_IN Address of an object, whose value has to convert into a string.

FORMAT String with specification of the output format, see Table 5

DATA_OUT formatted string according to the numerical input value

Description

The function converts a numerical object addressed via *PTR_DATA_IN* into an appropriate string, considering the given format specification. The format specification transferred at the input *FORMAT* defines the output format of the string returned as *DATA_OUT*, whose string conforms to the numerical input value. Table 5 describes possible format specifications.

Table 5: Format Specifications for BIN_TO_STR

Object Type	Format Specification	Description
BOOL	'd'	Numerical output { 0 | 1 }
	'b'	Literal output in small form letters { true | false }
	'B'	Literal output in capital letters { TRUE | FALSE }
BYTE, USINT, SINT WORD, UINT, INT DWORD, UDINT, DINT	'd'	Output in decimal notation, definition of minimal output of characters is possible (see text below)
	'x'	Output in hexadecimal notation with small form letters, definition of minimal output of characters is possible (see text below)
	'X'	Output in hexadecimal notation with capital letters, definition of minimal output of characters is possible (see text below)
REAL	'd' or 'f'	Output in decimal notation with decimal places; definition of minimal output of characters and decimal places is possible (see text below)

For numerical types of integers, the format specifications 'd', 'x' und 'X' can be extended optionally according to the definition of the minimal output of characters. This minimal number of characters has to be attached to the format specification (e.g. 'd4'). A '0' set before the minimal number of characters can effect that the output string is filled left-aligned with '0'-characters, if applicable, to achieve the demanded field width (e.g. 'd04'). Otherwise, the output string is filled left-aligned with spaces.

For objects of type REAL, format specifications 'd', and 'f' can be extended optionally through the definition of the minimal number of characters. It is thereby distinguished between the total number of characters and decimal places. Decimal places are to specify with '.y' (e.g. 'f.4' for 4 decimal places). Optionally, the minimal number of the whole output string can be defined in the form of 'x.y' left of the point, whereat the decimal point is included. Therefore, the format specification 'f9.4' for example causes the output of a string with 9 characters in total, of which one character is the decimal point itself followed by 4 decimal places, so that 4 pre-decimal places result (9 total – 1 decimal point – 4 decimal places = 4 integers). A '0' set before the total minimal number of characters ('x' in format 'x.y') effects that the output string is filled left-aligned with '0'-characters to achieve the demanded field width (e.g. 'f09.4'). Otherwise, the output string is filled left-aligned with spaces.

If a successful change is not possible due to invalid parameters, the output string *DATA_OUT* receives an appropriate error message in the plain text. (e.g. *'ERROR: data type not supported'* or *'ERROR: format type not supported'*).

Sample Program

```
VAR

    xBoolVar    : BOOL;
    iIntVar     : INT;
    rRealVar    : REAL;

    pVar        : POINTER;
    strResult   : STRING;

END_VAR

    xBoolVar := TRUE;
    pVar := &xBoolVar;
    strResult := BIN_TO_STR (pVar, 'd');        (* strResult: '1'         *)
    strResult := BIN_TO_STR (pVar, 'b');        (* strResult: 'true'      *)

    iIntVar := 123;
    pVar := &iIntVar;
    strResult := BIN_TO_STR (pVar, 'd');        (* strResult: '123'       *)
    strResult := BIN_TO_STR (pVar, 'x4');       (* strResult: '  7b'      *)
    strResult := BIN_TO_STR (pVar, 'X04');      (* strResult: '007B'      *)

    rRealVar := 123.456;
    pVar := &rRealVar;
    strResult := BIN_TO_STR (pVar, 'f.4');      (* strResult: '123.4560'  *)
    strResult := BIN_TO_STR (pVar, 'f09.4');    (* strResult: '0123.4560' *)
```

2.2.7 Function STR_TO_BIN

The function *STR_TO_BIN* converts a character string into an appropriate numerical value.

Prototype of the function

```
              +---------------------+
              |      STR_TO_BIN     |
              |                     |
    STRING  ---|DATA_IN          RES|--- USINT
    POINTER ---|PTR_DATA_OUT         |
    STRING  ---|FORMAT               |
              |                     |
              +---------------------+
```

Definition of Operands

DATA_IN	String, whose character sequence is to convert into a numerical value
FORMAT	String with specification of the input format, see Table 6
PTR_DATA_OUT	Address of an object, in which the changed numerical value is to store
RES	Information on the implementation result of the change, possible error codes are defined in Table 7.

<u>Description</u>

This function converts a string passed at input *DATA_IN* into an appropriate numerical value and stores it in the object addressed via *PTR_DATA_OUT.* The format specification passed at input *FORMAT* describes the input format of the string, which is to concert into a numerical value. Table 6 describes possible format specifications.

Table 6: Format specifications for STR_TO_BIN

<table>
<tr><th>Object-Type</th><th>Format Specification</th><th>Description</th></tr>
<tr><td rowspan="2">BOOL</td><td>'d'</td><td>Input string in numerical notation { 0 | 1 }</td></tr>
<tr><td>'b' or 'B'</td><td>Input string as literal, optional capital or small form letters,
{ true | false | TRUE | FALSE }</td></tr>
<tr><td rowspan="2">BYTE,
USINT,
SINT

WORD,
UINT,
INT

DWORD,
UDINT,
DINT</td><td>'d'</td><td>Input string in decimal notation</td></tr>
<tr><td>'x' or 'X'</td><td>Input string in hexadecimal notation, optional capital or small form letters</td></tr>
<tr><td>REAL</td><td>'d' or 'f'</td><td>Input string in decimal notation with decimal places</td></tr>
</table>

Table 7: Error-Codes of Function STR_TO_BIN

Error-Code	Meaning
0	No error has occurred during processing of the function block
1	Pointer refers to an object of unsupported data type
2	Invalid format specification (*FORMAT*) when selecting the function block
4	Invalid input string (*DATA_IN*) when selecting the function block

Sample Program

```
VAR

    xBoolVar    : BOOL;
    iIntVar     : INT;
    rRealVar    : REAL;

    pVar        : POINTER;
    usiRes      : USINT;

END_VAR

    pVar := &xBoolVar;
    usiRes := STR_TO_BIN ('1', pVar, 'd');
    usiRes := STR_TO_BIN ('true', pVar, 'b');

    pVar := &iIntVar;
    usiRes := STR_TO_BIN ('-123', pVar, 'd');
    usiRes := STR_TO_BIN ('ABCD', pVar, 'x');

    pVar := &rRealVar;
    usiRes := STR_TO_BIN ('123.456', pVar, 'd');
```

3 Data Communication via UDP

3.1 Data Communication Application via UDP

UDP (**U**ser **D**atagram **P**rotocol) is a minimal, connection-free and packet-oriented network protocol which belongs to the transport layer of the Internet protocol suite. Most systems with Ethernet interface used in the industrial sector support UDP. Therefore, this protocol can be recommended for the Ethernet-based data transfer between PLC and systems like terminals (HMI) or host computers.

Sending and receiving of UDP packets occurs via sockets. The function block *LAN_UDP_CREATE_SOCKET* is responsible for creating a local socket. The function block *LAN_UDP_SENDTO_STR* enables the sending of data packets and function block *LAN_UDP_RECVFROM_STR* enables the reception of data. A no longer required socket can be re-enabled via the function block *LAN_UDP_CLOSE_SOCKET*. Exiting the PLC program leads internally to an automatic shutdown of all occupied sockets.

3.2 Definitions for UDP Blocks

The following data types are globally defined in OpenPCS for the application via UDP blocks:

```
TYPE
    INETV4 : UDINT;
END_TYPE

TYPE
    SOCKID : UINT;
END_TYPE
```

Table 8: Error codes of the function blocks LAN_Xxx

Error Code	Meaning
0	No error occurred during function block execution
1	The specified network number (*NETNUMBER*) is not supported
2	An invalid parameter has been specified while calling the block
3	Error while initializing the UDP layer on the PLC
4	The UDP layer on the PLC reports an error while creating, sending or receiving a socket
5	No free socket available
6	The specified socket ID is invalid
7	The socket with the specified socket ID is not in use
8	The transferred send buffer is too big, the packet has been limited to the maximum possible number of data bytes
9	The transferred buffer is too small, no data has been copied
10	The specified host is unknown
11	Pointer references an object of an unsupported data type

3.3 Function Block LAN_GET_HOST_CONFIG

The function block *LAN_GET_HOST_CONFIG* is used to determine the local host configuration.

Prototype of the Function Block

```
              +---------------------------+
              |     LAN_GET_HOST_CONFIG    |
              |                           |
    STRING ---|HOST_NAME---------HOST_NAME|--- STRING
              |                  INET_ADDR|--- INETV4
              |              NUM_OF_SOCKETS|--- UINT
              |              SOCKETS_IN_USE|--- UINT
              |                           |
    BOOL   ---|ENABLE              CONFIRM|--- BOOL
              |                           |
    USINT  ---|NETNUMBER             ERROR|--- WORD
              |                  ERRORINFO|--- DWORD
              |                           |
              +---------------------------+
```

Definition of Operands

HOST_NAME	String variable for receiving the local host name of the PLC
INET_ADDR	Local IP address of the PLC
NUM_OF_SOCKETS	Number of maximum sockets which can be used for the PLC program
SOCKETS_IN_USE	Number of the sockets currently being used
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used to determine the local host configuration of the PLC.

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.4 Function LAN_ASCII_TO_INET

The function *LAN_ASCII_TO_INET* converts an IP address transferred as a string in default "." notation into the respective numeric presentation.

Prototype of the Function

```
              +---------------------------+
              |     LAN_ASCII_TO_INET     |
              |                           |
    STRING ---|IP_ADDR          INET_ADDR|--- INETV4
              |                           |
              +---------------------------+
```

Definition of Operands

IP_ADDR String with IP address in default "." notation (e.g. '192.168.1.20')

INET_ADDR Numeric presentation of the transferred IP address

Description

This function converts the string with the IP address in default "." notation (e.g. '192.168.1.20') transferred at input *IP_ADDR* into the respective numeric presentation and returns it at output *INET_ADDR*. The numeric format of the IP address is used by function blocks as, e.g., *LAN_GET_HOST_CONFIG*, *LAN_UDP_SENDTO_STR* or *LAN_UDP_RECVFROM_STR*.

The function *LAN_ASCII_TO_INET* complements function *LAN_INET_TO_ASCII* (see section 3.5).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.5 Function LAN_INET_TO_ASCII

The function *LAN_INET_TO_ASCII* converts an IP address transferred in a numeric presentation into the respective string with default "." notation.

Prototype of the Function

```
              +---------------------------+
              |     LAN_INET_TO_ASCII     |
              |                           |
    INETV4 ---|INET_ADDR          IP_ADDR|--- STRING
              |                           |
              +---------------------------+
```

Definition of Operands

INET_ADDR Numeric presentation of the IP address

IP_ADDR String with IP address in default "." notation (e.g. '192.168.1.20')

Description

This function converts the IP address transferred at input *INET_ADDR* in a numeric presentation into the respective string with default "." notation and returns it at output *IP_ADDR*. The numeric format of the IP address is used by function blocks as, e.g., *LAN_GET_HOST_CONFIG*, *LAN_UDP_SENDTO_STR* or *LAN_UDP_RECVFROM_STR*. This numeric presentation of the IP address can be converted into a presentable string via the function *LAN_INET_TO_ASCII*.

The function *LAN_INET_TO_ASCII* complements function *LAN_ASCII_TO_INET* (see section 3.4).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.6 Function LAN_GET_HOST_BY_NAME

The function *LAN_GET_HOST_BY_NAME* determines the IP address for the specified host name (only available on controls with DNS support).

Prototype of the Function

```
             +------------------------+
             |  LAN_GET_HOST_BY_NAME  |
             |                        |
STRING ---|HOST_NAME      INET_ADDR|--- INETV4
             |                        |
             +------------------------+
```

Definition of Operands

HOST_NAME	String with the name of the host to be searched for
INET_ADDR	Numeric presentation of the determined IP address

Description

This function determines the IP address for the host name specified at input *HOST_NAME* and returns it at output *INET_ADDR*. The determined IP address can, e.g., be used for calling function block *LAN_UDP_SENDTO_STR*.

Note: The function *LAN_GET_HOST_BY_NAME* is only available on controls with DNS support.

The function *LAN_GET_HOST_BY_NAME* complements function *LAN_GET_HOST_BY_ADDR* (see section 3.7).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.7 Function LAN_GET_HOST_BY_ADDR

The function *LAN_GET_HOST_BY_ADDR* determines the host name for the specified IP address (only available on controls with DNS support).

Prototype of the Function

```
             +------------------------+
             |  LAN_GET_HOST_BY_ADDR  |
             |                        |
INETV4 ---|INET_ADDR      HOST_NAME|--- STRING
             |                        |
             +------------------------+
```

Definition of Operands

INET_ADDR	Numeric presentation of the IP address to be resolved
HOST_NAME	String with the name of the determined host

Description

This function determines the respective host name for the numeric IP address transferred at input *INET_ADDR* and returns it as a string to output *HOST_NAME*. The function can, e.g., be used in connection with *LAN_UDP_RECVFROM_STR* to resolve the IP addresses returned by this function block as clear text names.

Note: The function *LAN_GET_HOST_BY_NAME* is only available on controls with DNS support.

The function *LAN_GET_HOST_BY_ADDR* complements function *LAN_GET_HOST_BY_NAME* (see section 3.6).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.8 Function Block LAN_UDP_CREATE_SOCKET

The function block *LAN_UDP_CREATE_SOCKET* creates a socket for sending or receiving data.

Prototype of the Function Block

```
             +----------------------------+
             |   LAN_UDP_CREATE_SOCKET    |
             |                            |
UINT   ---|PORT              SOCKET_ID|--- SOCKID
             |                            |
BOOL   ---|ENABLE              CONFIRM|--- BOOL
             |                            |
USINT  ---|NETNUMBER             ERROR|--- WORD
             |                 ERRORINFO|--- DWORD
             |                            |
             +----------------------------+
```

Definition of Operands

PORT	Port number to which the socket is to be bound (see text)
SOCKET_ID	Socket ID (an internal reference to the created socket assigned by the UDP layer of the PLC)
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped, since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block creates a socket for sending or receiving data. If the socket is intended for receiving data, a valid port number has to be specified at input *PORT*. In this case, the PLC internally calls the function *bind()* of the UDP layer and is thus capable of receiving data packets which are sent to its IP address with the specified port number. On most systems the use of port numbers smaller than 1024 is only permitted for privileged processes; the range from 1024 to 49151 is still reserved for default applications and administered by IANA (Internet Assigned Numbers Authority). If possible, port numbers from the private range from 49152 to 65535 should, therefore, preferably be used for the UDP communication with the PLC.

If the created socket should only be used for sending data, the specification of the port number is optional. If input *PORT* has been set to zero, the UDP layer of the PLC internally uses a free port number from the private range for sending. Calling of the internal function *bind()* within the UDP layer is then unnecessary. However, specification of a defined port number can also be necessary for sending, e.g., with an active firewall in the network which only forwards data to specific ports.

Upon its return the function block *LAN_UDP_CREATE_SOCKET* returns an internal reference assigned by the UDP layer of the PLC to the created socket at output *SOCKET_ID*. This socket ID has to be transferred when subsequently calling function blocks, e.g., *LAN_UDP_SENDTO_STR* or *LAN_UDP_RECVFROM_STR*.

A socket which is no longer required can be re-enabled by calling *LAN_UDP_CLOSE_SOCKET* (see section 0). Exiting the PLC program leads internally to an automatic shutdown of all occupied sockets.

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.9 Function Block LAN_UDP_CLOSE_SOCKET

The function block *LAN_UDP_CLOSE_SOCKET* is used to explicitly enable a socket which is no longer required.

Prototype of the Function Block

```
                +--------------------------+
                |    LAN_UDP_CLOSE_SOCKET   |
                |                          |
      SOCKID ---|SOCKET_ID                 |
                |                          |
      BOOL   ---|ENABLE             CONFIRM|--- BOOL
                |                          |
      USINT  ---|NETNUMBER            ERROR|--- WORD
                |                 ERRORINFO|--- DWORD
                |                          |
                +--------------------------+
```

Definition of Operands

SOCKET_ID	Socket ID of the socket to be enabled
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used to explicitly enable a socket which is no longer required. The *SOCKET_ID* is the internal reference to the respective socket returned by the UDP layer of the PLC while calling *LAN_UDP_CREATE_SOCKET* (see section 3.8). All the sockets which have not been explicitly enabled are automatically closed internally when exiting the PLC program.

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.10 Function Block LAN_UDP_RECVFROM_STR

The function block *LAN_UDP_RECVFROM_STR* is used to read UDP packets from the receive buffer of the UDP layer.

Prototype of the Function Block

```
               +--------------------------+
               |    LAN_UDP_RECVFROM_STR   |
               |                          |
      SOCKID ---|SOCKET_ID        PEER_ADDR|--- INETV4
               |                 PEER_PORT|--- UINT
               |                          |
      STRING ---|RXDATA--------------RXDATA|--- STRING
      INT    ---|MAXLENGTH         RXLENGTH|--- INT
               |                          |
      BOOL   ---|ENABLE             CONFIRM|--- BOOL
               |                          |
      USINT  ---|NETNUMBER            ERROR|--- WORD
               |                 ERRORINFO|--- DWORD
               |                          |
               +--------------------------+
```

Definition of Operands

SOCKET_ID	Socket ID of the socket to be polled
RXDATA	String variable for receiving the read characters
MAXLENGTH	Limitation of the number of characters to be read. If the number is 0, the buffer length of the transferred string is internally determined and used as the delimiter for the number of characters to be read (Note: the standard buffer size of a string in OpenPCS is 32 characters).
RXLENGTH	Length of the read character string
PEER_ADDR	Numeric presentation of the IP address of the opposite position from which the packet was received
PEER_PORT	Port number which was used by the opposite position to send the data
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used to read UDP packets from the receive buffer of the UDP layer. If output *CONFIRM* has been set to TRUE when the function block returns, the string variable specified as input/output parameter *RXDATA* contains the received string. Output *RXLENGTH* specifies the number of characters stored in *RXDATA*. If output *CONFIRM* has been set to FALSE when the block returns, no characters were received via the specified socket.

When receiving packets (*CONFIRM* set to TRUE) the outputs *PEER_ADDR* and *PEER_PORT* receive information about the IP address of the opposite position as well as the port number used for sending it. If the PLC should response to this received packet, the values from *PEER_ADDR* and *PEER_PORT* have to be used as destination specifications for the subsequent call of block *LAN_UDP_SENDTO_STR* (see section 3.11):

```
LAN_UDP_SENDTO_STR.PEER_ADDR := LAN_UDP_RECVFROM_STR.PEER_ADDR;
LAN_UDP_SENDTO_STR.PEER_PORT := LAN_UDP_RECVFROM_STR.PEER_PORT;
```

The socket to be used for receiving packets must have been created via the function block *LAN_UDP_CREATE_SOCKET* stating a valid port number prior to its use (see section 0).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.11 Function Block LAN_UDP_SENDTO_STR

The function block *LAN_UDP_SENDTO_STR* is used for sending UDP packets.

Prototype of the Function Block

```
                +---------------------------+
                |    LAN_UDP_SENDTO_STR     |
                |                           |
      SOCKID ---|SOCKET_ID                  |
      INETV4 ---|PEER_ADDR                  |
      UINT   ---|PEER_PORT                  |
                |                           |
      STRING ---|TXDATA--------------TXDATA|--- STRING
      INT    ---|TXLENGTH                   |
                |                           |
      BOOL   ---|ENABLE             CONFIRM|--- BOOL
                |                           |
      USINT  ---|NETNUMBER            ERROR|--- WORD
                |                 ERRORINFO|--- DWORD
                |                           |
                +---------------------------+
```

Definition of Operands

SOCKET_ID	Socket ID of the socket to be used for sending
PEER_ADDR	Numeric presentation of the IP address of the opposite position to which the packet is to be sent
PEER_PORT	Port number of the opposite position to which the packet is to be sent
TXDATA	String variable with the character string to be sent
TXLENGTH	Number of characters to be sent, if the number is 0, the length of the character string contained in the string *TXDATA* is internally determined (equals *LEN(TXDATA);*) and used as the number of characters to be sent.
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)

ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used for sending UDP packets. The inputs *PEER_ADDR* and *PEER_PORT* contain the address information of the opposite position to which the packet is to be sent. If the packet to be sent is a response to a message previously received with the function block *LAN_UDP_RECVFROM_STR*, the sender's address information has to be transferred here (see section 3.10).

The socket to be used for sending packets must have been created via the function block *LAN_UDP_CREATE_SOCKET* prior to its use (see section 3.8).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.12 Function Block LAN_UDP_RECVFROM_BIN

The function block *LAN_UDP_RECVFROM_BIN* is used to read UDP packets from the receive buffer of the UDP layer.

Prototype of the Function Block

```
                  +---------------------------+
                  |   LAN_UDP_RECVFROM_BIN    |
                  |                           |
      SOCKID   ---|SOCKET_ID        PEER_ADDR|--- INETV4
                  |                 PEER_PORT|--- UINT
                  |                           |
      POINTER  ---|PTR_RXDATA                 |
      INT      ---|MAXLENGTH         RXLENGTH|--- INT
                  |                           |
      BOOL     ---|ENABLE             CONFIRM|--- BOOL
                  |                           |
      USINT    ---|NETNUMBER            ERROR|--- WORD
                  |                 ERRORINFO|--- DWORD
                  |                           |
                  +---------------------------+
```

Definition of Operands

SOCKET_ID	Socket ID of the socket to be polled

PTR_RXDATA	Address of an object for receiving the read data bytes
MAXLENGTH	Limitation of number of bytes to read, if 0, the length of the object addressed by PTR_RXDATA is internally determined and used as the number of bytes to be read (there are max. read so much bytes as the object can take up)
RXLENGTH	Number of read data bytes
PEER_ADDR	Numeric presentation of the IP address of the opposite position from which the packet was received
PEER_PORT	Port number which was used by the opposite position to send the data
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used to read UDP packets from the receive buffer of the UDP layer. If output *CONFIRM* has been set to TRUE when the function block returns, then the object addressed by element *PTR_RXDATA* contains the received data bytes. Output *RXLENGTH* specifies the number of read data bytes. If output *CONFIRM* has been set to FALSE when the block returns, no data bytes were received via the specified socket.

When receiving packets (*CONFIRM* set to TRUE) the outputs *PEER_ADDR* and *PEER_PORT* receive information about the IP address of the opposite position as well as the port number used for sending it. If the PLC should response to this received packet, the values from *PEER_ADDR* and *PEER_PORT* have to be used as destination specifications for the subsequent call of block *LAN_UDP_SENDTO_BIN* (see section 3.13):

```
LAN_UDP_SENDTO_BIN.PEER_ADDR := LAN_UDP_RECVFROM_BIN.PEER_ADDR;
LAN_UDP_SENDTO_BIN.PEER_PORT := LAN_UDP_RECVFROM_BIN.PEER_PORT;
```

The socket to be used for receiving packets must have been created via the function block *LAN_UDP_CREATE_SOCKET* stating a valid port number prior to its use (see section 0).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.13 Function Block LAN_UDP_SENDTO_BIN

The function block *LAN_UDP_SENDTO_BIN* is used for sending UDP packets.

Prototype of the Function Block

```
                 +-------------------------+
                 |    LAN_UDP_SENDTO_BIN    |
                 |                          |
      SOCKID   ---|SOCKET_ID                 |
      INETV4   ---|PEER_ADDR                 |
      UINT     ---|PEER_PORT                 |
                 |                          |
      POINTER  ---|PTR_TXDATA                |
      INT      ---|TXLENGTH                  |
                 |                          |
      BOOL     ---|ENABLE             CONFIRM|--- BOOL
                 |                          |
      USINT    ---|NETNUMBER            ERROR|--- WORD
                 |                  ERRORINFO|--- DWORD
                 |                          |
                 +-------------------------+
```

Definition of Operands

SOCKET_ID	Socket ID of the socket to be used for sending
PEER_ADDR	Numeric presentation of the IP address of the opposite position to which the packet is to be sent
PEER_PORT	Port number of the opposite position to which the packet is to be sent
PTR_TXDATA	Address of an object with the binary data to be sent
TXLENGTH	Number of data bytes to be sent, if the number is 0, the length of the object addressed by *PTR_TXDATA* is internally determined and used as the number of characters to be sent
NETNUMBER	Network number (Note: If the PLC only supports one Ethernet interface, setting of this input can be skipped since numeric variables have already been preset with the initial value 0 according to IEC61131)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 8.
ERRORINFO	Reserved for additional error information
ENABLE	Input for enabling or disabling the FB
CONFIRM	Output for completed message via the FB

Description

The function block is used for sending UDP packets. The inputs *PEER_ADDR* and *PEER_PORT* contain the address information of the opposite position to which the packet is to be sent. If the packet to be sent is a response to a message previously received with the function block *LAN_UDP_RECVFROM_BIN*, the sender’s address information has to be transferred here (see section 3.12).

The socket to be used for sending packets must have been created via the function block *LAN_UDP_CREATE_SOCKET* prior to its use (see section 3.8).

Sample Program

A detailed sample program in section 3.14 displays the application of all the UDP blocks described in section 3.

3.14 Sample Program for applying UDP Function Blocks

The following sample program illustrates the application of all the UDP blocks described in section 3. The sample program realizes a simple server which accepts and executes commands in string format and returns a respective response string with the execution result to the client. Firstly, block *LAN_UDP_CREATE_SOCKET* is called to create a socket for exchanging data with the client. Secondly, the server remains in the subsequent program step until block *LAN_UDP_RECVFROM_STR* signals the reception of the command by a client. After interpreting and executing the command (in the user-specific function block "*ExecCommand*", not displayed here) the sample program returns the generated response string to the client via block *LAN_UDP_SENDTO_STR*. The IP address and port number which were received previously during command reception of *LAN_UDP_CREATE_SOCKET* are accepted as address information for calling *LAN_UDP_SENDTO_STR*.

Sample Program

```
PROGRAM UdpServer
VAR CONSTANT
    NETNUMBER       : USINT     := 0;
    SVRPORT         : UINT      := 55555;
    STOP_CMD        : STRING    := 'stop';
END_VAR

VAR
    xServerRunning  : BOOL;

    FB_ExecCommand  : ExecCommand;
    strRxCommand    : STRING(128);
    strTxResult     : STRING(250);

    FB_LanGetHostConfig : LAN_GET_HOST_CONFIG;
    strPlcHostName  : STRING(64);
    inetPlcIpAddr   : INETV4;
    uiNumOfSockets  : UINT;
    uiSocketsInUse  : UINT;
    strPlcIpAddr    : STRING;

    FB_LanUdpCreateSocket : LAN_UDP_CREATE_SOCKET;
    FB_LanUdpCloseSocket  : LAN_UDP_CLOSE_SOCKET;
    SocketID        : SOCKID;

    FB_LanUdpRecvfromStr : LAN_UDP_RECVFROM_STR;
    strRxData       : STRING(128);
    inetPeerIpAddr  : INETV4;
    uiPeerPort      : UINT;
    iRxDataLen      : INT;
    strRxDataLen    : STRING;
    strPeerIpAddr   : STRING;
    strPeerPort     : STRING;

    FB_LanUdpSendtoStr : LAN_UDP_SENDTO_STR;
    strTxData       : STRING(250);

    uiProcState     : UINT := 0;
END_VAR
```

```
(* ========== Program UdpServer ========== *)

CASE uiProcState OF

    (* ----- Inititialization ----- *)
    0:
        (*---------------------------------------------------*)
        (* The following block is not really necessary in this  *)
        (* application but it shows how to use some additional  *)
        (* LAN function(-blocks) which are maybe be helpful in  *)
        (* other projects.                                      *)
        FB_LanGetHostConfig (
            ENABLE    := TRUE,
            NETNUMBER := NETNUMBER,
            HOST_NAME := strPlcHostName
            |
            inetPlcIpAddr  := INET_ADDR,
            uiNumOfSockets := NUM_OF_SOCKETS,
            uiSocketsInUse := SOCKETS_IN_USE);

        strPlcIpAddr  := LAN_INET_TO_ASCII (inetPlcIpAddr);
        inetPlcIpAddr := LAN_ASCII_TO_INET (strPlcIpAddr);

        strPlcHostName := LAN_GET_HOST_BY_ADDR (inetPlcIpAddr);
        inetPlcIpAddr  := LAN_GET_HOST_BY_NAME (strPlcHostName);
        (*-------------------------------------------------*)

        (* ... continue with realy serious stuff for this application... *)
        FB_LanUdpCreateSocket (
            ENABLE := TRUE,
            NETNUMBER := NETNUMBER,
            PORT := SVRPORT
            |
            SocketID := SOCKET_ID);

        xServerRunning := TRUE;
        uiProcState := uiProcState + 1;     (* new state: Wait for Receipt *)



    (* ----- Wait for Receipt ----- *)
    1:
        (* Because this application acts as a server it is      *)
        (* necessary to save the output values PEER_ADDR and    *)
        (* PEER_PORT from the FB LAN_UDP_RECVFROM_STR. This     *)
        (* both parameters indentifies the client host from     *)
        (* which the command/request was receipt. They are used *)
        (* later to send back the answer to the peer client     *)
        (* via FB LAN_UDP_SENDTO_STR.                           *)
        FB_LanUdpRecvfromStr (
            ENABLE := TRUE,
            NETNUMBER := NETNUMBER,
            SOCKET_ID := SocketID,
            MAXLENGTH := 0,                 (* use StrAllocLen of strRxData *)
            RXDATA := strRxData
            |
            inetPeerIpAddr := PEER_ADDR,
            uiPeerPort     := PEER_PORT,
            iRxDataLen     := RXLENGTH);

        IF (FB_LanUdpRecvfromStr.CONFIRM = TRUE) THEN
            uiProcState := uiProcState + 1; (* new state: Process Command *)
        END_IF;
```

```
(* ----- Process Command ----- *)
2:
    IF (strRxData = STOP_CMD) THEN
        xServerRunning := FALSE;
    END_IF;

    IF (xServerRunning = TRUE) THEN
        (* execute command *)
        strRxCommand := strRxData;
        FB_ExecCommand (
            strCommand_i := strRxCommand
            |
            strTxResult := strResult_o);
    ELSE
        (* show good-by message *)
        strTxResult := '$NServer stopped.$N';
    END_IF;

    (* create answer string *)
    strRxDataLen  := INT_TO_STRING(iRxDataLen);
    strPeerIpAddr := LAN_INET_TO_ASCII(inetPeerIpAddr);
    strPeerPort   := UINT_TO_STRING(uiPeerPort);

    strTxData := CONCAT ('$NPLC: ', strRxDataLen, ' Byte(s) received ',
                         'from IP-Address=', strPeerIpAddr, '/',
                         'Port=', strPeerPort, ':',
                         '$N-> Command: ', strRxData,
                         '$N-> Result:  ', strTxResult);

    uiProcState := uiProcState + 1;     (* new state: Send Response *)



(* ----- Send Response ----- *)
3:
    (* The values PEER_ADDR and PEER_PORT identifies the    *)
    (* client host, to which the answer should be send now. *)
    (* Both values was output parameters from a previous    *)
    (* call of the FB LAN_UDP_RECVFROM_STR.                 *)
    FB_LanUdpSendtoStr (
        ENABLE := TRUE,
        NETNUMBER := NETNUMBER,
        SOCKET_ID := SocketID,
        PEER_ADDR := inetPeerIpAddr,
        PEER_PORT := uiPeerPort,
        TXDATA := strTxData,
        TXLENGTH := 0);

    IF (xServerRunning = TRUE) THEN
        uiProcState := uiProcState - 2; (* go back to reveive state *)
    ELSE
        uiProcState := uiProcState + 1; (* goto finish state *)
    END_IF;



(* ----- Finish Server ----- *)
4:
    FB_LanUdpCloseSocket (
        ENABLE := TRUE,
        NETNUMBER := NETNUMBER,
        SOCKET_ID := SocketID);

    uiProcState := uiProcState + 1;
```

```
    (* ----- Stop State ----- *)
    5:
        ;   (* simply do nothing *)


    (* --- unknown state --- *)
    ELSE
        uiProcState := 0;

END_CASE;


RETURN;

END_PROGRAM
```

4 Securing Process Data in the Nonvolatile Storage

4.1 Application of Nonvolatile Storage for Process Data

The defined PLC program variables can only store the information they contain during the program runtime. This information is usually lost when the program is terminated or the PLC shutdown. The function block *NVDATA_Xxx* (NV = nonvolatile), described in sections 4.2, 4.3 and 4.4, allows the filing of process data in a nonvolatile storage.

Storing process data in the nonvolatile storage allows a PLC program, for example, to continue the operation of production counters even after a system restart. It is also possible to retentively store parameters which have been reconfigured to the system runtime by the user, e.g. via an operating device.

4.2 Function Block NVDATA_BIT

The function block *NVDATA_BIT* writes logical process data (BYTE, WORD, DWORD) in as well as reads stored process data from the PLC nonvolatile storage (EEPROM, file).

<u>Prototype of the Function Block</u>

```
              +--------------+
              |  NVDATA_BIT  |
              |              |
BYTE   ---|DIN1      DOUT1|--- BYTE
WORD   ---|DIN2      DOUT2|--- WORD
DWORD  ---|DIN3      DOUT3|--- DWORD
              |              |
UINT   ---|ADDR       SIZE|--- UINT
USINT  ---|MODE           |
              |              |
USINT  ---|DEVICE    ERROR|--- USINT
              |              |
              +--------------+
```

<u>Definition of Operands</u>

DIN1	Data input for writing a BYTE value
DIN2	Data input for writing a WORD value
DIN3	Data input for writing a DWORD value
ADDR	Address in the nonvolatile storage to read and write data (parameter *MODE* dependent)
MODE	Setting of the read or write operation to be executed, Table 9 contains a list of the supported modes
DOUT1	Data output for reading a BYTE value
DOUT2	Data output for reading a WORD value
DOUT3	Data output for reading a DWORD value
SIZE	This output states the number of written or read bytes (*MODE* <> *0*) or the size of the usable nonvolatile storage (*MODE* = *0*, see text).
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 10.

DEVICE Device number, this parameter depends on the respective control.
(Note: most controls only support the device 0. Therefore, this input does not have to be set since it is already pre-occupied with the initial value 0).

Table 9 Call Modes for the Function Block NVDATA_BIT

Mode	Action
16#00	Determine the size of the usable nonvolatile storage (see text)
16#01	Read a BYTE from the nonvolatile storage at data output *DOUT1*
16#02	Read a WORD from the nonvolatile storage at data output *DOUT2*
16#03	Read a DWORD from the nonvolatile storage at data output *DOUT3*
16#81	Write a BYTE in the nonvolatile storage at data input *DIN1*
16#82	Write a WORD in the nonvolatile storage at data input *DIN2*
16#83	Write a DWORD in the nonvolatile storage at data input *DIN3*

Table 10 Error Codes of the Function Blocks NVDATA_Xxx

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	Invalid device number (*DEVICE*) when calling the function block
4	Invalid mode (*MODE*) when calling the function block
8	Specified address (*ADDR*) is too large, maximum available storage area exceeded
16	Pointer references an object of an unsupported data type

<u>Description</u>

Various types of data can be written in or read from a nonvolatile storage (EEPROM, file) via the function block. Depending on the data to be written or read, the respective mode has to be set at input *MODE* according to Table 9. Ensure that, depending on the selected mode, the data is either stored at the associated data input or read from the associated data output. The passed value at input *ADDR* is the basic address for the read or write operation to be executed. If addressing exceeds the maximum memory size, the function block returns with a corresponding error. The PLC program is fully responsible for partitioning the available memory and for ensuring that the value used at input *ADDR* does not cause overlapping of the data to be stored. The number of read or written bytes is returned to output *SIZE*. This value can then be used to calculate the next free address ($ADDR_{new} := ADDR_{old} + SIZE$).

Calling the block via *MODE = 0* determines the size of the usable nonvolatile storage. For this, the remaining residual size as from the value passed at input ADDR is returned to output *SIZE* ($SIZE := NVDATA_{FullSize} - ADDR$). Call the block via *ADDR := 0* to determine the overall size of the nonvolatile storage.

Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 10.

The following sample program shows the application of the function block *NVDATA_BIT*. At first, a data byte is written from address 10 onwards and subsequently read by the same address. Since input

DEVICE is not set by the user program, the standard setting remains the same and the block implicitly uses the device number 0.

Sample Program

```
PROGRAM SaveDataBit

VAR CONSTANT
    NVDBIT_MODE_GET_SIZE  : USINT := 16#00;
    NVDBIT_MODE_RD_BYTE   : USINT := 16#01;
    NVDBIT_MODE_RD_WORD   : USINT := 16#02;
    NVDBIT_MODE_RD_DWORD  : USINT := 16#03;
    NVDBIT_MODE_WR_BYTE   : USINT := 16#81;
    NVDBIT_MODE_WR_WORD   : USINT := 16#82;
    NVDBIT_MODE_WR_DWORD  : USINT := 16#83;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR

VAR
    WriteDataByte : BYTE;
    WriteDataSize : UINT;
    ReadDataByte : BYTE;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataBit : NVDATA_BIT;
END_VAR

(* write a BYTE value into EEPROM *)
LD      16#10
ST      WriteDataByte

CAL     FB_NvDataBit (
        DIN1 := WriteDataByte,
        ADDR := 10,
        MODE := NVDBIT_MODE_WR_BYTE
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a BYTE value from EEPROM *)
CAL     FB_NvDataBit (
        ADDR := 10,
        MODE := NVDBIT_MODE_RD_BYTE
        |
        ReadDataByte := DOUT1,
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.3 Function Block NVDATA_INT

The function block *NVDATA_INT* writes arithmetical process data (SINT, INT, DINT, REAL) in as well as reads stored process data from the PLC nonvolatile storage (EEPROM, file).

Prototype of the Function Block

```
                +--------------+
                |  NVDATA_INT  |
                |              |
      SINT   ---|DIN1     DOUT1|--- SINT
      INT    ---|DIN2     DOUT2|--- INT
      DINT   ---|DIN3     DOUT3|--- DINT
      REAL   ---|DIN4     DOUT4|--- REAL
                |              |
      UINT   ---|ADDR      SIZE|--- UINT
      USINT  ---|MODE          |
                |              |
      USINT  ---|DEVICE   ERROR|--- USINT
                |              |
                +--------------+
```

Definition of Operands

DIN1	Data input for writing a SINT value
DIN2	Data input for writing an INT value
DIN3	Data input for writing a DINT value
DIN4	Data input for writing a REAL value
ADDR	Address in the nonvolatile storage to read and write data (parameter *MODE* dependent)
MODE	Setting of the read or write operation to be executed, Table 11 contains a list of the supported modes.
DOUT1	Data output for reading a SINT value
DOUT2	Data output for reading an INT value
DOUT3	Data output for reading a DINT value
DOUT4	Data output for reading a REAL value
SIZE	This output states the number of written or read bytes (*MODE* <> *0*) or the size of the usable nonvolatile storage (*MODE* = *0*, see text).
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 10 (they are identical to the error codes of the block *NVDATA_BIT*).
DEVICE	Device number, this parameter depends on the respective control. (Note: most controls only support the device 0. Therefore, this input does not have to be set since it is already pre-occupied with the initial value 0).

Table 11 Call Mode for the Function Block NVDATA_INT

Mode	Action
16#00	Determine the size of the usable nonvolatile storage (see text)
16#01	Read a SINT from the nonvolatile storage at data output *DOUT1*
16#02	Read an INT from the nonvolatile storage at data output *DOUT2*
16#03	Read a DINT from the nonvolatile storage at data output *DOUT3*
16#04	Read a REAL from the nonvolatile storage at data output *DOUT4*
16#81	Write a SINT in the nonvolatile storage at data input *DIN1*
16#82	Write an INT in the nonvolatile storage at data input *DIN2*
16#83	Write a DINT in the nonvolatile storage at data output *DIN3*
16#84	Write a REAL in the nonvolatile storage at data input *DIN4*

Description

Various types of data can be written in or read from a nonvolatile storage (EEPROM, file) via the function block. Depending on the data to be read or written, the respective mode has to be set at input *MODE* according to Table 11. Ensure that, depending on the selected mode, the data is either stored at the associated data input or read from the associated data output. The value passed at input *ADDR* is the basic address for the read or write operation to be executed. If addressing exceeds the maximum memory size, the function block returns with a corresponding error. The PLC program is fully responsible for partitioning the available memory and for ensuring that the value used at input *ADDR* does not cause overlapping of the data to be stored. The number of read or written bytes is returned to output *SIZE*. This value can then be used to calculate the next free address ($ADDR_{new} := ADDR_{old} + SIZE$).

Calling the block via *MODE = 0* determines the size of the usable nonvolatile storage. For this, the remaining residual size as from the value passed at input ADDR is returned to output *SIZE* ($SIZE := NVDATA_{FullSize} - ADDR$). Call the block via *ADDR := 0* to determine the overall size of the nonvolatile storage.

Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 10 (they are identical to the error codes of the *NVDATA_BIT* block).

The following sample program shows the application of the function block *NVDATA_INT*. At first, a REAL value is written from address 20 onwards and subsequently read by the same address. Since input *DEVICE* is not set by the user program, the standard setting remains the same and the block implicitly uses the device number 0.

Sample Program

```
PROGRAM SaveDataInt

VAR CONSTANT
    NVDINT_MODE_GET_SIZE  : USINT := 16#00;
    NVDINT_MODE_RD_SINT   : USINT := 16#01;
    NVDINT_MODE_RD_INT    : USINT := 16#02;
    NVDINT_MODE_RD_DINT   : USINT := 16#03;
    NVDINT_MODE_RD_REAL   : USINT := 16#04;
    NVDINT_MODE_WR_SINT   : USINT := 16#81;
    NVDINT_MODE_WR_INT    : USINT := 16#82;
    NVDINT_MODE_WR_DINT   : USINT := 16#83;
    NVDINT_MODE_WR_REAL   : USINT := 16#84;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR

VAR
    WriteDataReal : REAL;
    WriteDataSize : UINT;
    ReadDataReal : REAL;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataInt : NVDATA_INT;
END_VAR

(* write a REAL value into EEPROM *)
LD      7.89
ST      WriteDataReal

CAL     FB_NvDataInt (
        DIN4 := WriteDataReal,
        ADDR := 20,
        MODE := NVDINT_MODE_WR_REAL
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a REAL value from EEPROM *)
CAL     FB_NvDataInt (
        ADDR := 20,
        MODE := NVDINT_MODE_RD_REAL
        |
        ReadDataReal := DOUT4,
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.4 Function Block NVDATA_STR

The function block *NVDATA_STR* writes character string-based process data (STRING) in as well as reads stored process data from the PLC nonvolatile storage (EEPROM, file).

Prototype of the Function Block

```
          +--------------+
          |  NVDATA_STR  |
          |              |
STRING ---|DINOUT--DINOUT|--- STRING
INT    ---|LENGTH        |
BOOL   ---|SETEOT        |
          |              |
UINT   ---|ADDR      SIZE|--- UINT
USINT  ---|MODE          |
          |              |
USINT  ---|DEVICE   ERROR|--- USINT
          |              |
          +--------------+
```

Definition of Operands

DINOUT	Data in and output for reading or writing a STRING value
LENGTH	Limitation of the number of characters to be read or written. If 0, the buffer length of the passed string is internally determined and used as the number of characters to be read or written (equals *LEN(DINOUT);*) Note: the standard string buffer size in OpenPCS is 32 characters.
ADDR	Address in the nonvolatile storage to read and write data (parameter *MODE* dependent)
MODE	Setting of the read or write operation to be executed: Table 12 contains a list of the supported modes
SETEOT	TRUE: The string is stored with terminating character FALSE: Storage of the terminating character is blanked (Default: TRUE, see text)
SIZE	This output states the number of written or read bytes (*MODE* <> *0*) or the size of the usable nonvolatile storage (*MODE = 0*, see text).
ERROR:	The error code states information about the execution result of the function block. Possible error codes are defined in Table 10 (they are identical to the error codes of the *NVDATA_BIT* block).
DEVICE	Device number, this parameter depends on the respective control. (Note: most controls only support the device 0. Therefore, this input does not have to be set since it is already pre-occupied with the initial value 0).

Table 12 Call Mode for the Function Block NVDATA_STR

Mode	Action
16#00	Determine the size of the usable nonvolatile storage (see text)
16#08	Read a STRING from the nonvolatile storage at data output *DINOUT*
16#88	Write a STRING into the nonvolatile storage at data input *DINOUT*

Description

Character string-based data can be written in or read from a nonvolatile storage (EEPROM, file) via the function block. Depending on the data to be read or written, the respective mode has to be set at input *MODE* according to Table 12. For this, the parameter *DINOUT* is used as in or output depending on the mode. The value passed at input *ADDR* is the basic address for the read or write operation to be executed. If addressing exceeds the maximum memory size, the function block returns with a corresponding error. The PLC program is fully responsible for partitioning the available memory and for ensuring that the value used at input *ADDR* does not cause overlapping of the data to be stored. The number of read or written bytes is returned to output *SIZE*. This value can then be used to calculate the next free address ($ADDR_{new} := ADDR_{old} + SIZE$).

Input *LENGTH* specifies the number of valid characters during writing. If this value is 0, the length of the character string in the string is determined internally (equals *LEN(DINOUT);*) and used as the number of characters to be written. In this case, the entire occupied string content is written. Input *LENGTH* can be used during reading to limit the number of characters to be processed to the specified value.

Use input *SETEOT* to set whether the string’s terminating character should also be stored or not (Default: TRUE). If the string is completely stored in the nonvolatile storage together with the terminating character, the length is not necessary during reading at input *LENGTH* (*LENGTH = 0*). The block accepts all the characters until the end delimiter in the string passed to parameter *DINOUT*. Storage of the terminating character is blanked when the block is called via *SETEOT = FALSE*. Therefore, one byte less is occupied per string in the non-volatile storage. However, in this case the string length has to be known and specified at input *LENGTH* during reading. If the terminating character has been written, it is taken into consideration when the processed characters are specified at output *SIZE*. Therefore, when calling the block via *SETEOT = TRUE*, the value of output SIZE is equal to *LEN(DINOUT) + 1*.

Calling the block via *MODE = 0* determines the size of the usable nonvolatile storage. For this, the remaining residual size as from the value passed at input ADDR is returned to output *SIZE* ($SIZE := NVDATA_{FullSize} - ADDR$). Call the block via *ADDR := 0* to determine the overall size of the nonvolatile storage.

Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 10 (they are identical to the error codes of the *NVDATA_BIT* block).

The following sample program shows the application of the function block *NVDATA_STR*. At first, a string is written from address 30 onwards and subsequently read by the same address. Since input *DEVICE* is not set by the user program, the standard setting remains the same and the block implicitly uses the device number 0.

Sample Program

```
PROGRAM SaveDataStr

VAR CONSTANT
    NVDSTR_MODE_GET_SIZE  : USINT := 16#00;
    NVDSTR_MODE_RD_STRING : USINT := 16#08;
    NVDSTR_MODE_WR_STRING : USINT := 16#88;

    NVDATA_ERROR_SUCCESS        : USINT := 0;
    NVDATA_ERROR_HW_ERROR       : USINT := 1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT := 2;
    NVDATA_ERROR_INVALID_MODE   : USINT := 4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT := 8;
END_VAR
```

```
VAR
    WriteDataString : STRING;
    WriteDataSize : UINT;
    ReadDataString : STRING;
    ReadDataSize : UINT;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataStr : NVDATA_STR;
END_VAR

(* write a STRING value into EEPROM *)
LD      'HelloWorld'
ST      WriteDataString

CAL     FB_NvDataStr (
        DINOUT := WriteDataString,
        LENGTH := 0,            (* save whole string             *)
        SETEOT := TRUE,         (* include termination character *)
        ADDR := 30,
        MODE := NVDSTR_MODE_WR_STRING
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a STRING value from EEPROM *)
CAL     FB_NvDataStr (
        DINOUT := ReadDataString,
        LENGTH := 0,            (* read whole string             *)
        ADDR := 30,
        MODE := NVDSTR_MODE_RD_STRING
        |
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

4.5 Function Block NVDATA_BIN

The function block *NVDATA_BIN* writes binary process data in as well as reads stored process data from the PLC nonvolatile storage (EEPROM, file).

<u>Prototype of the Function Block</u>

```
                  +--------------+
                  |  NVDATA_BIN  |
                  |              |
      POINTER  ---|PTR_DINOUT    |
      INT      ---|LENGTH        |
                  |              |
      UINT     ---|ADDR      SIZE|--- UINT
      USINT    ---|MODE          |
                  |              |
      USINT    ---|DEVICE   ERROR|--- USINT
                  |              |
                  +--------------+
```

Definition of Operands

DINOUT	Data in and output for reading or writing of binary data
LENGTH	Limitation of the number of bytes to be read or written. If 0, the length of the object addressed by *PTR_DINOUT* is internally determined and used as the number of characters to be read or written
ADDR	Address in the nonvolatile storage to read and write data (parameter *MODE* dependent)
MODE	Setting of the read or write operation to be executed: Table 13 contains a list of the supported modes
SIZE	This output states the number of written or read bytes (*MODE* <> *0*) or the size of the usable nonvolatile storage (*MODE = 0*, see text).
ERROR:	The error code states information about the execution result of the function block. Possible error codes are defined in Table 10 (they are identical to the error codes of the *NVDATA_BIT* block).
DEVICE	Device number, this parameter depends on the respective control. (Note: most controls only support the device 0. Therefore, this input does not have to be set since it is already pre-occupied with the initial value 0).

Table 13 Call Mode for the Function Block NVDATA_BIN

Mode	Action
16#00	Determine the size of the usable nonvolatile storage (see text)
16#09	Read binary data from the nonvolatile storage, read data are saved in the object addressed by *PTR_DINOUT*
16#89	Write binary data into the nonvolatile storage, data are taken from object addressed by *PTR_DINOUT*

Description

Binary data can be written in or read from a nonvolatile storage (EEPROM, file) via the function block. Depending on the data to be read or written, the respective mode has to be set at input *MODE* according to Table 13. For this, the object addressed by parameter *DINOUT* is used as data source or destination, depending on the mode. The value passed at input *ADDR* is the basic address for the read or write operation to be executed. If addressing exceeds the maximum memory size, the function block returns with a corresponding error. The PLC program is fully responsible for partitioning the available memory and for ensuring that the value used at input *ADDR* does not cause overlapping of the data to be stored. The number of read or written bytes is returned to output *SIZE*. This value can then be used to calculate the next free address ($ADDR_{new} := ADDR_{old} + SIZE$).

Input *LENGTH* specifies the number bytes to process. If this value is 0, the length of the object addressed by PTR_TXDATA is internally determined and used as the number of bytes to be read or written.

Calling the block via *MODE = 0* determines the size of the usable nonvolatile storage. For this, the remaining residual size as from the value passed at input ADDR is returned to output *SIZE* ($SIZE := NVDATA_{FullSize} - ADDR$). Call the block via *ADDR := 0* to determine the overall size of the nonvolatile storage.

Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 10 (they are identical to the error codes of the *NVDATA_BIT* block).

The following sample program shows the application of the function block *NVDATA_BIN*. At first, a data object is written from address 30 onwards and subsequently read by the same address. Since input *DEVICE* is not set by the user program, the standard setting remains the same and the block implicitly uses the device number 0.

Sample Program

```
PROGRAM SaveDataBin

VAR CONSTANT
    NVDSTR_MODE_GET_SIZE  : USINT := 16#00;
    NVDSTR_MODE_RD_BIN    : USINT := 16#09;
    NVDSTR_MODE_WR_BIN    : USINT := 16#89;

    NVDATA_ERROR_SUCCESS        : USINT :=  0;
    NVDATA_ERROR_HW_ERROR       : USINT :=  1;
    NVDATA_ERROR_UNKNOWN_DEVICE : USINT :=  2;
    NVDATA_ERROR_INVALID_MODE   : USINT :=  4;
    NVDATA_ERROR_OUT_OF_MEM     : USINT :=  8;
    NVDATA_ERROR_PTR_TYPE       : USINT := 16;
END_VAR

VAR
    WriteDataObject : ARRAY[0..3] OF BYTE := [ 16#01, 16#02, 16#03, 16#04 ];
    ReadDataObject  : ARRAY[0..3] OF BYTE;
    Error : ARRAY[0..1] OF USINT;

    FB_NvDataBin : NVDATA_BIN;
    pDataObject  : POINTER;
END_VAR

(* write a binary data object into EEPROM *)
LD      &WriteDataObject
ST      pDataObject

CAL     FB_NvDataBin (
        PTR_DINOUT := pDataObject,
        LENGTH := 0,            (* save whole object *)
        ADDR := 30,
        MODE := NVDSTR_MODE_WR_BIN
        |
        WriteDataSize := SIZE,
        Error[0] := ERROR)

(* read a binary data object from EEPROM *)
LD      &ReadDataObject
ST      pDataObject

CAL     FB_NvDataBin (
        PTR_DINOUT := pDataObject,
        LENGTH := 0,            (* read whole object *)
        MODE := NVDSTR_MODE_RD_BIN
        ADDR := 30,
        |
        ReadDataSize := SIZE,
        Error[1] := ERROR)

RET

END_PROGRAM
```

5 Access to Serial Interface (SIO)

5.1 Application of the Serial Interface

The serial interface enables data exchange with other devices via direct point-to-point connection. It can, e.g., be used for data output on a printer or operation terminal control. Depending on the hardware design of the serial interface, the data flow can be influenced by different handshake protocols, e.g., via modem control lines (RTS, CTS, DTR, DSR) or XON/XOFF. Due to the function blocks *SIO_INIT* and *SIO_STATE* it is possible to initialize and control the interface. Status information can also be retrieved. With the function blocks *SIO_READ_CHR* and *SIO_WRITE_CHR* it is possible to process single characters. With *SIO_READ_STR* and *SIO_WRITE_STR* on the other hand it is possible to transfer character strings.

5.2 Function Block SIO_INIT

The function block *SIO_INIT* initializes the serial interface and sets the handshake protocol for the flow control.

Prototype of the Function Block

```
                +----------------+
                |    SIO_INIT    |
                |                |
      UDINT  ---|BAUD            |
      USINT  ---|DATABITS        |
      USINT  ---|PARITY          |
      USINT  ---|STOPBITS        |
      USINT  ---|PROTOCOL        |
                |                |
      BOOL   ---|ENABLE          |
                |                |
      USINT  ---|PORT       ERROR|--- USINT
                |                |
                +----------------+
```

Definition of Operands

BAUD
Specification of the baud rate to be used in bps, this parameter depends on the properties of the respective interface, valid values are, e.g.:

1200, 2400, 9600, 19200, 38400, 57600, 115200

DATABITS
Specification of the number of data bits to be used, this parameter depends on the properties of the respective interface, valid values are, e.g.:
7 = 7 data bits
8 = 8 data bits

PARITY
Specification of the parity to be used for secure data transfer, this parameter depends on the properties of the respective interface, valid values are, e.g.:
0 = no parity
1 = odd parity
2 = even parity

STOP BITS	Specification of the number of stop bits to be used, this parameter depends on the properties of the respective interface, valid values are, e.g.: 1 = 1 stop bit 2 = 2 stop bits
PROTOCOL	Specification of the handshake protocols to be used, this parameter depends on the properties of the respective interface, valid values are, e.g.: 0 = no protocol 1 = XON/XOFF 2 = hardware handshake (RTS/CTS flow control)
ENABLE	Enable or disable the serial interface TRUE = initialize the serial interface FALSE = switch off the serial interface
PORT	Number of serial interface to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 14.

Table 14 Error Codes of the Function Block SIO_INIT

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
4	The selected bit rate (*BAUD*) is not supported
8	The selected number of data bits (*DATABITS*) is not supported
16	The selected parity (*PARITY*) is not supported
32	The selected number of stop bits is not supported
64	The selected handshake protocol is not supported

Description

The function block initializes the serial interface with the specified parameters. The actual availability or support of parameters depends on the respective hardware properties of the interface. Please see the respective manual of each control for more detailed information. Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 14. Due to the simultaneous setting of various bits it is possible to signalize several errors (e.g. 136 = 128 + 8 => invalid bit rate and non-supported protocol).

The following sample program shows the application of the function block *SIO_INIT* to initialize the serial interface with the following parameters: 9600 baud, 8 data bits, no parity, 1 stop bit, software flow control via XON/XOFF protocol.

Sample Program

```
VAR CONSTANT
    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT := 0;
    SIO_INIT_PARITY_ODD             : USINT := 1;
    SIO_INIT_PARITY_EVEN            : USINT := 2;

    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT := 0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT := 1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT := 2;

    (* Error Codes of FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT := 0;
    SIO_INIT_ERR_HW_ERROR           : USINT := 1;
    SIO_INIT_ERR_INVALID_PORT       : USINT := 2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT := 8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT := 16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT := 32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT := 64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;

    PORTNUM : USINT := 1;
END_VAR

VAR
    FB_SioInit : SIO_INIT;
    xInitOk    : BOOL := FALSE;
END_VAR

(* ----- Init Sio ----- *)
SioInit:
(* Initialize Serial Port *)
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_XON_XOFF,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
ST      xInitOk

...

RET
```

5.3 Function Block SIO_STATE

The function block *SIO_SATE* sets and retrieves status information of the serial interface.

Prototype of the Function Block

```
          +-----------------+
          |    SIO_STATE    |
          |                 |
SINT   ---|RTS           CTS|--- SINT
SINT   ---|DTR           DSR|--- SINT
          |              DCD|--- SINT
SINT   ---|CLR            RI|--- SINT
          |                 |
          |            SORXQ|--- UDINT
          |            CBRXQ|--- UDINT
          |                 |
          |            SOTXQ|--- UDINT
          |            CBTXQ|--- UDINT
          |                 |
          |          SIOSTAT|--- INT
          |                 |
USINT  ---|PORT        ERROR|--- USINT
          |                 |
          +-----------------+
```

Definition of Operands

RTS	RTS signal status to be set: -1 = do not influence current status 0 = set signal to inactive 1 = set signal to active
DTR	DTR signal status to be set: -1 = do not influence current status 0 = set signal to inactive 1 = set signal to active
CLR	Clear send and receive buffer: -1 = do not influence current status 1 = clear receive buffer 2 = clear send buffer 3 = clear send and receive buffer
CTS	Determined CTS signal status: -1 = signal is not supported 0 = signal is set as inactive 1 = signal is set as active
DSR	Determined DSR signal status -1 = signal is not supported 0 = signal is set as inactive 1 = signal is set as active
DCD	Determined DCD signal status: -1 = signal is not supported 0 = signal is set as inactive 1 = signal is set as active

RI	Determined RI signal status: -1 = signal is not supported 0 = signal is set as inactive 1 = signal is set as active
SORXQ	Determined overall size of the receive buffer (Size of Rx Queue)
CBRXQ	Current number of the characters in the receive buffer (Count of Bytes in Rx Queue)
SOTXQ	Determined overall size of the send buffer (Size of Tx Queue)
CBTXQ	Current number of characters in the send buffer (Count of Bytes in Tx Queue)
SIOSTAT	SIO specific status register (e.g. overrun, frame error etc.; see the manual of the respective control)
PORT	Number of serial interface to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 15.

Table 15 Error Codes of the Function Block SIO_STAT

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
8	The RTS signal cannot be influenced if a hardware handshake is active (*SIO_INIT* called via *PROTOCOL := 2*)
16	The DTR signal cannot be influenced if a hardware handshake is active (*SIO_INIT* called with *PROTOCOL := 2*)
32	Direct setting of the RTS signal is not supported
64	Direct setting of the DTR signal is not supported
128	Clearing of the send and receive buffer is not supported
255	The selected interface (*PORT*) is not initialized

Description

The function block sets and retrieves status information of the serial interface. The actual availability or support of parameters depends on the respective hardware properties of the interface. Please see the respective manual of each control for more detailed information. Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described Table 15. Due to simultaneous setting of various bits it is possible to signalize several errors (e.g. 24 = 16 + 8 => RTS and DTR signals cannot be influenced during an activated hardware handshake).

The following extract from the program shows the application of the function block *SIO_STAT* to determine the current status of the serial interface.
Note: See section 5.7 for the complete sample program including initialization and flow control.

Sample Program

```
VAR CONSTANT
    (* Definition of Control Codes *)
    SIO_STAT_DO_NOT_CHANGE          : SINT  :=  -1;
    SIO_STAT_CLR                    : SINT  :=  0;
    SIO_STAT_SET                    : SINT  :=  1;

    (* Error Codes of FB SIO_STAT *)
    SIO_STAT_ERR_SUCCESS            : USINT := 0;
    SIO_STAT_ERR_HW_ERROR           : USINT := 1;
    SIO_STAT_ERR_INVALID_PORT       : USINT := 2;
    SIO_STAT_ERR_RTS_SET_ERROR      : USINT := 8;
    SIO_STAT_ERR_DTR_SET_ERROR      : USINT := 16;
    SIO_STAT_ERR_RTS_NOT_SUPPORTED  : USINT := 32;
    SIO_STAT_ERR_DTR_NOT_SUPPORTED  : USINT := 64;
    SIO_STAT_ERR_CLR_NOT_SUPPORTED  : USINT := 128;
    SIO_STAT_ERR_NOT_INITIALIZED    : USINT := 255;

    PORTNUM : USINT := 1;
END_VAR

VAR
    FB_SioState : SIO_STATE;
    xStatOk     : BOOL := FALSE;

    siCts       : SINT;
    siDsr       : SINT;
    siDcd       : SINT;
    siRi        : SINT;
    udiSoRrQ    : UDINT;
    udiCbRxQ    : UDINT;
    udiSoTxQ    : UDINT;
    udiCbTxQ    : UDINT;
    iSioStat    : INT;
END_VAR

(* ----- Check Sio State ----- *)
CheckState:
(*      read current state from serial interface *)
CAL     FB_SioState (
        RTS := SIO_STAT_DO_NOT_CHANGE,
        DTR := SIO_STAT_DO_NOT_CHANGE,
        CLR := SIO_STAT_DO_NOT_CHANGE,
        PORT := PORTNUM
        |
        siCts := CTS,
        siDsr := DSR,
        siDcd := DCD,
        siRi := RI,
        udiSoRrQ := SORXQ,
        udiCbRxQ := CBRXQ,
        udiSoTxQ := SOTXQ,
        udiCbTxQ := CBTXQ,
        iSioStat := SIOSTAT)

LD      FB_SioState.ERROR
EQ      SIO_STAT_ERR_SUCCESS
ST      xStatOk

...

RET
```

5.4 Function Block SIO_READ_CHR

The function block *SIO_READ_CHR* reads a single character from the serial interface.

Prototype of the Function Block

```
                +----------------+
                |  SIO_READ_CHR  |
                |                |
      BOOL   ---|ECHO      RXDATA|--- USINT
                |                |
      USINT  ---|PORT       ERROR|--- USINT
                |                |
                +----------------+
```

Definition of Operands

ECHO — Character echo on/off
FALSE = no echo
TRUE = return echo

RXDATA — Received character (if *ERROR := 0*)

PORT — Number of serial interface to be used

ERROR — The error code states information about the execution result of the function block. Possible error codes are defined in Table 16.

Table 16 Error Codes of the Function Block SIO_READ_CHR

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
8	No character available in the receive buffer
16	A character echo is not supported
255	The selected interface (*PORT*) is not initialized

Description

The function block reads a single character from the serial interface. No character was available in the receive buffer if output *ERROR := 8* has been set when the block returns. The read character is available at output *RXDATA* if *ERROR := 0*. In this case, the received character is returned by the block as an echo if output *ECHO := TRUE* has been set. Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 16. Due to the simultaneous setting of various bits it is possible to signalize several errors.

Sample Program

The program extract in section 5.5 shows the joint application of *SIO_READ_CHR* and the function block *SIO_WRITE_CHR*. At first, the sample program calls the block *SIO_READ_CHR* to read a character from the interface. If this was successful, the block *SIO_WRITE_CHR* rewrites the character on the same interface as an echo.
Note: See section 5.7 for the complete sample program including initialization and flow control.

5.5 Function Block SIO_WRITE_CHR

The function block *SIO_WRITE_CHR* writes a single character onto the serial interface.

Prototype of the Function Block

```
              +-----------------+
              |  SIO_WRITE_CHR  |
              |                 |
     USINT  ---|TXDATA           |
              |                 |
     BOOL   ---|EXPANDCR         |
     BOOL   ---|EXPANDLF         |
              |                 |
     USINT  ---|PORT       ERROR|--- USINT
              |                 |
              +-----------------+
```

Definition of Operands

TXDATA	Input for the character to be sent
EXPANDCR	Automatic expansion of the carriage return on/off FALSE: No automatic expansion of the carriage return TRUE: Automatic expansion of the carriage return, CR ('$R'=13) is automatically expanded to CR+LF ('RL'=13+10)
EXPANDLF	Automatic expansion of the line feed on/off FALSE: No automatic expansion of the line feed TRUE: Automatic expansion of the line feed, LF ('$L'=10) is automatically expanded to CR+LF ('RL'=13+10)
PORT	Number of serial interface to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 17.

Table 17 Error Codes of the Function Block SIO_WRITE_CHR

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
8	There is no space available in the send buffer, the character has been discarded
16	The automatic expansion of the carriage return is not supported
32	The automatic expansion of the line break is not supported
255	The selected interface (*PORT*) is not initialized

Description

The function block writes a single character onto the serial interface. No space was available in the send buffer if output *ERROR := 1* has been set when the block returns. Output *TXDATA* has successfully written the character onto the interface if *ERROR := 0*. If input *EXPANDCR := TRUE* has been set, a check is carried out to see whether the transmitted character corresponds to the ASCII code for the carriage return. In this case, the block automatically expands this character to the character string carriage return+line break. Similar to this, the block also automatically adds (internally) a line break for the character string carriage return+line break if input *EXPANDLF := TRUE* has been set. Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 17. Due to the simultaneous setting of various bits it is possible to signalize several errors.

The following extract of the program shows the application of the function blocks *SIO_READ_CHR* (see section 5.4) and *SIO_WRITE_CHR* to read and write characters via the serial interface. At first, the sample program calls the block *SIO_READ_CHR* to read a character from the interface. If this was successful, the block *SIO_WRITE_CHR* rewrites the character on the same interface as an echo.
Note: See section 5.7 for the complete sample program including initialization and flow control.

Sample Program

```
VAR CONSTANT
    (* Error Codes of FB SIO_READ_CHR *)
    SIO_RCHR_ERR_SUCCESS            : USINT := 0;
    SIO_RCHR_ERR_HW_ERROR           : USINT := 1;
    SIO_RCHR_ERR_INVALID_PORT       : USINT := 2;
    SIO_RCHR_ERR_NO_CHAR            : USINT := 8;
    SIO_RCHR_ERR_ECHO_NOT_SUPPORTED : USINT := 16;
    SIO_RCHR_ERR_NOT_INITIALIZED    : USINT := 255;

    (* Error Codes of FB SIO_WRITE_CHR *)
    SIO_WCHR_ERR_SUCCESS            : USINT := 0;
    SIO_WCHR_ERR_HW_ERROR           : USINT := 1;
    SIO_WCHR_ERR_INVALID_PORT       : USINT := 2;
    SIO_WCHR_ERR_TXBUFFER_OVERFLOW  : USINT := 8;
    SIO_WCHR_ERR_EXCR_NOT_SUPPORTED : USINT := 16;
    SIO_WCHR_ERR_EXLF_NOT_SUPPORTED : USINT := 32;
    SIO_WCHR_ERR_NOT_INITIALIZED    : USINT := 255;

    PORTNUM : USINT := 1;
END_VAR
```

```
VAR
    xEcho            : BOOL := FALSE;
    FB_SioReadChr    : SIO_READ_CHR;
    usiRxData        : USINT;
    xRdCharSuccess   : BOOL := FALSE;

    xExpandCR        : BOOL := FALSE;
    xExpandLF        : BOOL := FALSE;
    FB_SioWriteChr   : SIO_WRITE_CHR;
    xWrCharOk        : BOOL := FALSE;
END_VAR

(* ----- Read Char ----- *)
CAL     FB_SioReadChr (
        ECHO := xEcho,
        PORT := PORTNUM
        |
        usiRxData := RXDATA)

(*      check receive result *)
LD      FB_SioReadChr.ERROR     (* character received ? *)
EQ      SIO_RCHR_ERR_SUCCESS
ST      xRdCharSuccess
RETCN

(* ----- Write Char ----- *)
CAL     FB_SioWriteChr (
        TXDATA := usiRxData,
        EXPANDCR := xExpandCR,
        EXPANDLF := xExpandLF,
        PORT := PORTNUM)

LD      FB_SioWriteChr.ERROR
EQ      SIO_WCHR_ERR_SUCCESS
ST      xWrCharOk

...

RET
```

5.6 Function Block SIO_READ_STR

The function block *SIO_READ_STR* reads a character string from the serial interface.

Prototype of the Function Block

```
              +--------------------+
              |    SIO_READ_STR    |
              |                    |
BOOL     ---|ENABLE       CONFIRM|--- BOOL
              |                    |
STRING ---|RXDATA--------RXDATA|--- STRING
INT      ---|MAXLENGTH   RXLENGTH|--- INT
              |                    |
USINT  ---|EOTCHR                |
BOOL     ---|ECHO                  |
BOOL     ---|EDIT                  |
              |                    |
USINT  ---|PORT            ERROR|--- USINT
              |                    |
              +--------------------+
```

Definition of Operands

RXDATA String variable for receiving the read characters

MAXLENGTH Limitation of the number of characters to be read. If the number is 0, the buffer length of the passed string is internally determined and used as the delimiter for the number of characters to be read (Note: the standard buffer size of a string in OpenPCS is 32 characters).

EOTCHR Character for the string end delimiter (Default: 10='$L'), e.g.:
0 (NUL), 10 ('$L'=line break), 13 ('$R'=carriage return)

ECHO Character echo on/off
FALSE = no echo
TRUE = return echo

EDIT Edit mode on/off
FALSE = BS (8) is stored as normal character in the receive string
TRUE = BS (8) is interpreted as a correction character

RXLENGTH Length of the read character string (if *ERROR := 0*)

ENABLE Input for enabling or disabling the FB (see text)

CONFIRM Output for completed message via the FB (see text)
FALSE = reception not successfully completed or terminated after error
TRUE = reception successfully completed, *RXLENGTH* characters are available in the receive buffer *RXDATA*

PORT Number of serial interface to be used

ERROR The error code states information about the execution result of the function block. Possible error codes are defined in Table 18.

Table 18 Error Codes of the Function Block SIO_READ_STR

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
8	No character received for the end delimiter, reception termination after *MAXLENGHTH* characters
16	A character echo is not supported
32	The edit mode is not supported
255	The selected interface (*PORT*) is not initialized

Description

The function block reads a character string from the serial interface. The read characters are stored in the string passed to input *RXDATA*. Reading of the character string is terminated if the character defined for the end delimiter at input *EOTCHR* has been received, or if the string set with the *MAXLENGTH* number of characters is full (if *EDIT* is taken into consideration; if *MAXLENGTH := 0,* the buffer length of the passed string is internally determined and used as the delimiter). In both cases, the returning block displays that reception has been completed and that the string passed to input RXDATA contains the read character string by setting output CONFIRM to TRUE. Output *RXLENGTH* shows the number of characters in the receive buffer (equals *LEN(RXDATA);*). If output *ERROR := 0* has also been set, the character for the end delimiter defined at input *EOTCHR* has been received. If *ERROR := 8,* reception has been terminated after reading the number of characters set as *MAXLENGTH*.

The block starts character reception after detecting a rising edge at input *ENABLE* (first call via *ENABLE := TRUE*). Repeatedly call the function block via the PLC program until character reception (end delimiter or *MAXLENGTH* characters) has been terminated. For this, input *ENABLE* has to be set as TRUE to enable character reception. The block signals successful termination of reception by setting output *CONFIRM* to TRUE. After processing the received character string, the PLC program has to call the block via *ENABLE := FALSE* to internally reset the block to its initial state. Further characters can subsequently be received by resetting input *ENABLE* to TRUE and thus detecting a rising edge. Active reception can be terminated at any time by calling the block via *ENABLE := FALSE*.

The block automatically returns each received character as an echo if input *ECHO := TRUE* has been set. The character backspace (BS=8) is not stored as a normal character but as a correction character in the receive buffer if input *EDIT := TRUE* has been set. The last received character is thus deleted and the number of the already received characters reported at output *RXLENGTH* reduced.

Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 18. Due to the simultaneous setting of various bits it is possible to signalize several errors.

The following sample program shows the application of the function block *SIO_READ_STR* for reading a character string from the serial interface.

Sample Program

The sample program in section 5.7 shows the joint application of *SIO_READ_STR* and the function block *SIO_WRITE_STR*. At first, the sample program calls the block *SIO_READ_STR* to read a character string from the interface. After the character string has been completely read, the block *SIO_WRITE_STR* rewrites it onto the interface.

5.7 Function Block SIO_WRITE_STR

The function block *SIO_WRITE_STR* writes a character string onto the serial interface.

Prototype of the Function Block

```
          +---------------------+
          |    SIO_WRITE_STR    |
          |                     |
BOOL   ---|ENABLE        CONFIRM|--- BOOL
          |                     |
STRING ---|TXDATA---------TXDATA|--- STRING
INT    ---|TXLENGTH             |
          |                     |
BOOL   ---|EXPANDCR             |
BOOL   ---|EXPANDLF             |
BOOL   ---|APPENDLF             |
          |                     |
USINT  ---|PORT            ERROR|--- USINT
          |                     |
          +---------------------+
```

Definition of Operands

TXDATA	String variable with the string to be written
TXLENGTH	Number of characters to be written, if the number is 0, the length of the character string contained in the string *TXDATA* is internally determined (equals *LEN(TXDATA);*) and used as the number of characters to be written.
EXPANDCR	Automatic expansion of the carriage return on/off FALSE: No automatic expansion of the carriage return TRUE: Automatic expansion of the carriage return, CR ('$R'=13) is automatically expanded to CR+LF ('RL'=13+10)
EXPANDLF	Automatic expansion of the line feed on/off FALSE: No automatic expansion of the line feed TRUE: Automatic expansion of the line feed, LF ('$L'=10) is automatically expanded to CR+LF ('RL'=13+10)
APPENDLF	Automatic appending of the line feed on/off FALSE: No automatic appending of the line feed TRUE: Automatic appending of a line feed, LF ('$L'=10) is appended if *EXPANDLF:=FALSE*, CR+LF ('RL'=13+10) is appended if *EXPANDLF:=TRUE*
ENABLE	Input for enabling or disabling the FB (see text)
CONFIRM	Output for completed message via the FB (see text) FALSE = transmission not successfully completed or terminated after error TRUE = transmission successfully completed
PORT	Number of serial interface to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 19.

Table 19 Error Codes of the Function Block SIO_WRITE_STR

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
16	The automatic expansion of the carriage return is not supported
32	The automatic expansion of the line break is not supported
64	The automatic attachment of the line break is not supported
255	The selected interface (*PORT*) is not initialized

Description

The function block writes a character string onto the serial interface. The string with the character string to be transferred should be passed at input *TXDATA*. Here, input *TXLENGTH* specifies the number of valid characters. If this value is 0, the length of the character string contained in string *TXDATA* is internally determined (equals *LEN(TXDATA);*) and used as the number of characters to be written. In this case, the entire occupied string content is written.

The block starts writing the character string after detecting a rising edge at input *ENABLE* (first call via *ENABLE := TRUE*). Repeatedly call the function block via the PLC program until character transfer has been completed. For this, input *ENABLE* has to be set as TRUE to enable character transmission. The block automatically signals successful termination by setting output *CONFIRM* to TRUE. During further processing, the PLC program has to call the block via *ENABLE := FALSE* to internally reset the block to its initial state. Further character transfer can subsequently be started by resetting input *ENABLE* to TRUE and thus detecting a rising edge. Active transmission can be terminated at any time by calling the block via *ENABLE := FALSE*.

If input *EXPANDCR := TRUE* has been set, the block automatically expands (internally) each character with the ASCII code for the carriage return to the character string carriage return+line break. Similar to this, the block also automatically adds (internally) a line break for the character string carriage return+line feed if input *EXPANDLF := TRUE* has been set. If input *APPENDLF := TRUE* has been set, the block internally appends a line break after complete transfer of the character string passed to input *TXDATA*. Depending on input *APPENDLF*, this line break is, if necessary, transferred as a character string consisting of carriage return+line break.

Possible errors during execution of the function block are displayed at output ERROR as a bit mask and described in Table 19. Due to the simultaneous setting of various bits it is possible to signalize several errors.

The following sample program shows the application of the function blocks *SIO_READ_STR* (see section 5.6) and *SIO_WRITE_STR*. At first, the block *SIO_READ_STR* is called to read a character string from the interface. After the character string has been completely read, the block *SIO_WRITE_STR* rewrites it onto the interface. This sample program is completed via initialization of the serial interface and the flow control of the program execution.

Sample Program

```
PROGRAM SioRwStr
VAR CONSTANT

    (* Definition of specail ASCII-Codes *)
    strDollar       : STRING := '$$';   (* Dollar         *)
    strApostroph    : STRING := '$'';   (* Apostroph      *)
    strLF           : STRING := '$L';   (* LineFeed       *)
    strCR           : STRING := '$R';   (* CarriageReturn *)
    strNL           : STRING := '$N';   (* NewLine        *)
    strFF           : STRING := '$P';   (* NewPage        *)
    strTab          : STRING := '$T';   (* Tabulator      *)

    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT :=   0;
    SIO_INIT_PARITY_ODD             : USINT :=   1;
    SIO_INIT_PARITY_EVEN            : USINT :=   2;

    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT :=   0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT :=   1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT :=   2;
```

```
    (* Error Codes for FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT :=   0;
    SIO_INIT_ERR_HW_ERROR           : USINT :=   1;
    SIO_INIT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT :=   8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT :=  16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT :=  32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT :=  64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;


    (* Error Codes for FB SIO_READ_STR *)
    SIO_RSTR_ERR_SUCCESS            : USINT :=   0;
    SIO_RSTR_ERR_HW_ERROR           : USINT :=   1;
    SIO_RSTR_ERR_INVALID_PORT       : USINT :=   2;
    SIO_RSTR_ERR_NO_EOT_CHAR        : USINT :=   8;
    SIO_RSTR_ERR_ECHO_NOT_SUPPORTED : USINT :=  16;
    SIO_RSTR_ERR_EDIT_NOT_SUPPORTED : USINT :=  32;
    SIO_RSTR_ERR_NOT_INITIALIZED    : USINT := 255;


    (* Error Codes for FB SIO_WRITE_STR *)
    SIO_WSTR_ERR_SUCCESS            : USINT :=   0;
    SIO_WSTR_ERR_HW_ERROR           : USINT :=   1;
    SIO_WSTR_ERR_INVALID_PORT       : USINT :=   2;
    SIO_WSTR_ERR_TXBUFFER_OVERFLOW  : USINT :=   8;
    SIO_WSTR_ERR_EXCR_NOT_SUPPORTED : USINT :=  16;
    SIO_WSTR_ERR_EXLF_NOT_SUPPORTED : USINT :=  32;
    SIO_WSTR_ERR_APLF_NOT_SUPPORTED : USINT :=  64;
    SIO_WSTR_ERR_NOT_INITIALIZED    : USINT := 255;


    PORTNUM : USINT := 1;

END_VAR

VAR

    FB_SioInit      : SIO_INIT;
    xInitDone       : BOOL := FALSE;

    strRxText       : STRING(32);      (* set string length := 32 *)
    usiEotChr       : USINT := 16#0D;  (* ==> '$R' = CR *)
    xEcho           : BOOL  := TRUE;
    xEdit           : BOOL  := TRUE;
    FB_SioReadStr   : SIO_READ_STR;
    iRxDataSize     : INT;
    xRdStrConfirm   : BOOL := FALSE;
    xWaitForReceipt : BOOL := FALSE;

    strTxText       : STRING;
    xExpandCR       : BOOL := TRUE;
    xExpandLF       : BOOL := FALSE;
    xAppendLF       : BOOL := TRUE;
    FB_SioWriteStr  : SIO_WRITE_STR;
    xWrStrConfirm   : BOOL := FALSE;
    xTransmitting   : BOOL := FALSE;

END_VAR
```

```
LD      xTransmitting
JMPC    WriteStringCont
LD      xRdStrConfirm
JMPC    WriteStringStart
LD      xWaitForReceipt
JMPC    ReadStringCont
LD      xInitDone
JMPC    ReadStringStart


(* ----- Init Sio ----- *)
SioInit:
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_NO,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
RETCN
LD      TRUE
ST      xInitDone


(* ----- Read String ----- *)
ReadStringStart:
CAL     FB_SioReadStr (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        RXDATA := strRxText,
        PORT := PORTNUM)

LD      FB_SioReadStr.ERROR
EQ      SIO_RSTR_ERR_SUCCESS
RETCN
LD      TRUE
ST      xWaitForReceipt

ReadStringCont:
CAL     FB_SioReadStr (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        RXDATA := strRxText,
        MAXLENGTH := 0,         (* no limit, use whole string length *)
        EOTCHR := usiEotChr,
        ECHO := xEcho,
        EDIT := xEdit,
        PORT := PORTNUM
        |
        xRdStrConfirm := CONFIRM,
        iRxDataSize := RXLENGTH)

LD      xRdStrConfirm
RETCN

LD      strCR
CONCAT  '-> '
CONCAT  strRxText
ST      strTxtext
```

```
(* ----- Write String ----- *)
WriteStringStart:
CAL     FB_SioWriteStr (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        TXDATA := strTxText,
        PORT := PORTNUM)

LD      FB_SioWriteStr.ERROR
EQ      SIO_WSTR_ERR_SUCCESS
RETCN
LD      TRUE
ST      xTransmitting

WriteStringCont:
CAL     FB_SioWriteStr (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        TXDATA := strTxText,
        TXLENGTH := 0,          (* no limit, transmit whole string *)
        EXPANDCR := xExpandCR,
        EXPANDLF := xExpandLF,
        APPENDLF := xAppendLF,
        PORT := PORTNUM
        |
        xWrStrConfirm := CONFIRM)

LD      xWrStrConfirm
RETCN


(* ----- Reset Flow Control Logic ----- *)
LD      FALSE
ST      xTransmitting
ST      xWrStrConfirm
ST      xWaitForReceipt
ST      xRdStrConfirm
RET


END_PROGRAM
```

5.8 Function Block SIO_READ_BIN

The function block *SIO_READ_BIN* reads a binary character stream from the serial interface.

Prototype of the Function Block

```
                +--------------------+
                |    SIO_READ_BIN    |
                |                    |
      BOOL    ---|ENABLE       CONFIRM|--- BOOL
                |                    |
      POINTER ---|PTR_RXDATA          |
      INT     ---|MAXLENGTH   RXLENGTH|--- INT
                |                    |
      USINT   ---|ETXCHR              |
      BOOL    ---|CHKETX              |
                |                    |
      USINT   ---|PORT           ERROR|--- USINT
                |                    |
                +--------------------+
```

Definition of Operands

PTR_RXDATA	Address of an object for receiving the read data bytes
MAXLENGTH	Limitation of number of bytes to read, if 0, the length of the object addressed by PTR_RXDATA is internally determined and used as the number of bytes to be read (there are max. read so much bytes as the object can take up)
EOTCHR	Character for the end delimiter of the binary character stream (only checked if CHKETX = TRUE)
CHKETX	Check of end delimiter character on/off FALSE = end delimiter character is not checked TRUE = check for end delimiter character is activated
RXLENGTH	Number of the read character (if *ERROR := 0*)
ENABLE	Input for enabling or disabling the FB (see text)
CONFIRM	Output for completed message via the FB (see text) FALSE = reception not successfully completed or terminated after error TRUE = reception successfully completed, *RXLENGTH* characters are available in the object addressed by *PTR_RXDATA*
PORT	Number of serial interface to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 20.

Table 20 Error Codes of the Function Block SIO_READ_BIN

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
8	No character received for the end delimiter, reception termination after *MAXLENGHTH* characters
128	Pointer references an object of an unsupported data type
255	The selected interface (*PORT*) is not initialized

Description

The function block reads a binary data stream from the serial interface. The read characters are stored in the object addressed by input *PTR_RXDATA*. If input *CHKETX* is set to TRUE, the data stream read from serial interface is checked for the occurrence of the end delimiter character defined at input *EOTCHR*. On recognition of the defined end delimiter character the reading operations finishes and the function block returns with output *CONFIRM* set to TRUE. If either input *CHKETX* is set to FALSE or the defined end delimiter character doesn't occur in the read binary stream, the the function block stops reading operation if the maximum number of bytes has been received (either internal size of data object addressed by PTR_RXDATA or *MAXLENGTH* characters). Also in this case the output *CONFIRM* is set to TRUE if the function block returns.

Output *RXLENGTH* shows the number of characters stored in the data object addressed by *PTR_RXDATA*. If output *ERROR := 0* has also been set, the character for the end delimiter defined at input *EOTCHR* has been received. If *ERROR := 8,* reception has been terminated after reading the maximum number of characters.

The block starts character reception after detecting a rising edge at input *ENABLE* (first call via *ENABLE := TRUE*). Repeatedly call the function block via the PLC program until character reception (end delimiter or *MAXLENGTH* characters) has been terminated. For this, input *ENABLE* has to be set as TRUE to enable character reception. The block signals successful termination of reception by setting output *CONFIRM* to TRUE. After processing the received character string, the PLC program has to call the block via *ENABLE := FALSE* to internally reset the block to its initial state. Further characters can subsequently be received by resetting input *ENABLE* to TRUE and thus detecting a rising edge. Active reception can be terminated at any time by calling the block via *ENABLE := FALSE*.

Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 18. Due to the simultaneous setting of various bits it is possible to signalize several errors.

The following sample program shows the application of the function block *SIO_READ_BIN* for reading a character string from the serial interface.

Sample Program

The sample program in section 5.9 shows the joint application of *SIO_READ_BIN* and the function block *SIO_WRITE_BIN*. At first, the sample program calls the block *SIO_READ_BIN* to read a binary character stream from the interface. After the character stream has been completely read, the function block *SIO_WRITE_BIN* rewrites it onto the interface.

5.9 Function Block SIO_WRITE_BIN

The function block *SIO_WRITE_BIN* writes a binary character stream onto the serial interface.

Prototype of the Function Block

```
                +---------------------+
                |    SIO_WRITE_BIN    |
                |                     |
      BOOL    ---|ENABLE        CONFIRM|--- BOOL
                |                     |
      POINTER ---|PTR_TXDATA           |
      INT     ---|TXLENGTH             |
                |                     |
                |                     |
      USINT   ---|PORT            ERROR|--- USINT
                |                     |
                +---------------------+
```

Definition of Operands

PTR_TXDATA	Address of an object with the binary data to be sent
TXLENGTH	Number of data bytes to be sent, if the number is 0, the length of the object addressed by *PTR_TXDATA* is internally determined and used as the number of characters to be sent
ENABLE	Input for enabling or disabling the FB (see text)
CONFIRM	Output for completed message via the FB (see text) FALSE = transmission not successfully completed or terminated after error TRUE = transmission successfully completed

PORT Number of serial interface to be used

ERROR The error code states information about the execution result of the function block. Possible error codes are defined in Table 21.

Table 21 Error Codes of the Function Block SIO_WRITE_BIN

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected interface (*PORT*) is not supported
128	Pointer references an object of an unsupported data type
255	The selected interface (*PORT*) is not initialized

Description

The function block writes a binary character stream onto the serial interface. The address of an object with the binary data to be written has to be transferred to input *PTR_TXDATA*. Input *TXLENGTH* specifies the number of valid bytes. If this value is 0, the length of the object addressed by *PTR_TXDATA* is internally determined and used as the number of bytes to be written.

The block starts writing the character string after detecting a rising edge at input *ENABLE* (first call via *ENABLE := TRUE*). Repeatedly call the function block via the PLC program until character transfer has been completed. For this, input *ENABLE* has to be set as TRUE to enable character transmission. The block automatically signals successful termination by setting output *CONFIRM* to TRUE. During further processing, the PLC program has to call the block via *ENABLE := FALSE* to internally reset the block to its initial state. Further character transfer can subsequently be started by resetting input *ENABLE* to TRUE and thus detecting a rising edge. Active transmission can be terminated at any time by calling the block via *ENABLE := FALSE*.

Possible errors during execution of the function block are displayed at output *ERROR* as a bit mask and described in Table 21. Due to the simultaneous setting of various bits it is possible to signalize several errors.

The following sample program shows the application of the function blocks *SIO_READ_BIN* (see section 5.8) and *SIO_WRITE_BIN*. At first, the block *SIO_READ_BIN* is called to read a binary character stream from the interface. After the character stream has been completely read, the function block *SIO_WRITE_BIN* rewrites it onto the interface. This sample program is completed via initialization of the serial interface and the flow control of the program execution.

Sample Program

```
PROGRAM SioRwBin
VAR CONSTANT

    (* Definition of Parity Type *)
    SIO_INIT_PARITY_NO              : USINT :=   0;
    SIO_INIT_PARITY_ODD             : USINT :=   1;
    SIO_INIT_PARITY_EVEN            : USINT :=   2;

    (* Definition of Protocol Type *)
    SIO_INIT_PROTOCOL_NO            : USINT :=   0;
    SIO_INIT_PROTOCOL_XON_XOFF      : USINT :=   1;
    SIO_INIT_PROTOCOL_RTS_CTS       : USINT :=   2;

    (* Error Codes for FB SIO_INIT *)
    SIO_INIT_ERR_SUCCESS            : USINT :=   0;
    SIO_INIT_ERR_HW_ERROR           : USINT :=   1;
    SIO_INIT_ERR_INVALID_PORT       : USINT :=   2;
    SIO_INIT_ERR_INVALID_BAUD       : USINT :=   8;
    SIO_INIT_ERR_INVALID_DATABITS   : USINT :=  16;
    SIO_INIT_ERR_INVALID_PARITY     : USINT :=  32;
    SIO_INIT_ERR_INVALID_STOPBITS   : USINT :=  64;
    SIO_INIT_ERR_INVALID_PROTOCOL   : USINT := 128;


    (* Error Codes for FB SIO_READ_BIN *)
    SIO_RBIN_ERR_SUCCESS            : USINT :=   0;
    SIO_RBIN_ERR_HW_ERROR           : USINT :=   1;
    SIO_RBIN_ERR_INVALID_PORT       : USINT :=   2;
    SIO_RBIN_ERR_NO_EOT_CHAR        : USINT :=   8;
    SIO_RBIN_ERR_ECHO_NOT_SUPPORTED : USINT :=  16;
    SIO_RBIN_ERR_EDIT_NOT_SUPPORTED : USINT :=  32;
    SIO_RBIN_ERR_PTR_TYPE           : USINT := 128;
    SIO_RBIN_ERR_NOT_INITIALIZED    : USINT := 255;


    (* Error Codes for FB SIO_WRITE_BIN *)
    SIO_WBIN_ERR_SUCCESS            : USINT :=   0;
    SIO_WBIN_ERR_HW_ERROR           : USINT :=   1;
    SIO_WBIN_ERR_INVALID_PORT       : USINT :=   2;
    SIO_WBIN_ERR_TXBUFFER_OVERFLOW  : USINT :=   8;
    SIO_WBIN_ERR_PTR_TYPE           : USINT := 128;
    SIO_WBIN_ERR_NOT_INITIALIZED    : USINT := 255;


    PORTNUM : USINT := 1;

END_VAR

VAR

    FB_SioInit      : SIO_INIT;
    xInitDone       : BOOL := FALSE;

    abDataBuffer    : ARRAY[0..127] OF BYTE;
    pDataObject     : POINTER;

    FB_SioReadBin   : SIO_READ_BIN;
    iRxDataSize     : INT;
    xRdBinConfirm   : BOOL := FALSE;
    xWaitForReceipt : BOOL := FALSE;
```

```
    FB_SioWriteBin  : SIO_WRITE_BIN;
    xWrBinConfirm   : BOOL := FALSE;
    xTransmitting   : BOOL := FALSE;

END_VAR

LD      xTransmitting
JMPC    WriteBinCont
LD      xRdBinConfirm
JMPC    WriteBinStart
LD      xWaitForReceipt
JMPC    ReadBinCont
LD      xInitDone
JMPC    ReadBinStart


(* ----- Init Sio ----- *)
SioInit:
CAL     FB_SioInit (
        BAUD := 9600,
        DATABITS := 8,
        PARITY := SIO_INIT_PARITY_NO,
        STOPBITS := 1,
        PROTOCOL := SIO_INIT_PROTOCOL_NO,
        ENABLE := TRUE,
        PORT := PORTNUM)

LD      FB_SioInit.ERROR
EQ      SIO_INIT_ERR_SUCCESS
RETCN

LD      &abDataBuffer
ST      pDataObject

LD      TRUE
ST      xInitDone


(* ----- Read Binary Data Stream ----- *)
ReadBinStart:
CAL     FB_SioReadBin (
        ENABLE := FALSE,         (* Step 1: Reset FB *)
        PORT := PORTNUM)

LD      FB_SioReadBin.ERROR
EQ      SIO_RBIN_ERR_SUCCESS
RETCN
LD      TRUE
ST      xWaitForReceipt

ReadBinCont:
CAL     FB_SioReadBin (
        ENABLE := TRUE,          (* Step 2: Start FB *)
        PTR_RXDATA := pDataObject,
        MAXLENGTH := 0,          (* no limit, use whole object size *)
        CHKETX := FALSE,
        PORT := PORTNUM
        |
        xRdBinConfirm := CONFIRM,
        iRxDataSize := RXLENGTH)

LD      xRdBinConfirm
RETCN
```

```
(* ----- Write Binary Data Stream ----- *)
WriteBinStart:
CAL     FB_SioWriteBin (
        ENABLE := FALSE,        (* Step 1: Reset FB *)
        PORT := PORTNUM)

LD      FB_SioWriteBin.ERROR
EQ      SIO_WBIN_ERR_SUCCESS
RETCN
LD      TRUE
ST      xTransmitting

WriteBinCont:
CAL     FB_SioWriteBin (
        ENABLE := TRUE,         (* Step 2: Start FB *)
        PTR_TXDATA := pDataObject,
        TXLENGTH := 0,          (* no limit, transmit whole object data *)
        PORT := PORTNUM
        |
        xWrBinConfirm := CONFIRM)

LD      xWrBinConfirm
RETCN


(* ----- Reset Flow Control Logic ----- *)
LD      FALSE
ST      xTransmitting
ST      xWrBinConfirm
ST      xWaitForReceipt
ST      xRdBinConfirm
RET


END_PROGRAM
```

6 Access to Hardware Counter

6.1 Application of Hardware Counters

Hardware counters enable the recording of fast, digital signals whose period duration is smaller than the cycle time of the PLC program. Therefore, hardware counters can also recognize and register fast consecutive signal changes. In contrast to hardware counters, software counters, e.g. the standard function blocks *CTU*, *CTD* and *CTUD,* only allow the processing of input signals whose change speed is larger than the cycle time of the PLC program. The function block *CNT_FUD* (Counter for Fast Up Down) enables the configuration of hardware counters for various operating modes (incrementing/reverse counters, counting of rising, falling or of both edges, etc.). At the same time, the block can retrieve current counter readings as well as check whether the limit value has been exceeded in either direction.

6.2 Function Block CNT_FUD

The function block *CNT_FUD* configures the operating mode (counter direction, count edge), retrieves the counter value and checks whether the limit value has been exceeded in either direction.

Prototype of the Function Block

```
                +--------------+
                |   CNT_FUD    |
                |              |
      USINT  ---|MODE          |
                |              |
      BOOL   ---|RESET      QU |--- BOOL
      BOOL   ---|LOAD       QD |--- BOOL
      DINT   ---|PV         CV |--- DINT
                |              |
      USINT  ---|CHANNEL  ERROR|--- USINT
                |              |
                +--------------+
```

Definition of Operands

MODE — Mode selection for the selected channel, the range of values depends on the modes supported by the hardware

0 = disabling of the selected channel, the outputs are reset

Incrementing/Reverse counter, software-controlled:
1 = incrementing counter, rising edge
2 = incrementing counter, falling edge
3 = incrementing counter, both edges

4 = reverse counter, rising edge
5 = reverse counter, falling edge
6 = reverse counter, both edges

Incrementing/Reverse counter, hardware-controlled:
7 = rising edge
8 = falling edge
9 = both edges
Direction control occurs via the respective digital control input:
Control input = 0 → incrementing counter
Control input = 1 → reverse counter

Incrementing/Reverse counter, hardware-controlled:
10 = rising edge
11 = falling edge
12 = both edges
Direction control occurs via the respective digital control input:
Control input = 0 → reverse counter
Control input = 1 → incrementing counter

RESET — The input value TRUE results in the internal counter being reset to zero. The inputs LOAD and PV do not have any influence. No counter pulses are processed as long as the input has the value TRUE. The block changes to the mode selected at input MODE if the edge is falling.

LOAD — The input value TRUE results in the initial value specified at input PV being passed into the counter

PV — The value specified at the input is passed to the counter via LOAD = TRUE. Due to the resulting parity of the current counter reading and the PV, the output QU is set to TRUE

CHANNEL — Channel number of the counter

QU — TRUE: The achieved counter reading is larger than or equal to PV

QD — TRUE: The achieved counter reading is smaller than or equal to zero

CV — Current counter reading

ERROR — The error code states information about the execution result of the function block. Possible error codes are defined in Table 22.

Table 22 Error Codes of the Function Block CNT_FUD

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected channel (*CHANNEL*) is not supported
4	The selected mode (*MODE*) is not supported

Description

The function block configures the mode (counter direction, counting edge, hard or software control), retrieves the counter reading and checks whether the limit value has been exceeded in either direction. The usable block modes depend on the support of the hardware in use. Please see the respective manual of each control for more detailed information.

Selection of the respective mode occurs via input *MODE*. This also includes configuration of the counter direction (incrementing/reverse counter) and the count edges to be processed (rising, falling or both edges). Depending on the hardware in use, it is also possible to use a further digital input for the conversion of the counter direction. This input then functions as the control input for the counter.

Via the inputs *LOAD* and *RESET* the counter can be set to any start value or the current counter reading can be cleared. The internal counter accepts the start value specified at input *PV* as the new counter value if the function block is called via input *LOAD,* which has been set to TRUE. The internal counter reading is reset to zero by calling the function block via input RESET, which has been set to TRUE. The state of input *LOAD* and *PV* is discarded. No counter pulses are processed as long as input *RESET* has the value TRUE. The block changes to the mode selected at input MODE if the edge is falling.

The function block output *CV* states the current counter reading. If a set output displays *QU := TRUE*, the achieved counter reading is larger than or equal to PV (overflow). If a set output displays *QD := TRUE*, the achieved counter reading is smaller than or equal to zero. Both outputs *QU* and *QV* are inactive if the current counter value C*V* is in the interval *0 < CV < PV*.

A specific digital counter input is allocated to each counter channel (please see the manual of the respective control for more information). Depending on the configured mode (input *MODE*), the selection of the counter direction is either implicit, prefixed by the selected mode (software-controlled, *MODE := 1...6*) or flexible to the runtime via the second digital control input (hardware-controlled, *MODE := 7...12*). This has to be considered when using the digital inputs. The current values of the digital counter input as well as, if necessary, of the control input are always stored in the process image of the digital inputs irrespective of the selected counter mode.

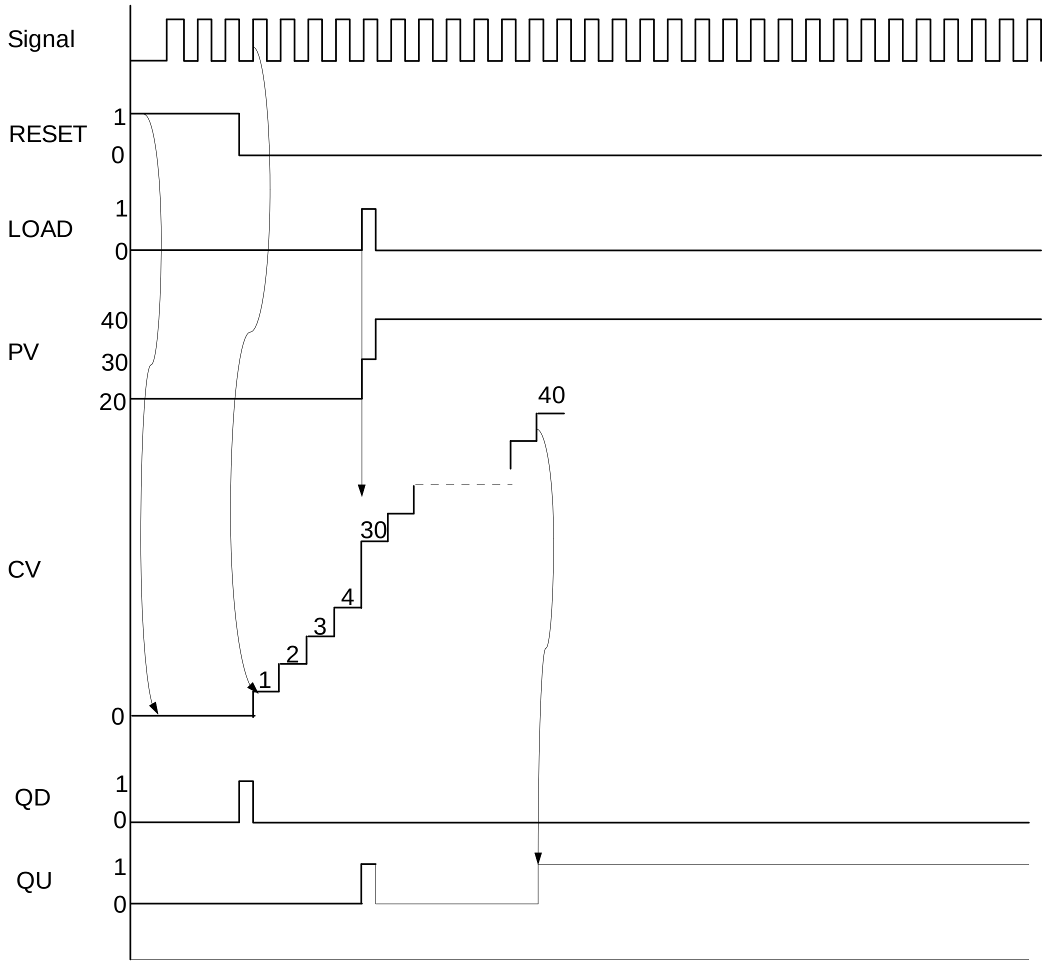

Figure 2: Signal run of the outputs of an incrementing counter

Figure 2: *Signal run of the outputs of an incrementing counter*
illustrates the runs of the individual signals of an incrementing counter (MODE := 1, count rising edge). The inputs LOAD and PV are not considered and the outputs are inactive if RESET := TRUE is active. The selected mode is set and the counter value CV is reset to zero if the edge is falling at input RESET. The value at input PV is passed to CV as the start value if LOAD := TRUE is active. Output QD changes to TRUE as soon as the counter reading is smaller than or equal to zero. Output QU changes to TRUE as soon as the counter reading is larger than or equal to PV.

<u>Sample Program</u>

```
PROGRAM CntDemo

VAR CONSTANT

    (* Error Codes of FB CNT_FUD *)
    CNT_FUD_ERROR_SUCCESS         : USINT := 0;
    CNT_FUD_ERROR_HW_ERROR        : USINT := 1;
    CNT_FUD_ERROR_UNKNOWN_CHANNEL : USINT := 2;
    CNT_FUD_ERROR_INVALID_MODE    : USINT := 4;

END_VAR
```

```
VAR
    xOvflUp      : BOOL;
    xOvflDwn     : BOOL;

    usiProcState : USINT := 0;
    ausiError    : ARRAY[0..3] OF USINT;

    FB_CntFUD   : CNT_FUD;
END_VAR

(* ----- Select current program step ----- *)
LD      usiProcState
EQ      0
JMPC    CounterInit
LD      usiProcState
EQ      1
JMPC    CounterRead
LD      0
ST      usiProcState


(* ----- Init Counter ----- *)
CounterInit:
CAL     CntFUD (              (* Reset Counter *)
        CHANNEL := 0,
        RESET := TRUE
        |
        ausiError[0] := ERROR)

CAL     CntFUD (             (* Set Mode and StartValue *)
        MODE := 1,
        RESET := FALSE,
        LOAD := TRUE,
        PV := 30
        |
        ausiError[1] := ERROR)

CAL     CntFUD (             (* Clear Input LOAD to start Counter *)
        LOAD := FALSE
        |
        ausiError[2] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Read Counter Value ----- *)
CounterRead:
CAL     CntFUD (
        PV := 40
        |
        xOvflUp := QU,
        xOvflDwn := QD,
        ausiError[3] := ERROR)


(* ----- Cycle End ----- *)
ProgExit:
RET

RET
```

7 Access to Real Time Clock (RTC)

7.1 Application of the Real Time Clock (RTC)

The RTC is a special battery-powered hardware block which can operate even when the PLC is switched off. However, very few control units have such a block. The RTC provides a PLC program with the absolute time and current date. This information can, for example, be used to control date and time-dependent processes as well as to log events with a time stamp.

The RTC can be set and the date and time retrieved via the function block *DT_CLOCK* (see section 7.2). The date and time are available at the outputs in absolute form (year/month/day, hour/minute/second) as well as in relative form (seconds since 01.01.1980). The function block *DT_ABS_TO_REL* converts the absolute time and date into the corresponding relative presentation (see section 7.3). The relative presentation simplifies arithmetic operations, e.g. the calculation of time differences or setting of a new switch time to easy subtraction or addition of integer UDINT variables. If required, the function block *DT_REL_TO_ABS* subsequently converts the calculated result back into the absolute form (see section 7.4).

7.2 Function Block DT_CLOCK

The function block *DT_CLOCK* sets the RTC and reads the date and time from the PLC’s RTC. This block is only available on controls which are equipped with an RTC block.

Prototype of the Function Block

```
             +------------------+
             |     DT_CLOCK     |
             |                  |
     UINT ---|SET_YEAR      YEAR|--- UINT
    USINT ---|SET_MONTH    MONTH|--- USINT
    USINT ---|SET_DAY        DAY|--- USINT
    USINT ---|SET_HOUR      HOUR|--- USINT
    USINT ---|SET_MINUTE  MINUTE|--- USINT
    USINT ---|SET_SECOND  SECOND|--- USINT
             |                  |
             |           RELTIME|--- UDINT
     BOOL ---|SET          ERROR|--- USINT
             |                  |
             |                  |
             +------------------+
```

Definition of Operands

SET_YEAR
SET_MONTH
SET_DAY   year/month/day of the date to be set

SET_HOUR
SET_MINUTE
SET_SECOND   hour/minute/second of the time to be set

SET	TRUE: The date at inputs *SET_YEAR*, *SET_MONTH* and *SET_DAY* as well as the time at inputs *SET_HOUR*, *SET_MINUTE* and *SET_SECOND* are written in the RTC when the block is called. At the same time, the set date and time can be read at the respective outputs. FALSE: Only the current time and date from the RTC are read when the block is called. The RTC is not reset.
YEAR MONTH DAY	year/month/day of the read date
HOUR MINUTE SECOND	hour/minute/second of the read time
RELTIME	Relative form of the read date and time (seconds since 01.01.1980)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 23.

Table 23 Error Codes of the Function Block DT_Xxx

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
4	Invalid mode (*MODE*) during function block call
8	Power failure, read time is invalid (see text)
16	Passed absolute time and date are invalid

<u>Description</u>

If the function block is called via input *SET* which has been set to TRUE, the date (*SET_YEAR*, *SET_MONTH* and *SET_DAY*) and time (*SET_HOUR*, *SET_MINUTE* and *SET_SECOND*) at the respective inputs are passed to the PLC's RTC. At the same time, the set date and time set can be read at the respective outputs. However, if the function block is called via input *SET* which has been set to FALSE, only the current date and time are read but the RTC is not influenced. The values of the set inputs are discarded. Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 23.

If the output is *ERROR* = 3 after execution of the function block *DT_CLOCK,* the power supply of the RTC has been interrupted (power failure, battery empty) and the read time is invalid. This error state remains until the PLC's RTC is reset (function block call via input *SET := TRUE*) or the PLC is reset via the reset switch.

The following sample program shows the application of the function block *DT_CLOCK* for setting and reading the RTC.

Sample Program

```
PROGRAM RtcTest

VAR
        Year : UINT;
        Month : USINT;
        Day : USINT;
        Hour : USINT;
        Minute : USINT;
        Second : USINT;
        RelTime : UDINT;

        ErrorCode : ARRAY [0..1] OF USINT;

        FB_DtClock : DT_CLOCK;
END_VAR

LD  0
ST      ErrorCode[0]
ST      ErrorCode[1]

(* setup RTC with new time/date *)
CAL     FB_DtClock (
        SET_YEAR := 2003,
        SET_MONTH := 8,
        SET_DAY := 6,
        SET_HOUR := 12,
        SET_MINUTE := 3,
        SET_SECOND := 0,
        SET := TRUE
        |
        Error[0] := ERROR)

(* read absolute and relative time from RTC *)
CAL     FB_DtClock (SET :=FALSE)

LD      FB_DtClock.YEAR
ST      Year
LD      FB_DtClock.MONTH
ST      Month
LD      FB_DtClock.DAY
ST      Day
LD      FB_DtClock.HOUR
ST      Hour
LD      FB_DtClock.MINUTE
ST      Minute
LD      FB_DtClock.SECOND
ST      Second
LD      FB_DtClock.RELTIME
ST      RelTime
LD      FB_DtClock.ERROR
ST      ErrorCode[1]

RET

END_PROGRAM
```

7.3 Function Block DT_ABS_TO_REL

The function block *DT_ABS_TO_REL* converts an absolute time and date into the corresponding relative presentation (seconds since 01.01.1980). The relative presentation simplifies arithmetic operations, e.g. the calculation of time differences or setting of a new switch time to easy subtraction or addition of integer UDINT variables.

Prototype of the Function block

```
             +-----------------+
             |  DT_ABS_TO_REL  |
             |                 |
     UINT ---|YEAR             |
    USINT ---|MONTH            |
    USINT ---|DAY              |
    USINT ---|HOUR             |
    USINT ---|MINUTE           |
    USINT ---|SECOND           |
             |                 |
             |          RELTIME|--- UDINT
             |            ERROR|--- USINT
             |                 |
             +-----------------+
```

Definition of Operands

YEAR MONTH DAY	year/month/day of the date to be converted
HOUR MINUTE SECOND	hour/minute/second of the time to be converted
RELTIME	Relative form of the converted date and time (seconds since 01.01.1980)
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 23.

Description

If the function block is called, the date (*YEAR*, *MONTH* and *DAY*) and time (*HOUR*, *MINUTE* and *SECOND*) at the respective inputs are converted from the absolute time and date to the respective relative presentation (seconds since 01.01.1980). The relative presentation simplifies arithmetic operations, e.g. the calculation of time differences or setting of a new switch time to easy subtraction or addition of integer UDINT variables. If required, the function block *DT_REL_TO_ABS* subsequently converts the calculated result back into the absolute form (see section 7.4). Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 23.

Sample Program

```
PROGRAM DtConv1

VAR
        RelTime : UDINT;
        ErrorCode : USINT;

        FB_DtAbsToRel : DT_ABS_TO_REL;
END_VAR
```

```
CAL     FB_DtAbsToRel(
        YEAR := 2003,
        MONTH := 7,
        DAY := 23,
        HOUR := 15,
        MINUTE := 10,
        SECOND := 20
        |
        Error := ERROR)

LD      FB_DtAbsToRel.RELTIME
ST      RelTime
RET

END_PROGRAM
```

7.4 Function Block DT_REL_TO_ABS

The function block *DT_REL_TO_ABS* converts a relative time and date (seconds since 01.01.1980) into the respective absolute form (year/month/day, hour/minute/second).

Prototype of the Function block

```
            +----------------+
            |  DT_REL_TO_ABS  |
            |                |
   UDINT ---|RELTIME      YEAR|--- UINT
            |            MONTH|--- USINT
            |              DAY|--- USINT
            |             HOUR|--- USINT
            |           MINUTE|--- USINT
            |           SECOND|--- USINT
            |                |
            +----------------+
```

Definition of Operands

RELTIME — Relative form of the date and time to be converted (seconds since 01.01.1980)

YEAR
MONTH
DAY — year/month/day of the converted date

HOUR
MINUTE
SECOND — hour/minute/second of the converted time

Description

If the function block is called, the relative time at input *RELTIME* (seconds since 01.01.1980) is converted into the corresponding absolute presentation and made available at the respective outputs for date (*YEAR*, *MONTH* and *DAY*) and time (*HOUR*, *MINUTE* and *SECOND*). The function block *DT_ABS_TO_REL* calculates a relative time and date from the absolute form (see section 7.3).

Sample Program

```
PROGRAM DtConv2

VAR
        Year : UINT;
        Month : USINT;
        Day : USINT;
        Hour : USINT;
        Minute : USINT;
        Second : USINT;

        RelTime : UDINT := 743440220;     (* = 23.07.2003, 15:10:20 *)

        FB_DtRelToAbs : DT_REL_TO_ABS;
END_VAR

CAL     FB_DtRelToAbs (RELTIME := RelTime)

LD      FB_DTRelToAbs.YEAR
ST      Year
LD      FB_DTRelToAbs.MONTH
ST      Month
LD      FB_DTRelToAbs.DAY
ST      Day
LD      FB_DTRelToAbs.HOUR
ST      Hour
LD      FB_DTRelToAbs.MINUTE
ST      Minute
LD      FB_DTRelToAbs.SECOND
ST      Second

RET

END_PROGRAM
```

8 Access to the Pulse Generator (PWM/PTO)

8.1 Application of the Pulse Generator (PTO/PWM)

The pulse generator (**PTO** = **P**ulse **T**imer **O**utput / **PWM** = **P**ulse **W**idth **M**odulation) enables the generation of one-time pulse trains (PTO mode) as well as continuous pulse trains (PWM mode). Examples of application are the low-loss power control of ohmic loads such as heating rods or lamps (PWM mode) as well as the control of stepper motors with single pulse trains (PTO mode). The function block *PTO_PWM* enables the application of pulse generators in PTO and PWM mode with direct generator parameterization. The function block *PTO_TAB* enables the definition of complex single impulse trains as a parameter table which is, e.g., required for the realization of ramp functions for stepper motor control.

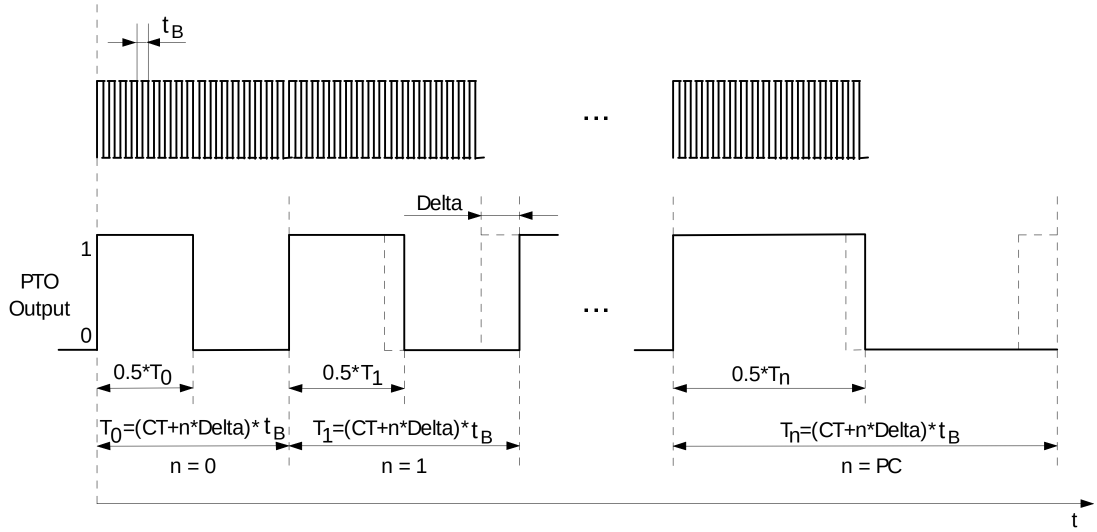

Figure 3: Runtime performance of the pulse generator in PTO mode

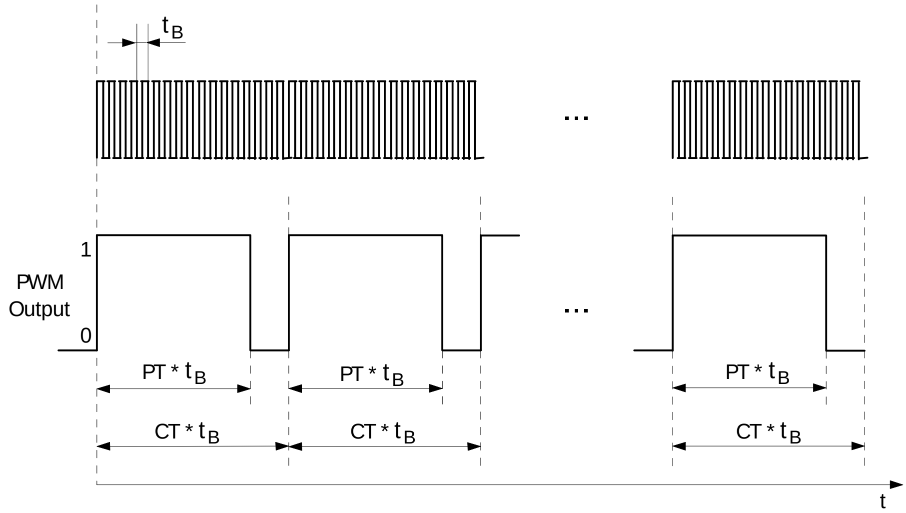

Figure 4: Runtime performance of the pulse generator in PWM mode

Figure 3: *Runtime performance of the pulse generator in PTO mode*
illustrates the runtime performance of the pulse generator in PTO mode and Figure 4: *Runtime performance of the pulse generator in PWM mode*
displays the run-time performance of the pulse generator in PWM mode.

8.2 Function Block PTO_PWM

The function block *PTO_PWM* directly parameterizes the pulse generator in PTO (pulse train output) and PWM mode (pulse duration output). This block is only available for controls which have PWM outputs.

Prototype of the Function block

```
                +-----------------+
                |     PTO_PWM     |
                |                 |
      BOOL   ---|ENABLE      READY|--- BOOL
                |             BUSY|--- BOOL
      BOOL   ---|PTO_MODE         |
      BOOL   ---|APPEND           |
                |                 |
      USINT  ---|TB_IDX           |
      UINT   ---|CT               |
      UINT   ---|PT               |
      INT    ---|DELTA            |
      UDINT  ---|PC               |
                |                 |
      USINT  ---|CHANNEL     ERROR|--- USINT
                |                 |
                +-----------------+
```

Definition of Operands

PTO_MODE	Mode selection TRUE = PTO generator (pulse counter output, one-time pulse train) FALSE = PWM generator (pulse duration output, continuous pulse train) Mode input change at input *ENABLE := TRUE* results in the termination of the previously set function
APPEND	Control input for appending a parameter set TRUE = the currently configured parameters are accepted as a further parameter set FALSE = only the block’s status outputs are updated, the configured parameters are discarded
TB_IDX	Index for setting the base cycle for the pulse generator, this parameter depends on the properties of the respective control, valid values are, e.g.: 0 = 800ns base cycle 1 = 1ms base cycle The base cycle is only accepted with the rising edge at input *ENABLE*.
CT	PTO mode: Period duration PWM mode: Cycle time Period duration or cycle time depend on the specified base cycle at input *TB_IDX* TB_IDX := 0 : 125 ... 65535 (100µs - 52428 µs) TB_IDX := 1 : 2 ... 65535 (2ms - 65535ms)
PT	PTO mode: Not used PWM mode: Pulse duration, range of values: 0 .. 65535

DELTA — PTO mode: Period duration change between two pulses, range of values: -32768 ... +32767
PWM mode: Not used

PC — PTO mode: Number of pulses, range of values: 1 ... 4294967295
PWM mode: Not used

ENABLE — Enable or disable the pulse generator
TRUE = Activation of the pulse generator; the generator accepts the control of the allocated digital output
FALSE = Deactivation of the pulse generator; the process image controls the allocated digital output (the PLC program directly influences the output)
With the rising edge at input *ENABLE*, the function block accepts the index for setting the base cycle (input *TB_IDX*).

READY — Status output of the pulse generator
TRUE = the pulse generator has been fully parameterized, the generator is ready for operation
FALSE = the pulse generator has not been parameterized or the block was terminated with an error, the generator is not ready for operation

BUSY — Status output of the pulse generator
TRUE = the pulse generator is active (pulse train is being generated); the generator controls the digital output
FALSE = the pulse generator is inactive (pulse train completed); the process image controls the digital output (the PLC program directly influences the output)

CHANNEL — Number of the channel to be used

ERROR — The error code states information about the execution result of the function block. Possible error codes are defined in Table 24.

Table 24 Error Codes of the Function Block PTO_PWM

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected channel (*CHANNEL*) is not supported
8	The selected index for the base cycle (*TB_IDX*) is not supported
16	Overflow error for recalculation of the period duration taking *DELTA* into account (period duration is larger than 65535 or smaller than 0)
32	There is no space available in the data record buffer, the data record has been discarded

<u>Description</u>

The function block enables direct parameterization of the pulse generator in PTO (pulse train output) and PWM mode (pulse duration output). The mode pulse output is an alternative function of the digital outputs. If the input is *ENABLE := FALSE*, the process image influences the respective digital output. If *ENABLE := TRUE,* the pulse generator controls the output.

PTO generator (PDO_MODE := TRUE, pulse counter output, one-time pulse train):
The PTO generator creates a one-time pulse train to control the digital output. The pulse train is described by a parameter set which consists of the period duration (initial value), Delta of the period duration (value of the change between two subsequent pulses) as well as the number pulse to be generated. The pulse width is set to 50% of the period duration (sampling ratio 1:1). Taking DELTA into account, the period duration T_n is calculated as follows (also see Figure 3):

$T_n = (CT + n*DELTA) * t_B$ (with 0 <= n <= PC)

If t_B = 1 ms (*TB_IDX := 1*) and *CT := 1000,* the initial period duration (n = 0) is:
T_n = 1 ms * 1000 = 1 second

Via the function block it is possible to string together pulse trains with different values for period duration, Delta (change of period duration) and the number of pulses. For this, call the block for each parameter set to be appended via *APPEND := TRUE*. This way, up to 255 parameter sets can be appended. The block acknowledges the definition of further parameter sets with *ERROR := 32* (no space available in the data record buffer, the data record has been discarded). All parameter sets are based on the same time base. The time base (input *TB_IDX*) can only be changed if the pulse generator has been deactivated. The base cycle index is only passed with the rising edge at input *ENABLE*. If a pulse train has been transmitted completely and no further parameter set is available, the PTO generator automatically switches off and the process image once again controls the digital output. Therefore, the PLC program has to store the desired digital output state after generator deactivation in the process image. The PTO generator also switches off automatically if an overflow or underflow error occurs during calculation of the period duration for the subsequent pulse. This is the case if T_n (see above) is larger than 65535 or smaller than 0. The block signalizes this error via *ERROR := 32* (overflow error during recalculation of the period duration). Due to the accumulative inclusion of *DELTA,* this error can only occur after a series of subsequent successful calculations.

PWM generator (PDO_MODE := TRUE, pulse duration output, continuous pulse train):
In the PWM generator function, a continuous pulse train is generated at the digital output. Here, the period duration as well as the pulse duration can be set as the number of base cycles. If the input is *ENABLE := TRUE*, the PWM generator is directly activated after parameter passing. If the value for pulse duration *PT* is 0, the respective output remains inactive during the entire period duration. But if the value for pulse duration *PT* is larger than or equal to the period duration, the output is active for the entire period duration. The period duration is always changed asynchronously. The current period is interrupted to accept the new value. The pulse duration is changed synchronously and accepted when the next period duration starts. Call the block with input *APPEND := TRUE* to change parameters. Calling the block via *ENABLE := FALSE* terminates generation of the continuous pulse train. The function block automatically switches off if an error occurs during execution. The block can only be reused after it has been reset via *ENABLE := FALSE*.

Output *READY* signalizes that the block is completely parameterized and thus ready for operation. The parameter *TB_IDX* (index for setting the base cycle for the pulse generator) can no longer be changed (*TB_IDX* is only read with rising edge at input *ENABLE*). Output READY returns to FALSE if the block is called via *ENABLE := FALSE*.

If the function block returns with *BUSY := TRUE,* it signalizes that the generator is active and controls the respective digital output (PTO mode: a parameterized pulse train is transmitted, PWM mode: a continuous pulse train is generated). *BUSY := FALSE* indicates that the generator is inactive and that the PLC program directly influences the respective digital output via the process image.

Possible errors during execution of the function block are displayed at output *ERROR* and described in Table 24.

The following sample program displays the application of the function block *PTO_PWM* for generating one-time pulse trains in PTO mode as well as for continuous pulse trains in PWM mode. Due to the especially selected parameter sets no additional technical or measuring devices are required to observe the pulse trains at the status LED of the PWM output. A cycle in PTO mode is started via a positive edge at output *xStartButtonPto.* The pulse train starts with a pulse of 1 second (CT * TB =

1000 * 1 ms = 1 sec), each subsequent pulse is shortened by 50 ms (Delta = -50). A total of 15 pulses is generated (PC = 15). The PWM mode is started via a positive edge at input *xStartButtonPwm*. The generated pulse train has a period duration of 500 ms (CT * TB = 500 * 1 ms = 0.5 sec -> 2 Hz), the on-time of each pulse is 150 ms (PT * TB = 150 * 1 ms = 150 ms).

Sample Program

```
PROGRAM PtoPwm

VAR CONSTANT

    (* Definition of TimeBase-Index *)
    PTO_TB_IDX_800_US : USINT := 0;     (* TimeBase-Index 800us *)
    PTO_TB_IDX_1_MS   : USINT := 1;     (* TimeBase-Index 1ms   *)

    (* Error Codes of FB PTO_TAB *)
    PTOTAB_ERROR_SUCCESS         : USINT :=  0;
    PTOTAB_ERROR_HW_ERROR        : USINT :=  1;
    PTOTAB_ERROR_UNKNOWN_CHANNEL : USINT :=  2;
    PTOTAB_ERROR_UNKNOWN_TB_IDX  : USINT :=  8;
    PTOTAB_ERROR_DELTA_OVERFLOW  : USINT := 16;
    PTOTAB_ERROR_INVALID_TAB     : USINT := 64;

PTO_PWM_CHANNEL : USINT := 0;
END_VAR

VAR
    xStartButtonPto AT %IX0.0 : BOOL;   (* DI0 at PmC14/phyPS-412 *)
    xStartButtonPwm AT %IX0.1 : BOOL;   (* DI1 at PmC14/phyPS-412 *)
    xPtoPwmOut      AT %QX2.4 : BOOL;   (* P0  at PmC14/phyPS-412 *)

    FB_RTrigPto     : R_TRIG;
    FB_RTrigPwm     : R_TRIG;

    usiPtoTbIdx     : USINT := 1;           (* PTO_TB_IDX_1_MS *)
    uiPtoCt         : UINT  := 1000;
    iPtoDelta       : INT   := -50;
    udiPtoPc        : UDINT := 15;

    usiPwmTbIdx     : USINT := 1;           (* PTO_TB_IDX_1_MS *)
    uiPwmCt         : UINT  := 500;
    uiPwmPt         : UINT  := 150;

    xPtoAppend      : BOOL  := TRUE;
    xPtoReady       : BOOL  := FALSE;
    xPtoBusy        : BOOL  := FALSE;

    xPwmAppend      : BOOL  := TRUE;
    xPwmReady       : BOOL  := FALSE;
    xPwmBusy        : BOOL  := FALSE;

    FB_PtoPwm       : PTO_PWM;
    usiPtoPwmError  : USINT;
END_VAR
```

```
(* ----- Wait for Start ----- *)
WaitForStart:
CAL     FB_RTrigPto (CLK := xStartButtonPto)
LD      FB_RTrigPto.Q
JMPC    StartPtoMode

CAL     FB_RTrigPwm (CLK := xStartButtonPwm)
LD      FB_RTrigPwm.Q
JMPC    StartPwmMode

LD      xPtoBusy
JMPC    RunPtoMode

LD      xPwmBusy
JMPC    RunPwmMode

JMP     ProgExit


(* ----- Run PTO Mode ----- *)
StartPtoMode:
LD      FALSE           (* preset output state, this state is    *)
ST      xPtoPwmOut      (* used when PTO Generator isn't running *)

LD      FALSE           (* reset state flags *)
ST      xPtoReady
ST      xPtoBusy
ST      xPwmReady
ST      xPwmBusy

CAL     FB_PtoPwm (
        ENABLE := FALSE,
        CHANNEL := PTO_PWM_CHANNEL)

CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := TRUE,
        APPEND := xPtoAppend,
        TB_IDX := usiPtoTbIdx,
        CT := uiPtoCt,
        DELTA := iPtoDelta,
        PC := udiPtoPc,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPtoReady := READY,
        xPtoBusy := BUSY,
        usiPtoPwmError := ERROR)

RunPtoMode:
CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := TRUE,
        APPEND := FALSE,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPtoReady := READY,
        xPtoBusy := BUSY,
        usiPtoPwmError := ERROR)

JMP     ProgExit
```

```
(* ----- Run PWM Mode ----- *)
StartPwmMode:
LD      FALSE           (* preset output state, this state is    *)
ST      xPtoPwmOut      (* used when PTO Generator isn't running *)

LD      FALSE           (* reset state flags *)
ST      xPtoReady
ST      xPtoBusy
ST      xPwmReady
ST      xPwmBusy

CAL     FB_PtoPwm (
        ENABLE := FALSE,
        CHANNEL := PTO_PWM_CHANNEL)

CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := FALSE,
        APPEND := xPwmAppend,
        TB_IDX := usiPwmTbIdx,
        CT := uiPwmCt,
        PT := uiPwmPt,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPwmReady := READY,
        xPwmBusy := BUSY,
        usiPtoPwmError := ERROR)

RunPwmMode:
CAL     FB_PtoPwm (
        ENABLE := TRUE,
        PTO_MODE := FALSE,
        APPEND := FALSE,
        CHANNEL := PTO_PWM_CHANNEL
        |
        xPwmReady := READY,
        xPwmBusy := BUSY,
        usiPtoPwmError := ERROR)

JMP     ProgExit


(* ----- Cycle End ----- *)
ProgExit:
RET

END_PROGRAM
```

8.3 Function Block PTO_TAB

The function block *PTO_TAB* indirectly parameterizes the pulse generator in PTO mode (pulse train output) via a parameter set table. This block is only available on controls which have PWM outputs.

Prototype of the Function Block

```
               +-----------------+
               |     PTO_TAB     |
               |                 |
BOOL      ---|ENABLE      READY|--- BOOL
               |              BUSY|--- BOOL
               |                 |
USINT     ---|TB_IDX            |
               |                 |
[...]     ---|TABLE-------TABLE|--- [...]
UINT      ---|RECORDS           |
               |                 |
USINT     ---|CHANNEL     ERROR|--- USINT
               |                 |
               +-----------------+
```

Definition of Operands

TB_IDX	Index for setting the base cycle for the pulse generator, this parameter depends on the properties of the respective control, valid values are, e.g.: 0 = 800ns base cycle 1 = 1ms base cycle The base cycle is valid for all the table parameter sets and is only accepted with the rising edge at input *ENABLE*.
TABLE	Parameter set table, contains the parameter sets of the pulse trains to be generated (see text)
RECORDS	Number of occupied entries in the parameter set table passed to input *TABLE* (see text)
ENABLE	Enable or disable the pulse generator TRUE = Activation of the pulse generator, the parameter set table specified at input *TABLE* is read and the generator controls the allocated digital output FALSE = Deactivation of the pulse generator, the process image controls the allocated digital output (the PLC program directly influences the output) The function block accepts the index for setting the base cycle (input *TB_IDX*) if the edge is rising at input *ENABLE*.
READY	Status output of the pulse generator TRUE = the pulse generator has been fully parameterized, the generator is ready for operation FALSE = the pulse generator has not been parameterized or the block was terminated with an error, the generator is not ready for operation
BUSY	Status output of the pulse generator TRUE = the pulse generator is active (pulse train is generated), the generator controls the digital output FALSE = the pulse generator is inactive (pulse train completed), the process image controls the digital output (the PLC program directly influences of the output)
CHANNEL	Number of the channel to be used
ERROR	The error code states information about the execution result of the function block. Possible error codes are defined in Table 25.

Table 25 Error Codes of the Function Block PTO_TAB

Error Code	Definition
0	No error occurred during execution of the function block
1	Hardware error occurred during execution of the function block
2	The selected channel (*CHANNEL*) is not supported
8	The selected index for the base cycle (*TB_IDX*) is not supported
16	Overflow error for recalculation of the period duration taking *DELTA* into account (period duration is larger than 65535 or smaller than 0)
64	The parameter set passed to input *TABLE* is invalid

<u>Description</u>

The function block enables indirect parameterization of the pulse generator in PTO mode (pulse train output) via a parameter set table. Define the table in the PLC program as follows:

```
PTO_TABLE : ARRAY [0..255] OF PTO_RECORD;
```

PTO_RECORD is globally defined in OpenPCS and has the following composition:

```
    PTO_RECORD : STRUCT
        CT    : UINT;
        DELTA : INT;
        PC    : UDINT;
    END_STRUCT;
```

The definition of the parameters *CT*, *DELTA* and *PC* corresponds to that of the function block *PTO_PWM* (see section 8.2). Adhere to the variable and parameter types according to standard IEC 61131-3. Therefore, always create the parameter set table in the PLC program with 256 available entries (*ARRAY [0..255] OF PTO_RECORD*). Specify the number of data records which have really been configured with valid parameters at output *RECORDS*. All parameter sets are based on the same time base. The time base (input *TB_IDX*) can only be changed if the pulse generator is deactivate.

The block accepts the parameter set table at input *TABLE* if the edge is rising at input *ENABLE* (only the number of parameter sets specified as *RECORDS* are considered) and starts generation of the pulse trains. When the table has been completely processed, the PTO generator automatically switches off and the process image once again controls the digital output. Therefore, the PLC program has to store the desired digital output state after generator deactivation in the process image. The PTO generator also switches off automatically if an overflow or underflow error occurs during calculation of the period duration for the subsequent pulse. This is the case if T_n (see description in section 8.2) is larger than 65535 or smaller than 0 after calculation. The block signalizes this error via *ERROR := 32* (overflow error during recalculation of the period duration). Due to the accumulative inclusion of *DELTA*, this error can only occur after a series of subsequent successful calculations.

The following sample program shows the application of the function block PTO_TAB for generating one-time pulse trains with the help of a parameter set table. The motor drive displayed in *Figure 5: Time diagram for sample program "MotorCtl"*

is simulated with 3 phases (start, continuous running and stop). Due to the especially selected parameter sets no additional technical or measuring devices are required to observe the pulse trains at the status LED of the PWM output. A cycle is started via a positive edge at input xStartButtonPto.

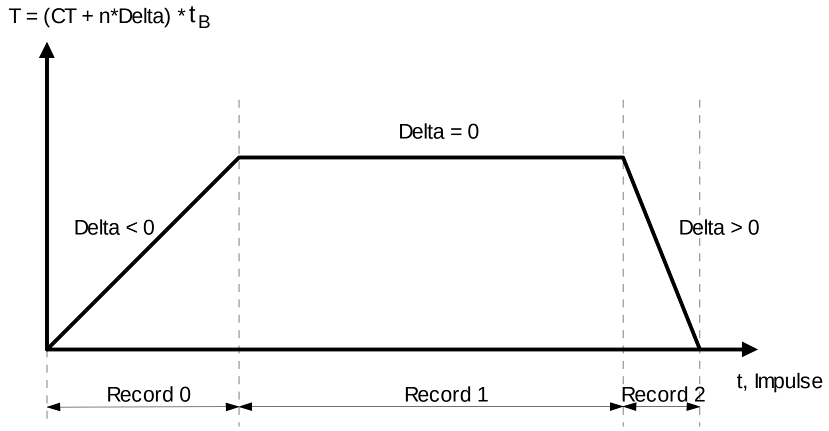

Figure 5: Time diagram for sample program "MotorCtl"

<u>Sample Program</u>

```
PROGRAM MotorCtl
VAR CONSTANT
    (* Definition of TimeBase-Index *)
    PTO_TB_IDX_800_US : USINT := 0;     (* TimeBase-Index 800us *)
    PTO_TB_IDX_1_MS   : USINT := 1;     (* TimeBase-Index 1ms   *)

    (* Error Codes of FB PTO_TAB *)
    PTOTAB_ERROR_SUCCESS          : USINT :=  0;
    PTOTAB_ERROR_HW_ERROR         : USINT :=  1;
    PTOTAB_ERROR_UNKNOWN_CHANNEL  : USINT :=  2;
    PTOTAB_ERROR_UNKNOWN_TB_IDX   : USINT :=  8;
    PTOTAB_ERROR_DELTA_OVERFLOW   : USINT := 16;
    PTOTAB_ERROR_INVALID_TAB      : USINT := 64;

    PTO_CHANNEL : USINT := 0;
END_VAR

VAR
    aPdoTab : ARRAY[0..255] OF PTO_RECORD :=
    [
        (* CT :  UINT   DELTA :  INT    PC :  UDINT *)
        (  CT := 1000,  DELTA := -100,  PC := 9      ),
        (  CT :=  100,  DELTA := 0,     PC := 50     ),
        (  CT :=  100,  DELTA := 200,   PC := 5      )
    ];
    uiRecords       : UINT  := 3;

    usiProcState    : USINT := 0;

    FB_PtoTab       : PTO_TAB;
    ausiError       : ARRAY[0..2] OF USINT;

    xStartButton AT %IX0.0 : BOOL;  (* DI0 at PmC14/phyPS-412 *)
    xMotorOut    AT %QX2.4 : BOOL;  (* P0  at PmC14/phyPS-412 *)

    FB_RTrig : R_TRIG;
END_VAR
```

```
(* ----- Select current program step ----- *)
LD      usiProcState
EQ      0
JMPC    WaitForStart
LD      usiProcState
EQ      1
JMPC    PtoInit
LD      usiProcState
EQ      2
JMPC    PtoSetTab
LD      usiProcState
EQ      3
JMPC    PtoRun
LD      0
ST      usiProcState


(* ----- Wait for Start ----- *)
WaitForStart:
CAL     FB_RTrig (CLK := xStartButton)
LD      FB_RTrig.Q
JMPCN   ProgExit

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Init PTO Generator ----- *)
PtoInit:
LD      FALSE           (* preset output state, this state is    *)
ST      xMotorOut       (* used when PTO Generator isn't running *)

CAL     FB_PtoTab (
        ENABLE := FALSE,
        CHANNEL := PTO_CHANNEL,
        TABLE := aPdoTab,
        RECORDS := 0
        |
        ausiError[0] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Set Table ----- *)
PtoSetTab:
CAL     FB_PtoTab (
        ENABLE := TRUE,
        CHANNEL := PTO_CHANNEL,
        TB_IDX := PTO_TB_IDX_1_MS,
        TABLE := aPdoTab,
        RECORDS := uiRecords
        |
        ausiError[1] := ERROR)

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit
```

```
(* ----- Run PTO Generator ----- *)
PtoRun:
CAL     FB_PtoTab (
        ENABLE := TRUE,
        CHANNEL := PTO_CHANNEL,
        TB_IDX := PTO_TB_IDX_1_MS,
        TABLE := aPdoTab,
        RECORDS := uiRecords
        |
        ausiError[2] := ERROR)
LD      FB_PtoTab.BUSY
JMPC    ProgExit

LD      usiProcState
ADD     1
ST      usiProcState
JMP     ProgExit


(* ----- Cycle End ----- *)
ProgExit:
RET

END_PROGRAM
```

9 Processing of Process Data

9.1 Application of the PID Controller

A controller is used if the output variable of a system cannot be directly controlled by the input variable due to unpredictable disturbance variables. The task of a controller is to monitor the output variable (actual value, process variable PV), to compare it with the command variable (set value, set point SP, error signal = set value – actual value) and to adjust the system input variable via a setting unit (see

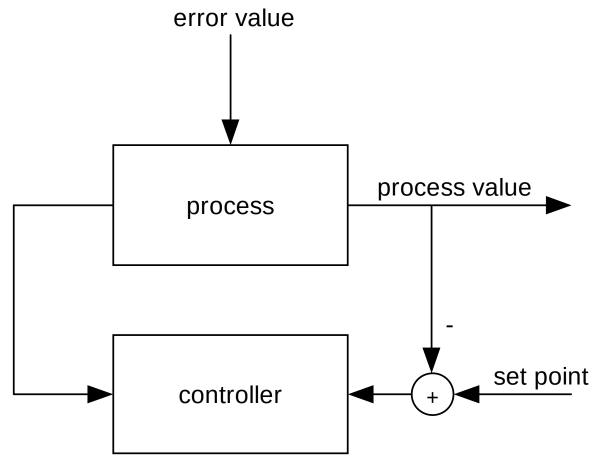

Figure 6: Principle of a control loop

). The result is a new, adjusted output variable. The system is fed-back. An adjustment requires monitoring of the output variable. Therefore, it may be necessary to find or create suitable variables to monitor the system.

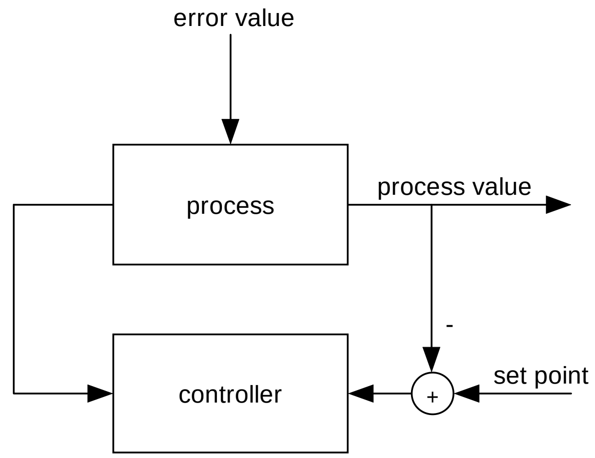

Figure 6: Principle of a control loop

In contrast to this, the known connection between the output variable, disturbance variable and the input variable of a path and the known performance of the disturbance variable does not require observation of the output variable. The output variable can be guided at any time via targeted influence of the input variable according to a set value. In this case, it is known as a control. The system is not fedback.

The function block *PID1* (see section 9.2) calculates the correcting variable CO (controller output) according to the method of the quasi continuous PID control (proportional, integral, derivative controller) from the input values set value SP (set point) and actual value PV (process variable). The properties of the PID controller regarding frequency and phase response are described via its parameters controller gain KR, derivative time TD and reset time TI and sampling period T0.

Call the block at constant intervals during the sampling period T0 to achieve correct functioning of the controller.

Basic Principles (PID Algorithms)

With an analog controller the correcting variable y(t) results from the sum of proportional gain $y_P(t)$, integral gain $y_I(t)$ and derivative gain $y_D(t)$:

$$y(t) = y_P(t) + y_I(t) + y_D(t) + y_0$$

$$y(t) = K_R e(t) + \frac{K_R}{T_I}\int e(t)dt + K_R T_D \frac{de(t)}{dt} + y_0$$

$$e(t) = SP(t) - PV(t)$$

This equation can be passed to a quasi continuous PID controller by sampling the error signal. The integral gain is replaced by a sum of all the error signals, and the derivative gain by a difference of the last two error signals.

$$y(kT0) = K_R[e(kT_0) + \frac{T_0}{T_I}\sum_{n=0}^{k-1} e(kT_0) + \frac{T_D(e(kT_0) - e((k-1)T_0)}{T_0} + y_0$$

Calculation of the integral gain can be simplified even further: the new integral gain values can be determined from the result of the last value plus the new error signal. Therefore, it is not necessary to store all the error signal values since the start of the controller. The sum of all the previous values is called the integral sum and the initial value y_0 of the integral sum is known as bias.

$$y_I(kT0) = K_R \frac{T_0}{T_I} e(kT_0) + y_I((k-1)T_0) + y_0$$

The derivative gain is replaced by the difference of the last two error signals. In order not to create any jumps in the correcting variable when changing the set value, it is supposed for the D gain that the set value between two sampling period points is always constant. Therefore, the calculation of the D gain is restricted to the difference of the last two actual values ($PV(kT_0)$, $PV((k-1)T_0)$).

$$y_D(kT0) = K_R \frac{T_D(PV(kT_0) - PV((k-1)T_0)}{T_0}$$

9.2 Function Block PID1

The function block PID1 realizes a PID controller block according to the control algorithm described in section 9.

Prototype of the Function Block

```
              +--------------+
              |     PID1     |
              |              |
BOOL   ---|ENABLE   READY|--- BOOL
              |              |
REAL   ---|PV         CO |--- REAL
REAL   ---|SP            |
              |              |
REAL   ---|KR            |
TIME   ---|T0            |
TIME   ---|TI            |
TIME   ---|TD            |
REAL   ---|BIAS     ERROR|--- USINT
              |              |
              +--------------+
```

Definition of Operands

PV	Standardized actual value (process variable) of the controlled system valid values range from 0.0 ... 1.0, the block automatically checks this parameter (but not for overflow errors)
SP	Standardized set value (command variable, set point) for the controller valid values range from 0.0 ... 1.0, the block automatically checks this parameter (but not for overflow errors)
KR	Standardized controller gain A positive or negative gain can be selected. With a positive gain the controller displays forward control and reverse control with a negative gain. With a gain of 0, the PID calculation results in a P gain of zero and is therefore not considered. Since the gain is also connected to the I and D gains, a gain of 1 is used for the I and D gains here.
T0	Sampling period of the controller (valid range of values T0 > 0)
TI	Reset time
TD	Derivative time
BIAS	Initial value of the correcting variable or integral sum during the start of the PID calculation (valid range of values 0.0 ... 1.0)
ENABLE	The control parameters KR, T0, TI, TD are accepted and the integral sum set at initial value BIAS if the edge is rising. Outputs READY, ERROR and CO are reset via ENABLE = FALSE. The block checks the area of validity and, if necessary, signals an overflow error at the error output during the transfer of the parameters T0 and BIAS.
CO	Controller output, calculated correcting variable of the controller (range of values 0.0 ... 1.0)

READY Status output of the PID controller
TRUE = the controller block has been completely parameterized and is ready for operation
FALSE = the controller block has not been parameterized or has been incorrectly parameterized (the controller parameters are outside the area of validity), the controller block is not ready for operation

ERROR The error code states information about the execution result of the function block. Possible error codes are defined in Table 26.

Table 26 Error Codes of the Function Block PID1

Error Code	Definition
0	No error occurred during execution of the function block
8	The specified value for the parameter *BIAS* is invalid (smaller than 0 or larger than 1)
16	The specified value for the parameter *T0* is invalid (time equals 0)

Description

The function block realizes a PID controller block according to the control algorithm described in section 9. If the edge is rising at input *ENABLE*, the block accepts the controller parameters *KR*, *T0*, *TD*, *TI* and *BIAS* and starts the controller. The set value and the last actual value are set to the current actual value for the first calculation. The first calculation of the correcting variable always results in the value *BIAS*, since zero is set for the proportional, integral and derivative gain. The performance of the controller can be influenced by the choice of the controller parameters. If the time *TI = t#0ms*, the integral gain of the controller is not calculated and set to zero. If the time *TD = t#0ms*, the value for the derivative gain is zero. If the gain *KR* is zero, the proportional gain is not required. Since the gain *KR* is also linked to the integral and derivative gains, a gain of 1 is used for the I and D gains here.

P controller TI = TD = 0, KR ≠ 0
PI controller TD = 0, KR, TI ≠ 0
PID controller KR, TI, TD ≠ 0

There are various methods for determining the controller parameters (inflection point tangent, oscillation test). The parameters can also be determined via simulation tools. However, in regard to the frequency and phase response, it is necessary that the complete controlled system can be described as a model. A model for the relevant transfer elements of the controlled system can then be created from known values and behavior to determine the parameters during simulation.

The set value, actual value, the bias and the correcting variable are used as standardized variables. The values for the standardized variables range from 0.0 to 1.0. The area of validity for the bias is checked during the start of the block. An error at output *ERROR* signalizes if the value is outside the area. The set value and actual value are not checked due to the runtime. The function block restricts the values of the correcting variable and the integral sum . If the calculation results in a negative value, the variable is set to 0.0. If the result exceeds the value 1, delimiting occurs to 1.0. Furthermore, the integral sum is, depending on the correcting variable, limited according to the following rule:

- If the result of the correcting variable is larger than 1.0, the integral sum is calculated as follows:

  Integral sum = 1.0 - (proportional gain + derivative gain)
- If the result of the correcting variable is larger than 0.0, the integral sum is calculated as follows:

  Integral sum = 1.0 - (proportional gain + derivative gain)

Normalization of the Input Value

During normalization the value range of a variable is mapped onto another number range. An analog input has a value range from 0 ... 32767. This number range has to be mapped onto the value range of the controller (0.0 ... 1.0):

$$y_{nom} = y / 32767$$

The value range to be mapped (-32768 ... +32767) is doubled for a bipolar input variable. Here, the mapping in the positive number range of the standardized variable is considered with an offset of 0.5:

$$y_{nom} = y / 65535 + 0.5$$

Example:

(1) The actual value of the path is accepted with an analog input 0-10V. The current actual value is 7.5V. The actual value is stored as a 15 bit = 7.5 * 32767 / 10 = 24575 in the process image:

$$y_{nom} = y / 32767 = 24575 / 32767 = 0.7499924$$

(2) The actual value of the path is accepted with an analog input ±010V. The current actual value is 7.5V. The actual value is stored as a 15 bit signed = 7.5 * 32767 / 10 = 24575 in the process image:

$$y_{nom} = y / 65535 + 0.5 = 24575 / 65535 + 0.5 = 0.87499$$

(3) The actual value of the path is accepted with an analog input ±010V. The current actual value is -7.5V. The actual value is stored as a 15 bit signed = -7.5 * 32767 / 10 = -24575 in the process image:

$$y_{nom} = y / 65535 + 0.5 = -24575 / 65535 + 0.5 = 0.12501$$

Normalization of the Output Value

Reverse-calculation of output values from the normalized values occurs in reverse order. An analog output has a value range from 0 ... 32767. This number range has to be mapped onto the value range of the controller output (0.0 ...

$$y = y_{nom} \cdot 32767$$

The value range to be mapped (-32768 ... +32767) is doubled for a bipolar output variable. Here, the mapping in the positive number range of the standardized variable is considered with an offset of 0.5:

$$y = (y_{nom} - 0.5) \cdot 65535$$

The following sample program shows the execution of a controlled system via the PLC block-C14. The path is combined of several 1st order transfer elements:

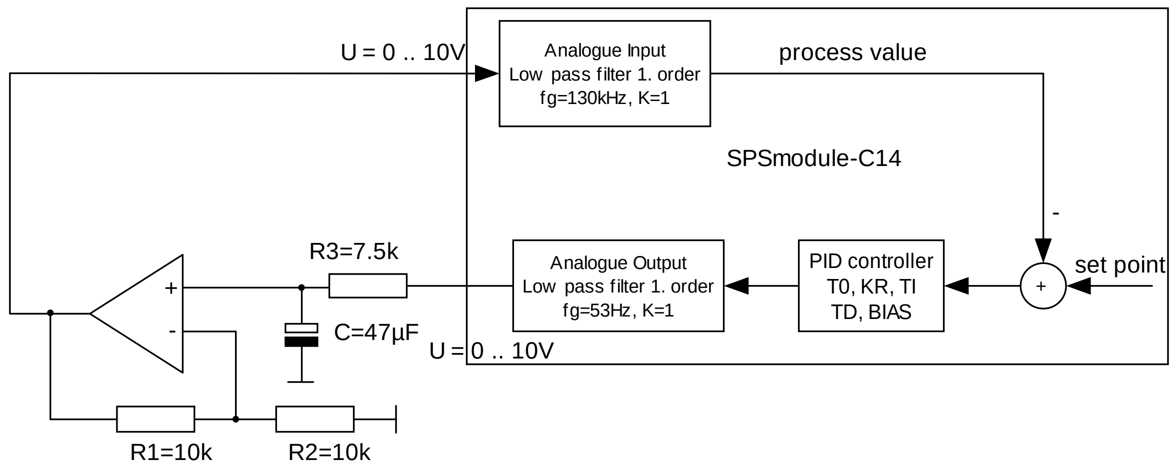

Figure 7: Composition of the controlled system for sample program "PidTest"

The actual value of the path is read via the analog input AI0, the correcting variable is created with the analog output AO0.

<u>Sample Program</u>

```
PROGRAM PidTest

VAR CONSTANT
    (* Error Codes of FB PID1 *)
    PID1_ERR_SUCCESS              : USINT :=   0;
    PID1_ERR_INVALID_BIAS         : USINT :=   8;
    PID1_ERR_INVALID_T0           : USINT :=  16;
END_VAR

VAR_GLOBAL
    (* Prozess Variables *)
    ADC_Result    AT %IW8.0 : UINT;
    ControlOutput AT %QW8.0 : UINT;
END_VAR

VAR
    SetPoint_V   : REAL := 1.0;
    ProcessVar_V : REAL;
    Bias         : REAL;
    FB_PID       : PID1;
    FB_Timer     : TON;
END_VAR


(* to get periodical time stamps start an TON timer         *)
FB_Timer(IN := TRUE, PT := t#25ms);
IF (FB_Timer.Q = FALSE) THEN
      (* the timer intervall is not left                    *)
      RETURN;
END_IF;
```

```
(* The timer intervall is left. Restart the timer for next  *)
(* periode.                                                  *)
FB_Timer(IN := FALSE);
FB_Timer(IN := TRUE, PT := t#25ms);

(*-----------------------------------------------------------*)
(* Prepare calculating PID algorithm                         *)
(* Scale the result of AD converter to a REAL number         *)
ProcessVar_V := UINT_TO_REAL(ADC_Result) * 10.0 / 32767.0;

(*-----------------------------------------------------------*)
(* calculating PID algorithm                                 *)
(* The inputs must scalled by 10.0V                          *)
FB_PID(ENABLE := TRUE,
       PV     := ProcessVar_V  / 10.0,
       SP     := SetPoint_V    / 10.0,
       KR     := 1.5,
       T0     := t#25ms,    (* sample time is 25ms           *)
       TI     := t#20ms,    (* integral time is 20ms         *)
       TD     := t#6ms,     (* derivative time is 6ms        *)
       BIAS   := Bias
      );

(* The result is scalled to unsigned integer value.          *)
ControlOutput := REAL_TO_UINT (FB_PID.CO * 32767.0);
(* The control output is stored to bias to prevent high      *)
(* steps in the reaction curve of controler output if a re- *)
(* start (the PLC was stopped and starts again) is happend. *)
Bias          := FB_PID.CO;
RETURN;

END_PROGRAM
```

Figure 8 illustrates the control effect of the sample program above based on the actual value change during a command variable jump (set value) from 1V to 6V.

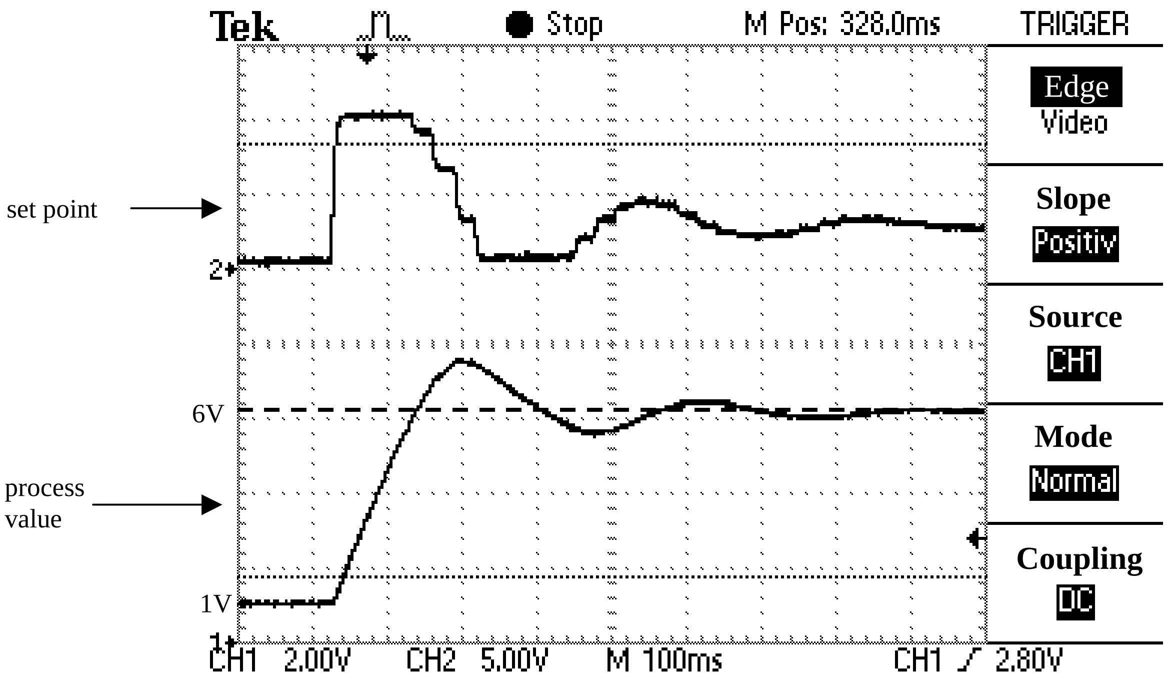

Figure 8: "PidTest" - Change of the actual value during a command variable jump (set value) from 1V to 6V

10 Index

Document:	**SYS TEC Specific Extension for OpenPCS / IEC 61131-3**
Document number:	**L-1054-04, July 2011**

Do you have any suggestions for improving this manual?

Have you discovered any errors in this manual? Page

Sent from:

Customer number:

Name:

Company:

Address:

Send to:

SYS TEC electronic GmbH
August-Bebel-Str. 29
07973 Greiz, Germany
Fax: +49 (0)3661 / 6279-99