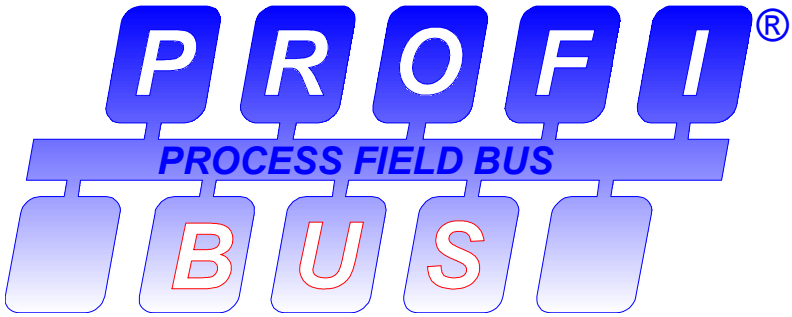
PROFIBUS



PROFIBUS Guideline

PROFIBUS Communication and Proxy Function Blocks acc. to IEC 61131-3

Version 1.2

Juli 2001

PROFIBUS Guideline – Order No. 2.182

PROFIBUS Guideline, Order No. 2.182

PROFIBUS Communication and Proxy Function Blocks acc. to IEC 61131-3

Version 1.20

Juli 2001

Developed by the PROFIBUS Working Group „Communication Function Blocks“ (WG4) in the Technical Committee for „Communication Profiles“ (TC2).

Publisher:
PROFIBUS Nutzerorganisation e.V.
Haid-und-Neu-Str. 7
D-76131 Karlsruhe

Phone: ++ 721 / 96 58 590
Fax: ++ 721 / 96 58 589
PROFIBUS_International@compuserve.com
www.profibus.com

CONTENTS

List of tables

Table of figures

Foreword

For Profibus DP a set of communication services is defined in IEC 61158-5 [1]. The representation of these services in the application program is dependent of the various controllers and devices provided by different manufactures.
State of the art for the programming model and programming languages in the area of the programmable controllers (PLC) is the international standard IEC 61131-3 [2]. This standard defines a set of language elements and mechanisms (e.g. data types, function blocks) which are commonly applied in a well defined set of programming languages (e.g. Ladder Diagram, Structured Text).

Note: The IEC 61131-3 Standard is currently in a revision phase and the updated 2nd Edition of IEC 61131-3 is expected to be finally issued in 2001. Since the vendors of PLCs and corresponding programming tools will need a certain time to implement the new features of the 2nd edition of the IEC Standard this PNO specification restricts the required language features to the content of the 1st Edition of IEC 61131-3.

The application of this specification can provide benefits for the following three groups:

End users
want to implement applications (of programming and devices) using predefined solutions and having a wide choice and independent mixture of various PLCs and field devices (FD) on Profibus.

PLC manufacturers
want to offer the PLC series with a wide choice of field devices (FD) from various manufactures.

Field device manufacturers
wants to have applied his FD easily with a wide choice of minimise the effort to use these FD with different PLCs.

This specification has two parts:

The main part of this specification contains the definitions of a set of *Communication Function Blocks (Comm FB)* for the Profibus DP communication using the IEC 61131-3 programming language standard and the standard Profibus DP services. This set of Comm FB shall be provided by the manufacturers of PLC and intelligent field device for their IEC 61131 programmable controllers.

The second part of this specification presents informative "Guidelines for the application of Communication Function blocks". The "Guidelines" should be applied by field device manufactures to implement specific "*Proxy Function Blocks*" for the integration of their field devices (e.g. weigh control, valve control) in the program of the DP-Master. These *Proxy FB* can be offered to the IEC 61131 application programmers.

1 General

1.1 Scope

This specification defines a set of *Communication Function Blocks (Comm FB)* for communication among programmable controllers and field devices over Profibus DP including DP profiles e.g. Profibus PA. These *Communication Function Blocks* are defined according the international standard for programming languages of PLCs IEC 61131-3.
This specification also gives some "Guidelines for the application of the *Communication Function Block*s", i.e. for implementing field device (DP slave) specific *Proxy Function Blocks* usable in the DP-Master PLC .

1.2 References

The following normative documents contain provisions which constitute provisions of this specifications.

[1] IEC 61158-5: 1999, Digital Data Communications for Measurement and Control - Fieldbus for Use in Industrial Control Systems – Part 5: Application layer service definition

[2] IEC 61158-6: 1999, Digital Data Communications for Measurement and Control - Fieldbus for Use in Industrial Control Systems – Part 6: Application layer protocol specification

[3] IEC 61131-3: 1993, Programmable Controllers, Part 3 Programming languages

[4] PROFIBUS Profile PROFIBUS-PA Version 3.0, October 1999

1.3 Definitions and abbreviations

For the purpose of this specification the following definitions apply. For definitions adopted from other standards the source reference is given.

1.3.1 application program

A self-contained sequence of computer instructions to solve a task. In this specification is a application program a set of function blocks and functions written in a IEC 61131-3 language.

1.3.2 array

IEC 61131-3: An aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. (ISO)

1.3.3 cyclic (exchange of data)

IEC 61158-5: Term used to describe events which repeat in a regular and repetitive manner.

1.3.4 Communication Function Block (Comm FB)

A basic function block defined in this specification and supplied by the PLC manufacturer for the access to field devices.

1.3.5 data type

IEC 61131-3: Set of values together with a set of permitted operation. (ISO)

1.3.6 declaration

IEC 61131-3: The mechanism for establishing the definition of a language element. A declaration normally involves attaching an identifier to the language element, and allocating attributes such as data types and algorithms to it.

1.3.7 DP-Master (Class 1)

IEC 61158-5: A controlling device which controls several DP-Slaves (field devices); usually a programmable controller or distributed control system.

1.3.8 DP-Master (Class 2)

IEC 61158-5: A controlling device which manages configuration data (parameter sets) and diagnosis data of a DP-Master (Class 1). Additionally the DP-Master (Class 2) can perform all communication capabilities of a DP-Master (Class 1)

1.3.9 DP-Slave

IEC 61158-5: A field device that is assigned to one DP-Master (Class 1) as a provider for cyclic I/O data exchange. In addition acyclic functions and alarms could be provided.

1.3.10 European Norm (EN)

The official standard approved and applied by the European countries. Many of the IEC standards were adopted as EN.

1.3.11 field device (FD)

A part of an equipment connected over the field bus and used for a specific function.

1.3.12 function block (FB)

IEC 61131-3: A programmable controller programming language element consisting of: (i) the definition of a data structure partitioned into input, output, and internal variables; and (ii) a set of operations to be performed upon the elements of the data structure when an instance of the function block type is invoked.

1.3.13 function

IEC 61131-3: A program organisation unit which, when executed, yields exactly one data element and possibly additional output variables (which may be multi-

valued, e.g., an array or structure), and whose invocation can be used in textual languages as an operand in an expression.

1.3.14 Human Machine Interface (HMI)

Reading and writing interface for the machine or control equipment operator or shop floor personell to the process data.

1.3.15 identifier

IEC 61131-3: A combination of letters, numbers, and underline characters, as specified in 2.1.2, which begins with a letter or underline and which names a language element.

1.3.16 **instance** (of a function block)

IEC 61131-3: An individual, named copy of the data structure associated with a *function block type* or program type, which persists from one invocation of the associated operations to the next.

1.3.17 index

IEC 61158-5: Address of an object within an application process.

1.3.18 I/O data

IEC 61158-6: Data which have to be transferred cyclically for the purpose of processing.

1.3.19 invocation

IEC 61131-3: The process of initiating the execution of the operations specified in a program organization unit like *function block* and *function*

1.3.20 language element

IEC 61131-3: Any item identified by a symbol on the left-hand side of a production rule in the formal specification given in annex B of IEC 61131-3.

1.3.21 library (of function blocks)

A organised set of function blocks for the use in application programs.

1.3.22 module

IEC 61158-5: Addressable unit inside the DP-Slave.

1.3.23 **parameter** (input and output parameter)

IEC 61131-3: A variable assuming a constant used as an argument to pass in or out a function block or function.

1.3.24 process data

IEC 61158-5: Data which are already pre-processed and transferred acyclically for the purpose of information or further processing.

1.3.25 Proxy Function Block (Proxy FB)

A function block used in the IEC 61131 application program representing a field device or a functional part of a field device.

1.3.26 slot

IEC 61158-5: The address of a module within a DP-Slave.

1.3.27 Structured Text (ST)

IEC 61131-3: The textual PLC programming language using (i) the same common elements as all IEC 61131-3 languages like data types, function blocks and (ii) the specific operators like +, - and (iii) language statements like IF, CASE, WHILE, which are adopted from the well know general purpose languages BASIC or PASCAL.

1.3.28 task

IEC 61131-3: An execution control element providing for periodic or triggered execution of a group of associated program organisation units like function block or funktions.

1.3.29 type (of a function block)

IEC 61131-3: A PLC languages element consisting of: (i) the definition of a data structure partitioned into input, output, and internal variables; and (ii) a set of operations to be performed upon the elements of the data structure when in instance of the function block type is invoked.

1.4 Compliance

This subclause defines the requirements which shall be met by programmable controller systems and field devices which claim compliance with this PNO specification.

The definition of the function blocks in this specification is based on the elements and rules of the "common elements" of IEC 61131-3 (1st edition). Therefore it is required that a programming system which uses the here defined function blocks is compliant to the IEC standard.

This specifications defines a set of Communication Function Blocks (Comm FB) which have defined names, interfaces and functionality. The function block interface comprises the names, the data type and the order of the input and output parameters. The functionality is defined by the state diagram and the associated transition and action table.

A system which claims compliance with this PNO specification shall provide a subset of the here defined of Comm FB. All provided FB shall have the full set of parameters and functionality. The compliant function blocks shall be listed in the "*Compliance Table*" according Annex A.

In a second table also shown in Annex A the permitted "*Implementation dependant features"* shall be listed.

2 Principles for modelling Communication Function Blocks (Comm FB)

2.1 Principles of Modelling

The following principles of modelling for the Communication Function Blocks (Comm FB) have to be met:

- to fit into the existing PLC systems, e.g. using the existent addressing concept
- to be efficient and without overhead; that means the model shall be performance oriented
- to enable a easy application program portation between different PLC systems
- to use directly the existing DP V1 functions, i.e. if possible one Comm FB shall cover one DP service.
- to apply good programming style is to avoid dependencies of the hardware configuration data such as addressing in the application program.

2.2 IEC 61131-3 Function Blocks for Profibus Communication and as Devices Proxies

There are various possibilities in a PLC program (DP Master Class 1) to access to the data in remote modules and devices (DP-Slave) :

- A typical solution in the PLC is the "cyclic access" via the so-called *process image* to the remote inputs and outputs. In the application program these remote variables are used like local I/O variables. The data exchange over the fieldbus happens cyclically. The variables are transfered independently of the execution of the application program and mapped in the process image.
- In this specification a set of *Communication Function Blocks (Comm FB)* is defined like read and write record to achieve a data transfer which is triggered by the application program in the PLC as DP-Master Class 1.

The figure 1 shows an instance of the so called *Proxy Function Block* (*Proxy FB*) representing the field device in the IEC 61131-3 application program in the PLC. This device specific *Proxy FB* exhibits the input and output parameters of the represented field device. Inside the Proxy FB standardised *Communication Function Blocks* (*Comm FB*) provide the reading and writing access to the field device data using the standard Profibus protocols.

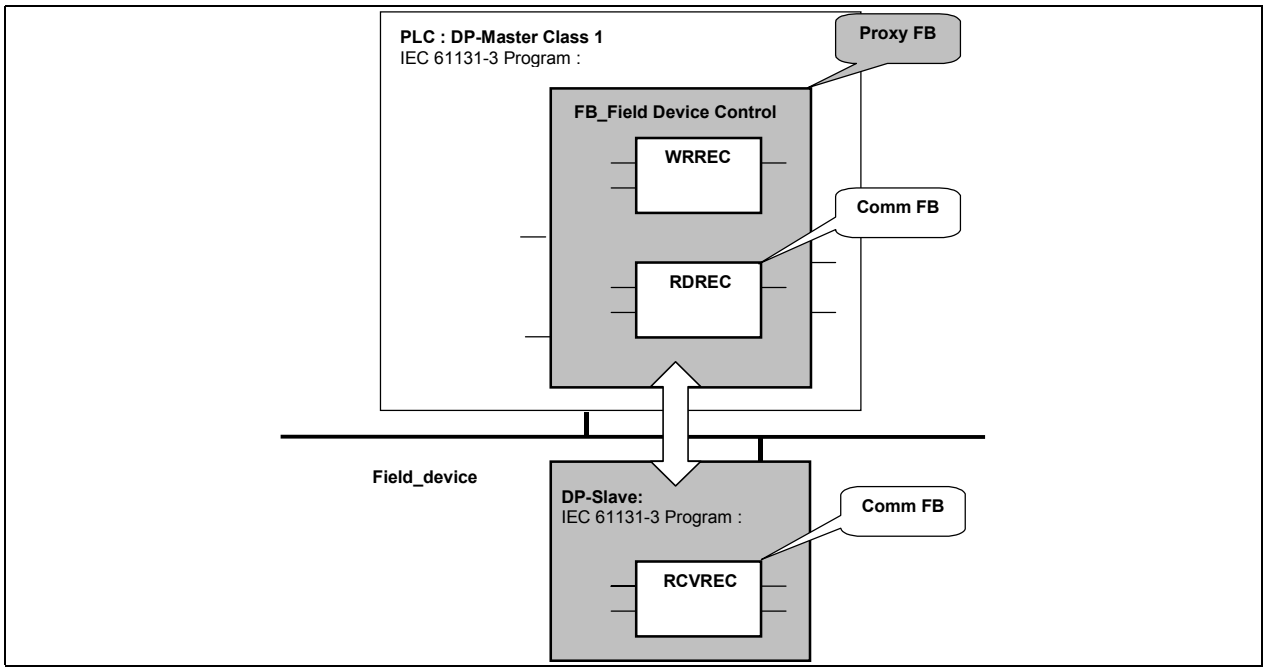


Figure 1 – Proxy FB and Comm FB

2.3 Library Concept and Program Portation

Figure 2 illustrates the usage of libraries of two PLC systems with standardised Comm FB.
The manufacturers of the PLC systems A and B provide their programming systems according IEC 61131-3 and offer both their own library with the Comm FB as defined in this PNO specification. These Comm FB have the identical interface and functionality but are specifically implemented for the different PLC systems.
The Comm FB can be used by the application programmers and also by the field device manufacturers to build the specific Proxy FB.
PLC manufacturers may provide libraries of standardised Proxy FB based on fieldbus profiles and the Comm FB to support application development integrating field device functionality and support access to field device maintenance and diagnose features.

As shown in figure 2 two different FD manufacturers C and D can provide their own libraries with specific Proxy FB C and D for the application support of their field devices connected via Profibus to the PLC systems A and B. The device manufacturers use the standardised Comm FB for the implementation of their Proxy FB runnable in different PLC systems. They can apply the same Proxy FB implementation using the standard Comm FB of the required PLC library.
The application programmers of the PLC systems A and B can use the specific Proxy FB C and D as well as the basic Comm FB.
The application programs using same the IEC 61131 programming language and the standardised Comm FB and Proxy FB can easily ported from PLC system A to B.

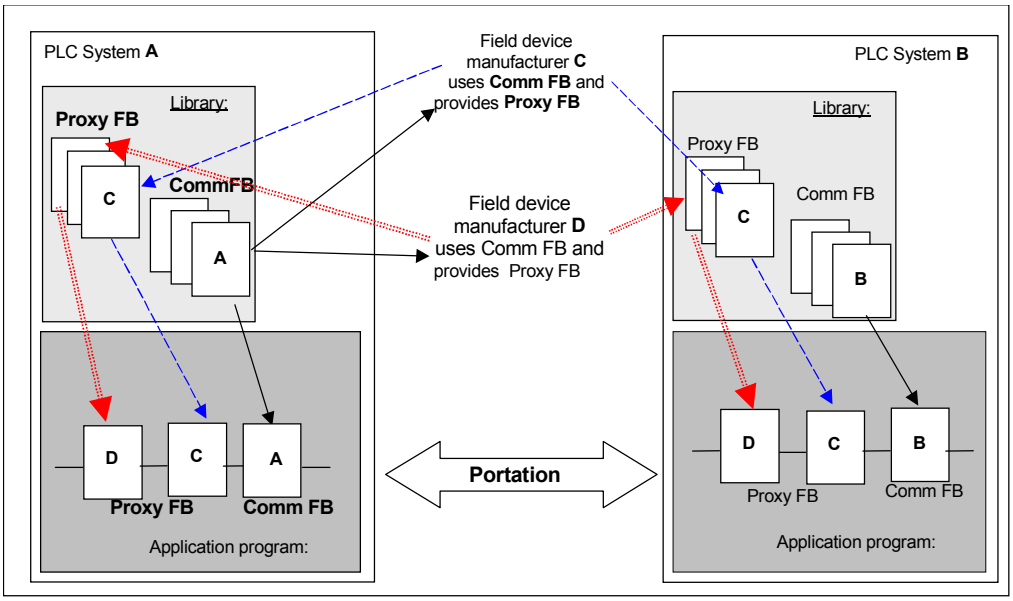


Figure 2 – Usage of Function block libraries with Comm FB and Proxy FB

The guidelines in clause 4 offer additional information to implement and apply of the Comm FB and Proxy FB.

2.4 Mapping to IEC 61158-6

The Communication Function Blocks (Comm FB) map onto objects and services defined in IEC 61158-5 Type 3 (Profibus DP).

2.5 Mapping to Function Blocks

The communication between the application program written in an IEC 61131-3 language and the field devices connected via Profibus DP to the PLC is modelled by Comm FB. This specification defines the Comm FB for the cyclic and acyclic data access.

All Comm FB defined in this specification have a name beginning with “”.
As far as possible and useful the naming and behaviour of all Comm FB are following similar rules.

2.6 FB Parameters

Parameters of different Comm FB with the same or a similar meaning shall have the same name and data type:

Table 1 – Comm FB Parameters

Parameter	Data Type	Meaning
REQ	BOOL	Request function
ID	DWORD	Identification of a DP-Slave or a slot of a DP-Slave
INDEX	INT	Identifier of a process data object (DP data record)
LEN	INT	Actual data length of a DP data record
DONE	BOOL	Flag that the function has finished successfully
VALID	BOOL	Flag that the function has finished successfully and the received output data are valid
BUSY	BOOL	Flag that the function is still performing its task, its not ready to perform a new task
ERROR	BOOL	Flag that the function has finished with an error
STATUS	DWORD	Completion or error code

The I/O data, the process data records or the alarm and diagnosis information is passed by input-output parameters to the Comm FB. Typically these data can be described as an array of byte. The length of the byte arrays may vary from instance to instance of one Comm FB. It shall also be possible to use a structured data type if the used data is structured.

The ANY data type is used to allow the use of byte arrays of different lengths or the use of structured data types as buffers for I/O data, the process data records or the alarm and diagnosis information. The user shall use variables of appropriate size which can contain the information wished. The implementer may cause an error if he can detect that a given variable does not fit to the requested service.

The DONE and the ERROR outputs pulses only from one invocation of the instance of the Comm FB.

The FB parameters shall use those data types of IEC 61131-3 which are supported in a wide range of PLCs and is contained in the portability level of PLCopen. Therefore the data type INT is preferred to the data type USINT, SINT or UINT to represent value ranges of 0..127 or 0..240.

2.7 Address Concept

Addresses shall allow to identify technological functions which communicate with the PLC and its application program via Profibus DP. Typically these technological functions are represented by one slot of a DP-Slave, a continuous series of slots or a whole DP-Slave. Additionally the configured allocation of technological functions to slots of a DP-Slave e.g. in the PA profile shall be supported.

All Comm FB use the same address concept, i.e. the address of one identified functionality can be used with all Comm FB. The application program shall be able to use the Comm FB without knowledge of the explicit hardware configuration e.g. the station number of the DP-Slaves or the position of a slot in a modular slave. It shall be able to use symbolic addressing.

The address concept shall consider existing DP addressing and existing address concepts of PLCs. A PLC may communicated to DP-Slaves which are connected to different DP systems, i.e. the PLC is DP-Master (Class 1) to different DP systems. In IEC 61158-6 a DP-Slave is addressed using a station number unique within the DP system. To address a slot of a DP-Slave an additional slot number is used. All numbers have ranges that do not exceed 0 .. 255, i.e. one DWORD can hold this information.

If a PLC is acting as a DP-Master (Class 2) it can address different DP systems using a segment number. As a DP-Master (Class 1) the segment number is not relevant and set to zero.

The input parameter ID of the Comm FB addresses one slot of a DP-Slave or a DP-Slave. The ID contains a handle of data type DWORD, the value of it is implementor-specific.

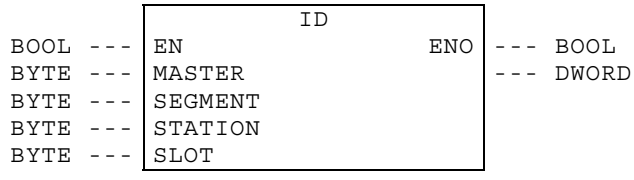
The handle may be generated by local means of the PLC or its configuration system or may be generated by using one of the following functions:

Note: IEC 61131-3 distinguishes between function block and function. The function does not have the instance construct like the function block but it can be used as a simple means to provide a value.

- ID: Conversion of a physical address of a DP-Slave to the handle
- ADDR: Conversion of a handle to the physical address of a DP-Slave
- SLOT: Addressing a slot of a DP-Slave
- NSLOT: Addressing the next slot of a DP-Slave

2.7.1 Function ID

The function ID converts the physical identification of a slot to a handle which can be used with the Comm FB. The slot has a unique slot number within a DP-Slave, the DP-Slave has a unique station number in a DP system, and a DP system is identified by an identification of its master interface. The identification of its master interface may be PLC-specific or unique in the automated system. A DP-Master (Class 2) can use a DP segment number additionally.



Note: According IEC 61131-3 2nd Edition the main function output does not have a formal name in the graphical representation.

```
FUNCTION ID : DWORD    (* Generate DP slot handle *)
VAR_INPUT
  MASTER  : BYTE;      (* ID of the DP system *)
  SEGMENT : BYTE;      (* Number of the DP segment *)
  STATION : BYTE;      (* Number of the DP-Slave (station address) *)
  SLOT    : BYTE;      (* Number of the slot *)
END_VAR
```

Figure 3 – Function ID

The slot number 0 is used to address the DP-Slave interface.

If no slot exists at the given physical address 16#FFFF_FFFF is returned as an error indication. The output ENO shall be false.

2.7.2 Function ADDR

The function ADDR converts a handle which addresses a slot or a DP-Slave into its physical address.

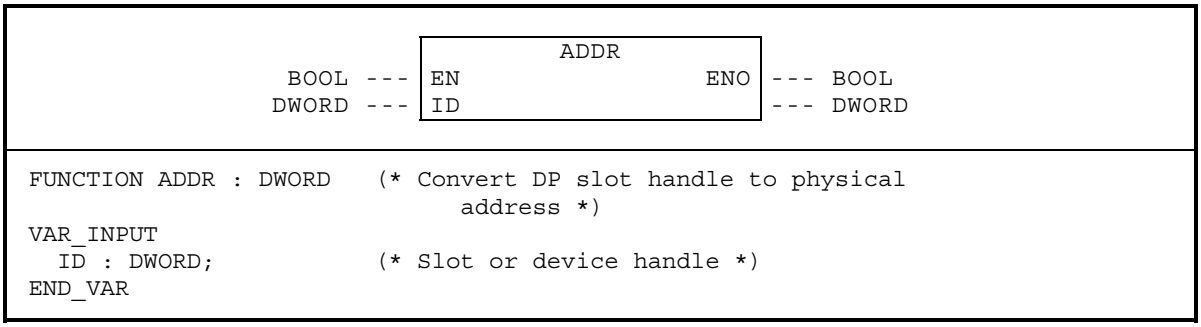


Figure 4 – Function ADDR

The result of the function ADDR is a DWORD which is interpreted as a packed array of four bytes as described in the following table.

NOTE: An output of data type DWORD is used because it is not allowed to use a structured variable as a function result with IEC 61131-3.

Table 2 – Structure of the Output of ADDR

Byte	Name	Definition	Date type
0	master	DP master interface identification	byte
1	segment	segment number	byte
2	station	station number	byte
3	slot	slot number	byte

If the handle of a DP-Slave interface is given, the returned slot number shall be 0.

2.7.3 Function SLOT

The function SLOT provides the handle of a slot identified by its number to a given DP-Slave.

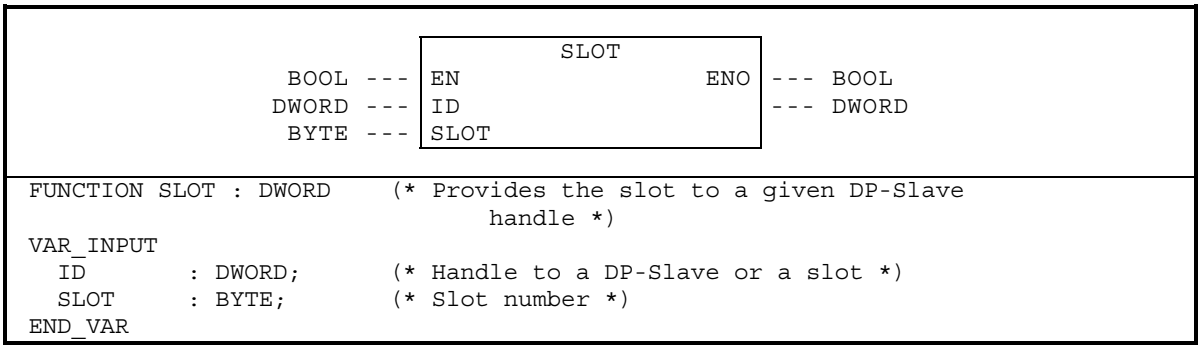


Figure 5 – SLOT

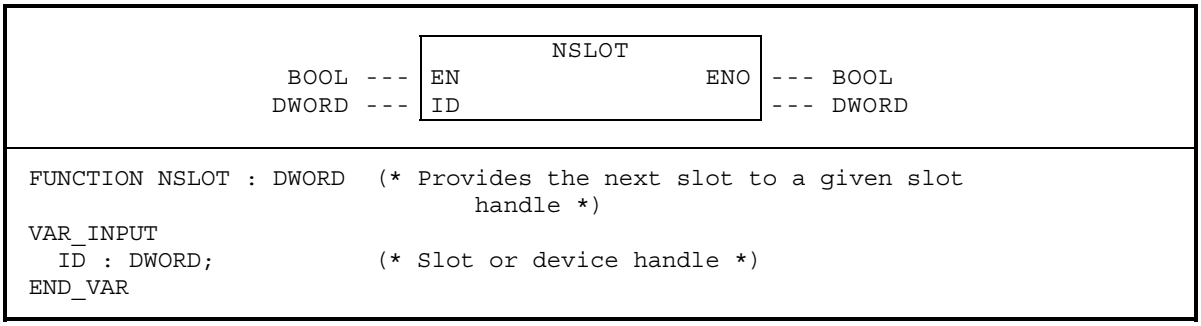
If the handle of a DP-Slave is given, the returned handle addresses the slot identified by the number in the SLOT input of this DP-Slave. If the handle at input ID addresses a slot of a DP-Slave the same handle is returned as if the DP-Slave is

addressed at the input ID. If a slot with this number does not exist 16#FFFF_FFFF is returned as an error indication. The output ENO shall be false.

NOTE: If an implementor uses the physical address in its handle, these functions may easily be implemented using logic and arithmetic expressions.

2.7.4 Function NSLOT

The function NSLOT provides the next slot to a given slot of a DP-Slave.




```
FUNCTION NSLOT : DWORD  (* Provides the next slot to a given slot
                           handle *)
VAR_INPUT
  ID : DWORD;           (* Slot or device handle *)
END_VAR
```

Figure 6 – Function NSLOT

If the handle of the DP-Slave interface is given, the returned slot shall be the first slot of the DP-Slave. If no next slot exists 16#FFFF_FFFF is returned as an error indication. The output ENO shall be false.

2.8 Error Concept

Communication function blocks indicate if the requested function (block) was performed successfully or not. The error indication is typically used for two purposes:

1. To change the reaction to the process i.e. to implement a substitute reaction e.g. to repeat the request at another time or another place or to abort the process task.
2. To issue an alarm message to an HMI system by the application program or by the PLC system automatically.

NOTE: In case 1 only very few different reactions dependent on the indicated error are typical. Detailed error information is hardly used.

If the communication function block maps directly to one service primitive of IEC 61158-6 the error indications of the used service primitive is used as error indication of the function block too. The Function_Num byte, Error_Decode byte, Error_Code_1 byte, and Error_Code_2 byte of the DP service primitives are combined to the STATUS output.

The STATUS output has the data type DWORD which is interpreted as a packed array of four bytes as described in the following table.

Table 3 – Structure of the Output STATUS

Byte	Name	Definition	Date type
0	Function_Num	contains Function_Code / Error_Code, PDU_Identifier, and Frame_Selector	byte
1	Error_Decode	defines the meaning of Error_Code_1 and Error_Code_2, see Table 4	byte

Byte	Name	Definition	Date type
2	Error_Code_1	see Table 5	byte
3	Error_Code_2	implementor specific	byte

NOTE: The error code 2 should be used only for the purpose to detail an error defined with error decode byte and error code 1 byte e.g. for the use of an protocol analyser or an other diagnosis device.

The Function_Num byte is used as defined in IEC 61158-6. The value 16#40 shall be used, if no DP V1 protocol element is used.

The Error_Decode byte defines the meaning of Error_Code_1 and Error_Code_2.

Table 4 – Error_Decode values

Error_Decode	Source	Meaning
16#00 .. 16#7F	PLC	No error or warning
16#80	DP V1	Error reported according to IEC 61158-6
16#81 .. 16#8F	PLC	18#8x reports an error according the x-th parameter of the call of the Comm FB
16#FE .. 16#FF	DP Profile	profile-specific error

The Error_Code_1 defines the reason of the reported error, see Table 5.

Table 5 – Error_Code_1 values

Error_Decode	Error_Code_1	Source	Meaning
16#00	16#00	---	No error and no warning
16#00 .. 16#7F	16#00 .. 16#FF	PLC	Warning
16#80	16#00 .. 16#9F	PLC	Implementer specific
16#80	16#A0	PLC	Read error
16#80	16#A1	PLC	Write error
16#80	16#A2	PLC	Module failure
16#80	16#A3 .. 16#A6	PLC	Implementer specific
16#80	16#A7	PLC	Busy
16#80	16#A8	PLC	Version conflict
16#80	16#A9	PLC	Feature not supported
16#80	16#AA .. 16#AF	PLC	DP-Master specific
16#80	16#B0	Access	Invalid index
16#80	16#B1	Access	Write length error
16#80	16#B2	Access	Invalid slot
16#80	16#B3	Access	Type conflict
16#80	16#B4	Access	Invalid area
16#80	16#B5	Access	State conflict
16#80	16#B6	Access	Access denied
16#80	16#B7	Access	Invalid range
16#80	16#B8	Access	Invalid parameter
16#80	16#B9	Access	Invalid type
16#80	16#BA .. 16#BF	Access	User specific
16#80	16#C0	Resource	Read constrain conflict
16#80	16#C1	Resource	Write constrain conflict
16#80	16#C2	Resource	Resource busy
16#80	16#C3	Resource	Resource unavailable
16#80	16#C4 .. 16#C7	Resource	Implementer specific
16#80	16#C8 .. 16#CF	Resource	User specific
16#80	16#D0 .. 16#FF	User specific	User specific
16#81	16#00 .. 16#FF	PLC	Error concerning the value the 1st parameter
16#82	16#00 .. 16#FF	PLC	Error concerning the value the 2nd parameter
:	:	:	:
16#8F	16#00 .. 16#FF	PLC	Error concerning the value the 15th parameter
16#FE .. 16#FF	16#00 .. 16#FF	Profile	Profile-specific errors

General DP V1 errors shall use the value 16#80 as the value of the Error_Decode byte. Errors which can explicitly mapped to one parameter of the function block code the parameter number in the least significant nibble of the Error_Decode byte, e.g. 16#82xx means that an error was detected for parameter number 2. The parameters are counted beginning with the input parameters starting with 1 and continuing with the output and input/output parameters as defined in the function block declaration.

The outputs at function blocks for the cyclic I/O services e.g. GETIO is set or resetted at every invocation of the function block instance. In this case an error will appear for a longer time in the STATUS output if it is not only temporary. Temporary errors may be treated as irrelevant by the application program.

The outputs at function blocks for a acyclic services e.g. RDREC shows the state of the last requested service. They will stay at the same values until the service is requested with the same function block instance again.

2.9 Representation

The representation of the interface of function block types is given in graphical and textual form according IEC 61131-3. The behaviour of the function blocks is presented as a graphical state diagram with tables for the transitions and the actions.

3 Communication Function Blocks for DP-Master (Class 1)

3.1 General Information

This clause defines the Comm FB for the DP Master (Class 1) programmed in IEC 61131-3.

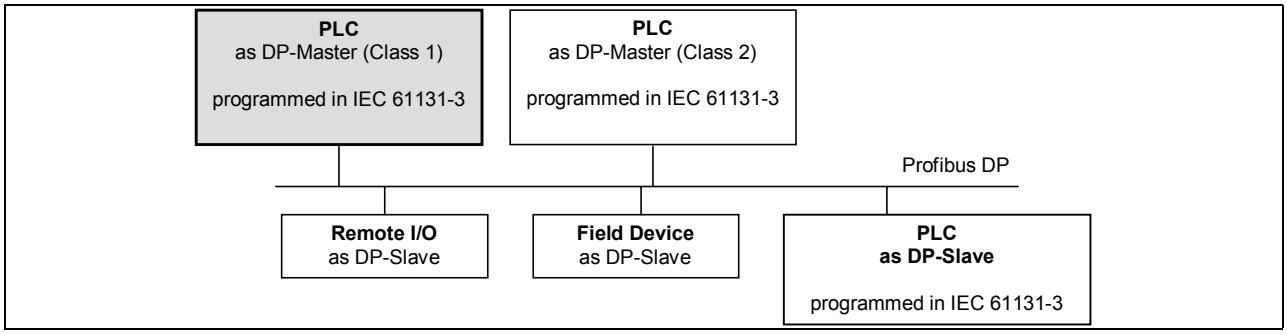


Figure 7 – Profibus system with a PLC as DP-Master (Class 1)

The following function blocks define the application program interface to the basic DP services for a PLC acting as a DP-Master (Class 1):

- GETIO: Get input data of a DP-Slave
- SETIO: Set output data of a DP-Slave
- RDREC: Read a process data record from a slot of a DP-Slave
- WRREC: Write a process data record to a slot of a DP-Slave
- RALRM: Receive an alarm from a DP-Slave
- RDIAG: Read diagnosis information from a DP-Slave
- SYCFR: Synchronise and freeze of a group of DP-Slaves

The following function blocks define a application program interface with higher communication functions using the basic DP services for a PLC acting as a DP-Master (Class 1):

- ICRTL: Request a interlocked control function from a DP-Slave

3.2 Cyclic Exchange of I/O Data

3.2.1 General

The figure below shows the two FBs for cyclic exchange of I/O data.

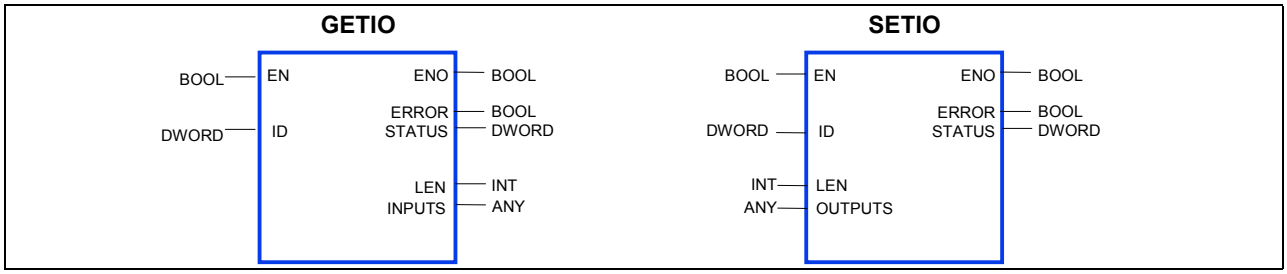


Figure 8 – Communication function blocks for cyclic exchange of data

The function blocks for cyclic exchange of I/O data are used when a PLC is acting as a DP-Master (Class 1).

A DP-Master (Class 1) transfers output data cyclically to its DP-Slaves, in return it gets the input data from the DP-Slave. The I/O data may be accessed by the application program via the %I or %Q areas or be read or written using the function blocks GETIO and SETIO as defined below. Enabling a communication function block for cyclic exchange of I/O data means, that the I/O data is transferred to or from the DP-Master interface.

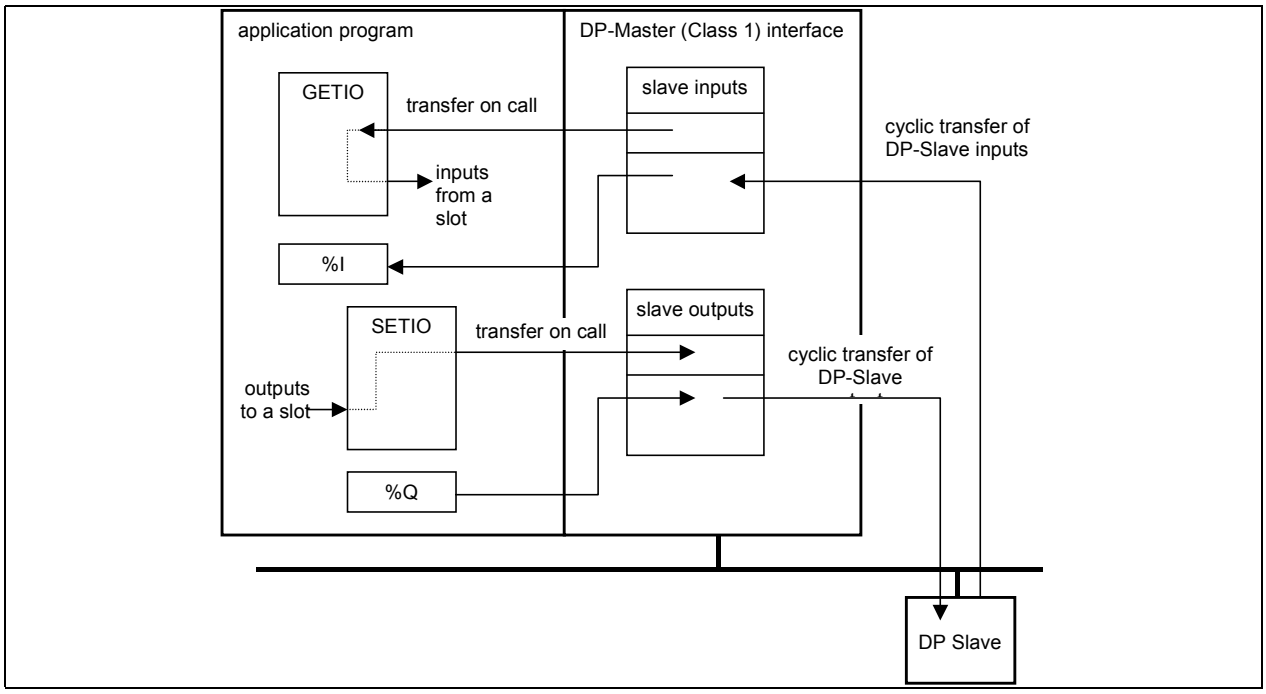


Figure 9 – Communication path for cyclic I/O data

NOTE 1: The same output data should not be written by different function block instances or be written via the %Q interface, because which values are transferred to the slave may be unpredictable.

NOTE 2: Process image vs. direct access, manufacturer dependent

The GETIO function block gets the input data of the addressed slot of a DP-Slave from the DP-Master interface out of the cyclically read input data of the DP-Slave. The SETIO function block transfers the output data of a slot to the DP-Master interface. The DP-Master collects the data of the slots of the DP-Slave and cyclically transfers these outputs to the DP-Slave.

3.2.2 Get I/O Data (GETIO)

The communication function Get I/O Data for a DP-Master (Class 1) uses the GETIO function block defined in this clause. One instance of a GETIO function block provides one instance of the PLC function Get I/O Data. The function is invoked by a 1 of the EN input.

The ID parameter identifies the DP-Slave or the slot of a DP-Slave the I/O data is read from.

NOTE: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

If the input data are valid, the ENO output is set to 1 and the Input data are stored in the variable given at the INPUTS parameter. The variable passed to the INPUTS parameter shall be of appropriate size to receive the input data. An ARRAY[1..244] OF BYTE can hold the data in all cases. The LEN output contains the length of the read input data in byte. The output parameters of this FB are set synchronously.

If a variable at the %I area is referenced at the INPUTS parameter the implementor shall specify the rules using this variable with this parameter.

If an error occurred, the ENO output is set to 0 and the STATUS output contains the error code. The STATUS values are defined in table 3.

```
                                GETIO
                    BOOL --- EN                ENO --- BOOL
                   DWORD --- ID              STATUS --- DWORD
                                               LEN --- INT
                     ANY --- INPUTS--     --INPUTS
```

```
FUNCTION_BLOCK GETIO  (* Get I/O data *)
VAR_INPUT
  EN      : BOOL;        (* Enable *)
  ID      : DWORD;       (* Identifier of a slot of a DP-Slave *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;        (* Flag *)
  STATUS  : DWORD;       (* Last detected status *)
  LEN     : INT;         (* Length of the I/O data *)
END_VAR

VAR_IN_OUT
 INPUTS   : ANY;         (* Input data of the slot *)
END_VAR
```

```
Example: Get inputs of a slot of a DP-Slaves

   VAR D2S3: DWORD;
       I1  : ARRAY [1..20] OF BYTE;
       GET1: GETIO;

   D2S3:= ID(1,0,2,3) (* Handle for Slave 2, Slot 3 at DP system 1 *)
   GET1 (ID:=D2S3, INPUTS:=I1);
   IF STATUS=0 THEN (* process input data *)
   ..
```

Figure 10 – GETIO function block

The following state diagram describes the algorithm of the GETIO function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the GETIO function block outputs.

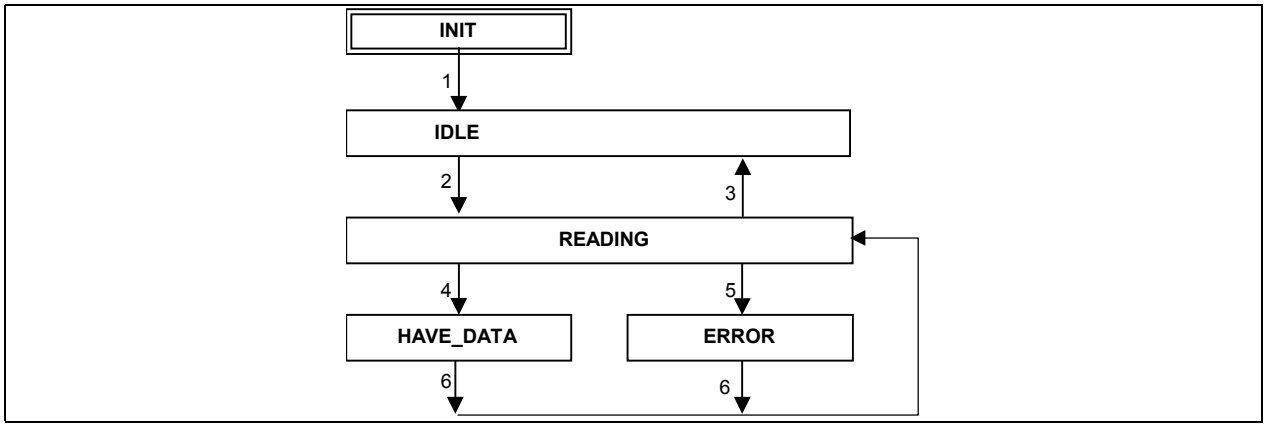


Figure 11 – State diagram of GETIO function block

The following table defines the transitions given in the state diagram above.

Table 6 - Transitions of the GETIO state diagram

Transition	Condition
1	Initialisation done
2	EN=1
3	EN=0
4	Valid I/O data
5	No valid I/O data
6	Next invocation

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters ENO and STATUS and they may have an effect on the parameter IO and LEN.

Table 7 - Action table for GETIO state diagram

		FB outputs		
State	**Actions**	**ENO**	**STATUS**	**INPUTS, LEN**
INIT[1)]	Initialise outputs	0	0	System null
IDLE	No actions	0	0	--- ???
READING	Evaluate FB input ID. Get I/O data from DP-Master with Rem Add= out of ID input Inp Data= INPUTS parameter	---	---	---
HAVE_DATA	Deposit I/O data of the slot in INPUTS parameter and set LEN output	1	0	New data
ERROR	Indicate error	0	New error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

NOTE: Reading the I/O data in READING state is performed continuously. The actual results are transferred at the next invocation in the HAVE_DATA or ERROR state.

3.2.3 Set I/O Data (SETIO)

The communication function Set I/O Data for a DP-Master (Class 1) uses the SETIO function block defined in this clause. One instance of a SETIO function block provides one instance of the PLC function Set I/O Data. The function is invoked by a 1 of the EN input.

The ID parameter identifies the slot of the DP-Slave the I/O data is set for. The IO input contains the I/O data that shall be written to the slot of the DP-Slave. The variable passed to the OUTPUTS parameter shall be of appropriate size to provide the output data. An ARRAY[1..244] OF BYTE can hold the data in all cases. The LEN input contains the length of the Output data in byte.

NOTE: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

If the Output data are stored successfully and the DP-Slave is still in Data_Exchange, the ENO output is set to 1. The output parameters of this FB are set synchronously.

If an error occurred, the ENO output is set to 0 and the STATUS output contains the error code.


```
                                SETIO
                    BOOL ---| EN                  ENO |--- BOOL
                   DWORD ---| ID                STATUS |--- DWORD
                     INT ---| LEN                      |
                     ANY ---| OUTPUTS---    ---OUTPUTS |
```

```
FUNCTION_BLOCK SETIO  (* Set Output data *)
VAR_INPUT
  EN      : BOOL;         (* Enable *)
  ID      : DWORD;        (* Identifier of a slot of a DP-Slave *)
  LEN     : INT;          (* Length of the I/O data *)
END_VAR

VAR_OUTPUT
  ENO    : BOOL;          (* Flag *)
  STATUS : DWORD;         (* Last detected status *)
END_VAR

VAR_IN_OUT
  OUTPUTS : ANY;          (* I/O data to write *)
END_VAR
```

Example: Set outputs of a slot of a DP-Slaves

```
VAR D2S3: DWORD;
    Q1  : ARRAY [1..10] OF BYTE;
    SET1: SETIO;

D2S3:= ID(1,0,2,3) (* Handle for Slave 2, Slot 3 at DP system 1 *)
Q1[1]:= 17; ...
SET1 (ID:=D2S3, OUTPUTS:=Q1);
```

Figure 12 – SETIO function block

The following state diagram describes the algorithm of the SETIO function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the SETIO function block outputs.

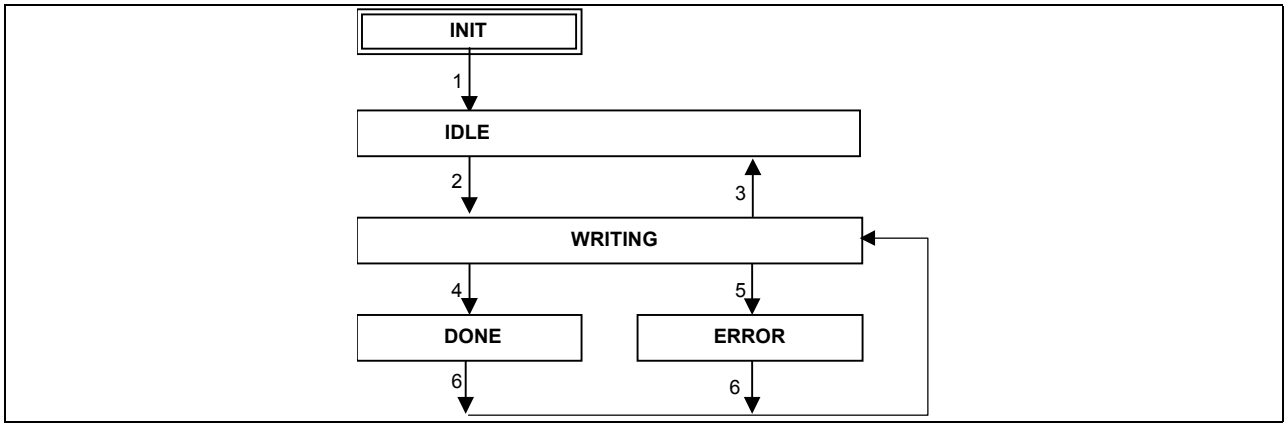


Figure 13 – State diagram of SETIO function block

The following table defines the transitions given in the state diagram above.

Table 8 - Transitions of the SETIO state diagram

Transition	Condition
1	Initialisation done
2	EN=1
3	EN=0
4	No communication problems detected
5	Communication problems detected
6	Next invocation

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters ENO and STATUS.

Table 9 - Action table for SETIO state diagram

		FB outputs	
State	**Actions**	**ENO**	**STATUS**
INIT[1)]	Initialise outputs	0	0
IDLE	No actions	0	0
WRITING	Evaluate FB input ID. Transfer I/O data to DP-Master with Rem Add= out of ID input Out Data= OUTPUTS parameter	---	---
DONE	Deposit data in parameter LEN	1	0
ERROR	Indicate error	0	New error code
--- indicates "unchanged" FB outputs. [1)] INIT is the cold start state.			

NOTE: Writing the I/O data of a DP-Slave in WRITING state is performed continuously. The actual results are transferred at the next invocation in the DONE or ERROR state.

3.3 Exchange of process data

3.3.1 General

The figure below shows the Comm FB for acyclic exchange of process data records.

The function blocks for acyclic exchange of process data records are used when a PLC is acting as a DP-Master (Class 1).

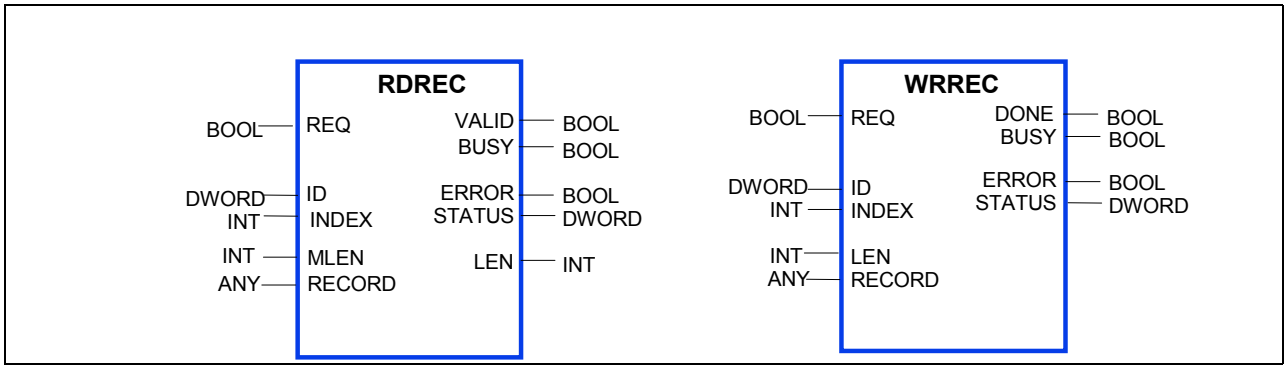


Figure 14 – Communication function blocks for acyclic exchange of data records

The function blocks for writing and reading have the similar parameter set except the different association of the send/read data as input or output parameters.

3.3.2 Read Process Data Record (RDREC)

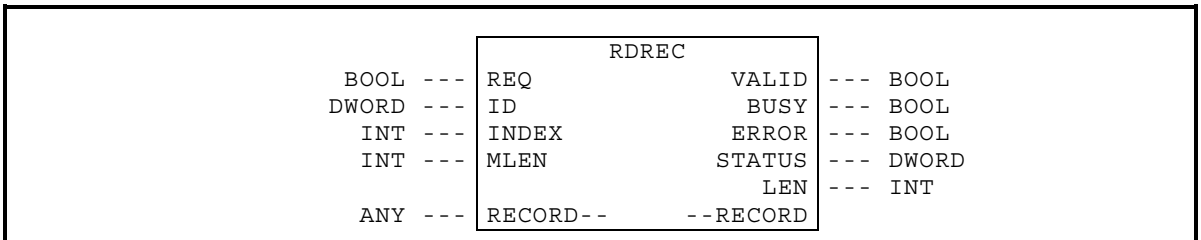
The communication function Read Process Data Record for a DP-Master (Class 1) uses the RDREC function block defined in this clause. One instance of a RDREC function block provides one instance of the PLC function Read Process Data Record. The function is invoked when REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the data record is read from. The INDEX input of the READ function block contains an integer which identifies the data record to be read.

The MLEN parameter specifies the count of bytes which shall be read as an maximum. The variable given as RECORD parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 240.

If the data record is read successfully, the VALID output indicates that the read data record is stored in the RECORD parameter. The LEN output contains the length of the data record in byte.

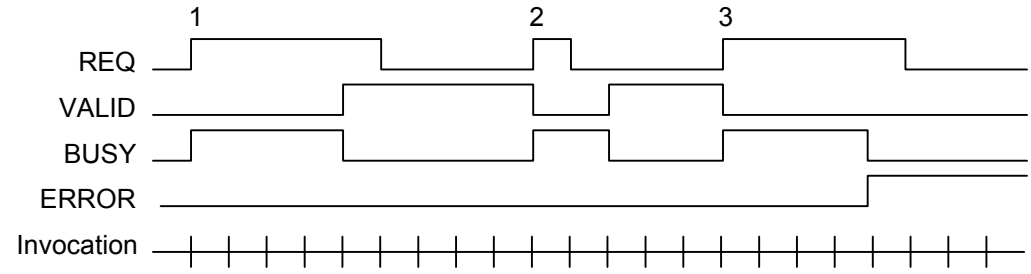
If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK RDREC  (* Read process data record *)
VAR_INPUT
  REQ     : BOOL;        (* Request *)
  ID      : DWORD;       (* Identifier of a slot of a DP-Slave *)
  INDEX   : INT;         (* Index of the data record *)
  MLEN    : INT;         (* maximum length to be read *)
END_VAR

VAR_OUTPUT
  VALID  : BOOL;         (* New data record received and is valid *)
  ERROR  : BOOL;         (* Error detected *)
  BUSY   : BOOL;         (* FB is busy *)
  STATUS : DWORD;        (* Last detected status *)
  LEN    : INT;          (* Length of the read data record *)
END_ VAR
VAR_IN_OUT
  RECORD : ANY;          (* Read data record *)
END_VAR
```

Timing Diagram:



Case 1:
The REQ input remains 1 until the function block invocation has completed, it is reset by the user when VALID gets 1.
Case 2:
The user pulses the REQ input only for one invocation. The request is not aborted.
Case 3:
Like case 1, but an error occurred.

Figure 15 – RDREC function block

The following state diagram describes the algorithm of the RDREC function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RDREC function block outputs.

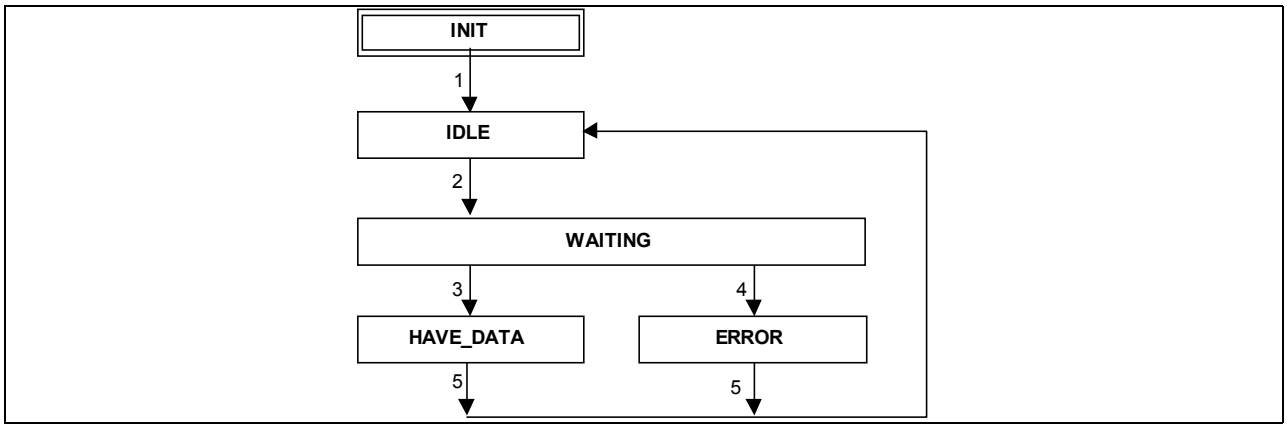


Figure 16 – State diagram of RDREC function block

The following table defines the transitions given in the state diagram above.

Table 10 - Transitions of the RDREC state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: ProcessData.Read.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: ProcessData.Read.Cnf(-) or Abort.Ind or local problems
5	Next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, STATUS and they may have an effect on the parameter RECORD, LEN.

Table 11 - Action table for RDREC state diagram

		FB outputs				
State	**Actions**	**VALID**	**BUSY**	**ERROR**	**STATUS**	**RECORD, LEN**
INIT[1)]	Initialise outputs	0	0	0	0	System null
IDLE	No actions	---	---	---	---	---
WAITING	Evaluate FB inputs ID and INDEX. Request variables from remote communication partner: ProcessData.Read.Req with AREP= slave id out of ID Slot number = slot number out of ID Index= INDEX Length= MLEN	0	1	0	-1 (is busy)	---
HAVE_DATA	Deposit data in parameter LEN and RECORD	1	0	0	0	New data
ERROR	Indicate error	0	0	1	New error code	---

--- indicates "unchanged" FB outputs.
1) INIT is the cold start state.

3.3.3 Write Data Record (WRREC)

The communication function Write Process Data Record for a DP-Master (Class 1) uses the WRREC function block defined in this clause. One instance of a WRREC function block provides one instance of the PLC function Write Process Data Record. The function is invoked when the REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the process data record is written to. The INDEX input of the WRREC function block contains an integer which identifies the data record to be written. The data record shall be stored in the variable given at the RECORD parameter. The LEN input contains the length of the data record to be written in byte. The possible value range of the LEN input is 0 .. 240. The variable given as RECORD parameter shall be at least of LEN byte.

NOTE: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

The values of the RECORD and LEN parameters shall not be changed as long as the BUSY output is true.

If the data record is written successfully, the DONE output indicates that the read data record is written to the DP-Slave.

If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.

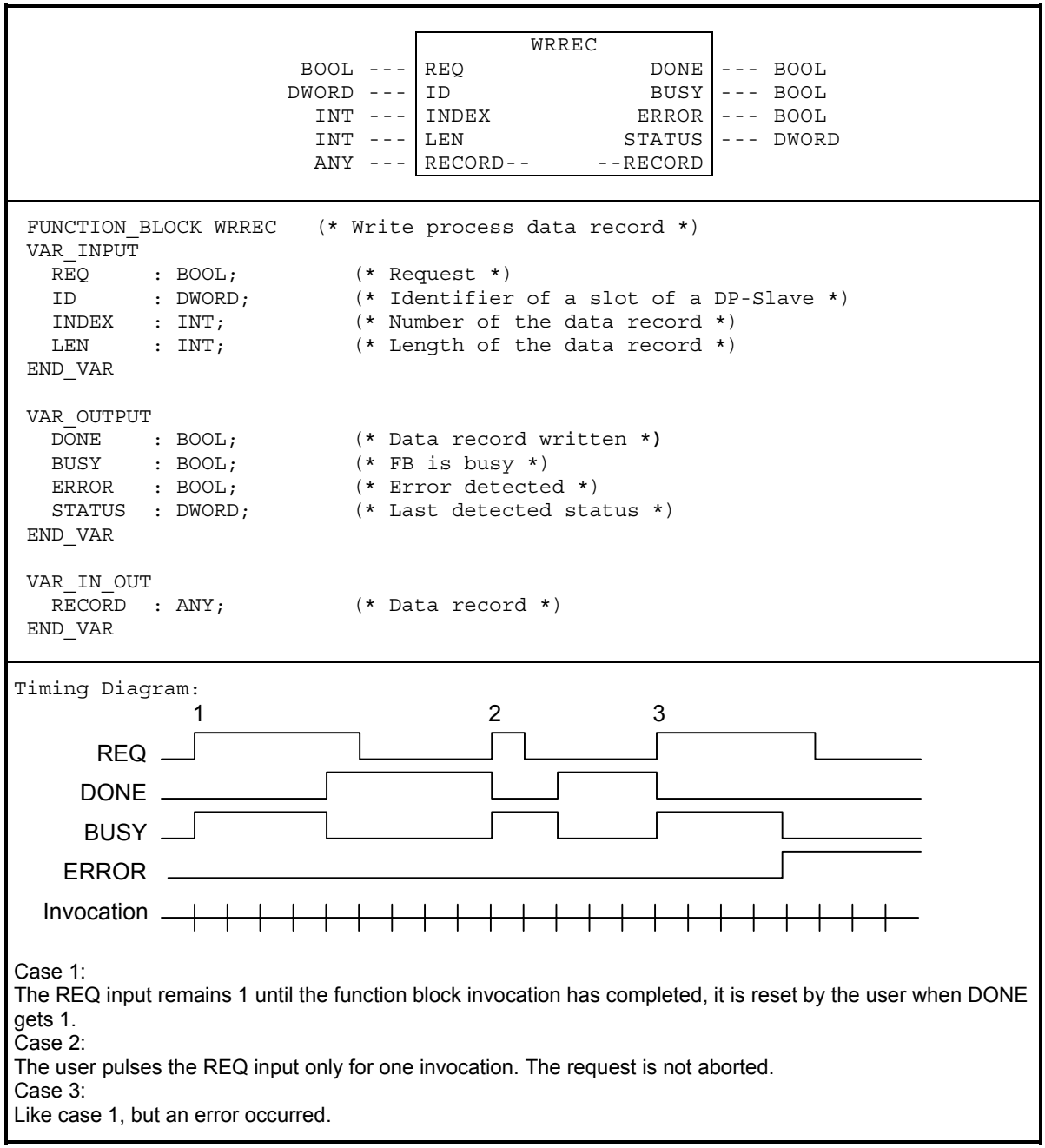


Figure 17 – WRREC function block

The following state diagram describes the algorithm of the WRREC function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the WRREC function block outputs.

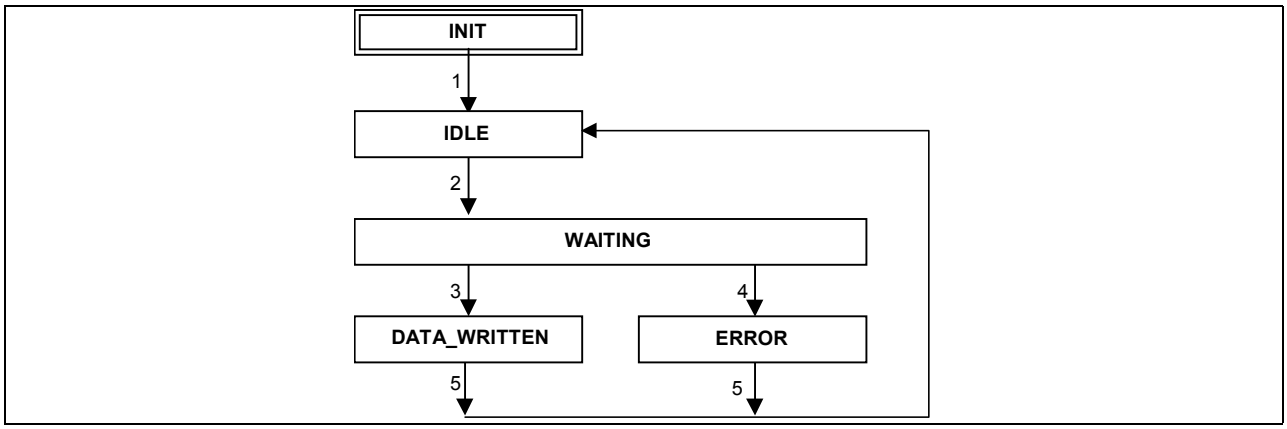


Figure 18 – State diagram of WRREC function block

The following table defines the transitions given in the state diagram above.

Table 12 - Transitions of the WRREC state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: ProcessData.Write.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: ProcessData.Write.Cnf(-) or Abort.Ind or local problems
5	After next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters DONE, BUSY, ERROR, STATUS.

Table 13 - Action table for WRREC state diagram

		FB outputs			
State	**Actions**	**DONE**	**BUSY**	**ERROR**	**STATUS**
INIT[1)]	Initialise outputs	0	0	0	0
IDLE	No actions	---	---	---	---
WAITING	Evaluate FB parameters ID, INDEX, LEN and RECORD Request variables from remote communication partner: ProcessData.Write.Req with AREP= slave id out of ID Slot number= slot number out of ID Index= INDEX Length= LEN Data= RECORD	0	1	0	-1 (is busy)
DATA_ WRITTEN	Indicate done	1	0	0	0
ERROR	Indicate error	0	0	1	New error code

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

3.4 Alarms and Diagnosis

A DP-Slave may send alarms to a DP-Master (Class 1). The PLC operating system may react on an incoming alarm by activating a task (see IEC 61131-3) of the application program. Or the application program may poll for alarm cyclically. Using the function block RALRM the application program can poll for the alarm or get more information about the alarm when activated by a task.

If the alarm activates a task the PLC operating system shall acknowledge the alarm automatically when the task is terminated. If the alarm is polled by the application program, the application program is responsible to acknowledge the alarm by using the Comm FB RALRM.

Additionally the DP system provides diagnosis status information about the DP-Slaves associated to a DP-Master (Class 1). This information can be retrieved using the function block RDIAG.

3.4.1 Receiving Alarms (RALRM)

The communication function Receive Alarm for a DP-Master (Class 1) uses the RALRM function block defined in this clause. One instance of a RALRM function block provides one instance of the PLC function Receive Alarm.

The function is invoked by EN=1. The MODE input controls the functionality of the RALRM function block.

This function blocks contains the methods to receive and acknowledge an alarm. All aspects of receiving an alarm shall use one function block instance, the different methods are distinguished using the MODE input.

MODE	Meaning
1	Receive all alarms: If the DP-Master interface has received an alarm, all function block outputs are updated. The alarm is acknowledged.
2	Receive alarms from one slot: If the DP-Master interface has received an alarm for the slot the identification of which is given in F_ID input, all function block outputs are updated. The alarm is acknowledged.

The MLEN parameter specifies the count of bytes which shall be received as a maximum of the alarm information. The byte array given as AINFO parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 4 .. 63.

NOTE: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

If an alarm is received (with MODE=1 or MODE=2), the NEW output indicates that the alarm information is stored in the outputs. The LEN output contains the length in byte of the additional alarm information stored in the AINFO parameter.

The AINFO parameter contains alarm information of the following structure:

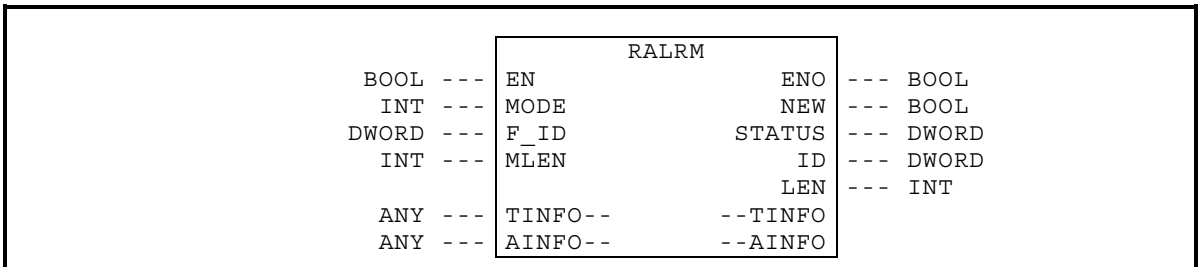
Table 14 - Structure of the variable at AINFO parameter

Component name	Data type	Meaning
D_LEN	BYTE	length of the alarm information: 4 .. 63
ATYPE	BYTE	Alarm Type: 1 Diagnosis Alarm 2 Process Alarm 3 Pull Alarm 4 Plug Alarm 5 Status Alarm 6 Update Alarm 32-126 Manufacturer Specific
SLOT	BYTE	Slot number
ASPEC	BYTE	Alarm Specifier: 0 no further differentiation 1 Alarm appears and the related module is disturbed 2 Alarm disappears and the related module has no further errors 3 Alarm disappears and the related module is still disturbed
N_ADDR	BYTE	only if D_TYPE=1: Network address
ADD_INFO	ARRAY [1..x] OF BYTE	Additional alarm information with a maximum of 59 bytes

The variable passed to the AINFO parameter shall be of appropriate size to receive the additional alarm information. An ARRAY[1..63] OF BYTE can hold the data in all cases.

NOTE: The RALRM function block may be used to receive alarm information of alarms not coming from a DP communication system. In this case the AINFO parameter may be of different length and structure. The implemetor shall specify the content of the alarm information in these cases.

If a task is started when an alarm is received the variable given at the TINFO parameter may contain additional task information. The implementor shall specify the content of the task information.




```
FUNCTION_BLOCK RALRM  (* Receive alarm *)
VAR_INPUT
  EN      : BOOL;     (* Enable *)
  MODE    : INT;      (* Function specifier *)
  F_ID    : DWORD;    (* Slot identification to filter *)
                         (* the alarms to receive *)
  MLEN    : INT;      (* Maximum length of the alarm info to receive *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;     (* Function enabled, no error *)
  NEW     : BOOL;     (* New alarm received *)
  STATUS  : DWORD;    (* DP-Master interface status *)
  ID      : DWORD;    (* Identifier of the slot the alarm is received from *)
  LEN     : INT;      (* Length of the received data record *)
END_VAR

VAR_IN_OUT
  TINFO   : ANY;      (* Additional task information *)
  AINFO   : ANY;      (* Additional alarm information *)
END_VAR
```

Example: Receive an alarm for all slots of all DP-Slaves

```
VAR A1: AINFO_TYPE;
    T1: ARRAY [1..26] OF BYTE;
    ALRM1: RALRM;

ALRM1 (EN:=1, MODE:=1, MLEN:=63, TINFO:= T1, AINFO:=A1);
IF NEW=1 THEN (* process alarm, the source identification is in
                 ALRM1.ID, the alarm type in ALRM1.A1.ATYPE, ... *)
..
```

Figure 19 – RALRM function block

The following state diagram describes the algorithm of the RALRM function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RALRM function block outputs.

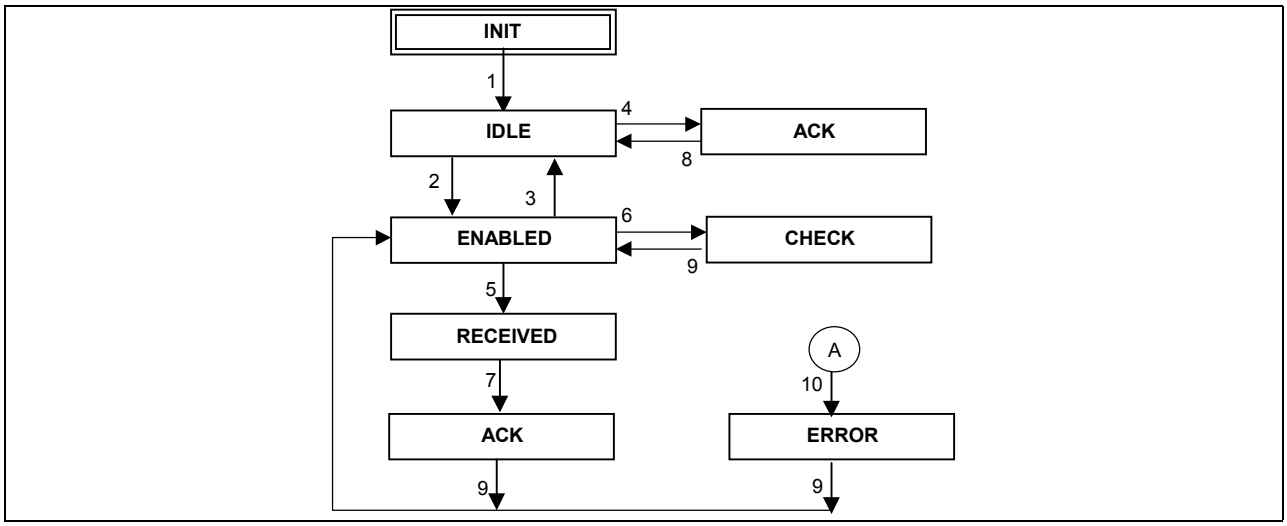


Figure 20 – State diagram of RALRM function block

The ERROR state may be entered from the states ENABLED, CHECK, RECEIVED, POS_ACK or NEG_ACK if a communication error is detected.

The following table defines the transitions given in the state diagram above.

Table 15 - Transitions of the RALRM state diagram

Transition	Condition
1	Initialisation done
2	EN = 1
3	EN = 0
4	Unacknowledged alarm from DP-Slave: Alarm.Ind
5	(MODE=1 or (MODE=2 and FSLOT=slot identification of the alarm)) and indication from DP-Slave: Alarm.Ind
6	MODE=0 and unacknowledged indication from DP-Slave
7	Immediate
8	Immediate
9	Next invocation
10	Communication error detected

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters.

Table 16 - Action table for RALRM state diagram

		FB outputs				
State	**Actions**	**ENO**	**NEW**	**ID, LEN**	**TINFO, AINFO**	**STATUS**
INIT[1)]	Initialise outputs	0	0	System null	---	0
IDLE	No actions	0	---	---	---	0
ENABLED	No actions	1	0	0	---	---
CHECK	Update outputs ID and LEN	1	1	New data	---	---
RECEIVED	Deposit data in parameter ID, LEN, TINFO, and AINFO	1	1	New data	New info	---
ACK	Positive response to DP-Master: Alarm.rsp(+) with parameters taken from the indication	1	---	---	---	---
ERROR	Update status output	0	0	---	---	New status

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

3.4.2 Read Diagnosis (RDIAG)

The communication function Read Diagnosis for a DP-Master (Class 1) uses the RDIAG function block defined in this clause. The DP system provides diagnosis status information about the DP-Slaves to a DP-Master. One instance of a RDIAG function block provides one instance of the PLC function Read Diagnosis. The function is invoked when the REQ input is equal to 1.

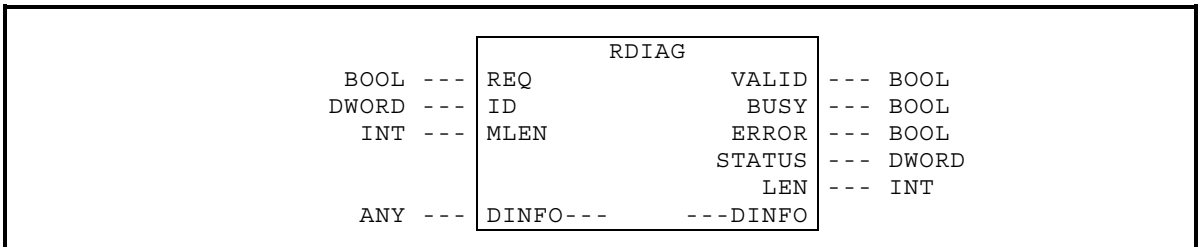
The ID parameter identifies the slot of the DP-Slave the diagnosis is read from.

The MLEN parameter specifies the count of bytes which shall be read as an maximum. The variable given at the DINFO parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 238.

If the diagnosis information is read successfully, the VALID output indicates that the data is stored in the DINFO parameter. The variable passed to the DINFO parameter shall be of appropriate size to receive the diagnosis data. An ARRAY[1..238] OF BYTE can hold the data in all cases. The LEN output contains the length of the data in byte.

NOTE: If the interface to the DP-Master can provide the diagnosis information synchronously e.g. at the time requested, the BUSY output is never seen to be 1, and the other outputs are valid.

If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK RDIAG      (* Read diagnosis *)
VAR_INPUT
  REQ     : BOOL;         (* Request *)
  ID      : DWORD;        (* Identifier of a DP-Slave *)
  MLEN    : INT;          (* maximum length to be read *)
END_VAR

VAR_OUTPUT
  VALID   : BOOL;         (* New diagnosis data received and is valid *)
  BUSY    : BOOL;         (* FB is busy *)
  ERROR   : BOOL;         (* Error detected *)
  STATUS  : DWORD;        (* Last detected status *)
  LEN     : INT;          (* Length of the diagnosis data *)
END_ VAR
VAR_IN_OUT
  DINFO   : ANY;          (* Read diagnosis data *)
END_VAR
```

Figure 21 – RDIAG function block

The following state diagram describes the algorithm of the RDIAG function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RDIAG function block outputs.

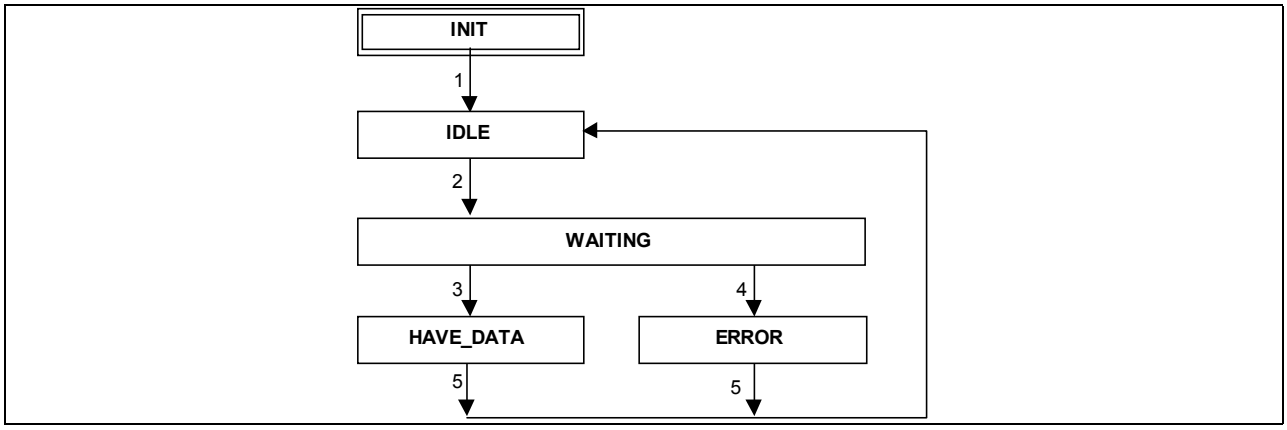


Figure 22 – State diagram of RDIAG function block

The following table defines the transitions given in the state diagram above.

Table 17 - Transitions of the RDIAG state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from DP-Master interface: GetSlaveDiag.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: GetSlaveDiag.Cnf(-) or other communication problems
5	Next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram above.

Table 18 - Action table for RDIAG state diagram

		FB outputs		
State	**Actions**	**VALID**	**LEN, DINFO**	**ERROR, STATUS**
INIT[1)]	Initialise outputs	0	System null	0
IDLE	No actions	---	---	---
WAITING	Evaluate FB inputs ID. Get diagnosis GetSlaveDiag.Req with AREP= slave id out of ID	0	---	---
HAVE_DATA	Deposit data in parameter LEN and DINFO	1	New data	0
ERROR	Indicate error	0	---	New error code

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

3.5 DP Control

The DP system can be controlled by same DP services.

A group of DP-Slaves can be directed to synchronise their inputs and outputs using the SYCFR function block.

3.5.1 Synchronise and Freeze (SYCFR)

This function block is under further consideration.

3.6 Higher Communication Functions

3.6.1 Interlocked Control (ICTRL)

For the purpose to achieve interlocked control including an order from the DP-Master to the slot of the DP-Slave accompanied by a set of data a standard function block can be used. The following timeline illustrates the sequence of this function.

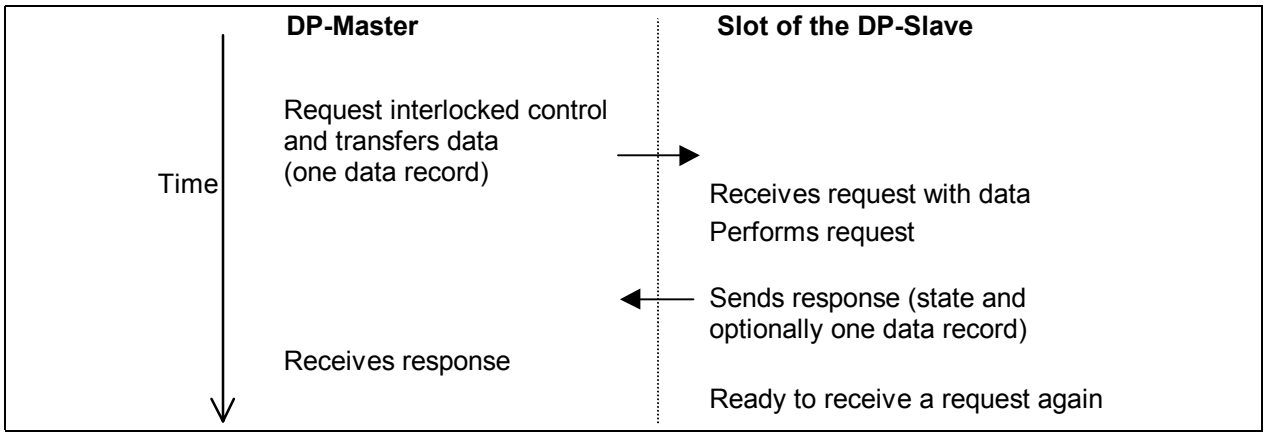


Figure 23 – Interlocked Control Timeline

The interface of this communication function "Interlocked Control" for a DP-Master (Class 1) uses the ICTRL function block defined in this clause. When interlocked control is requested the function block writes a data record to a slot of a DP-Slave. The DP-Slave performs the request and transfers the state of its execution to the function block. If the request is done and a result data record is available at the DP-Slave, the function block reads this data record.

The maximum amount of data for the request and the maximum amount of data which can be received as a result is one data record, i.e. a maximum of 240 bytes.

One instance of a ICTRL function block provides one instance of the PLC function Interlocked Control. The function is invoked when the REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the request shall be executed. The I_REQ input of the ICTRL function block contains an integer which identifies the data record of the request. The I_RES input of the ICTRL function block contains an integer which identifies the data record of the result. This feature is optional: If no result is available or no result data record shall be transferred, the value –1 shall be given. The R_STATE input of the ICTRL function block gives the remote state byte in the user data of the slot which will be used to get the state of the request execution.

The DATA_REQ input of the ICTRL function block contains the data of the request. The first byte of these data may be used as a method identifier. The first byte shall not have the value 255.

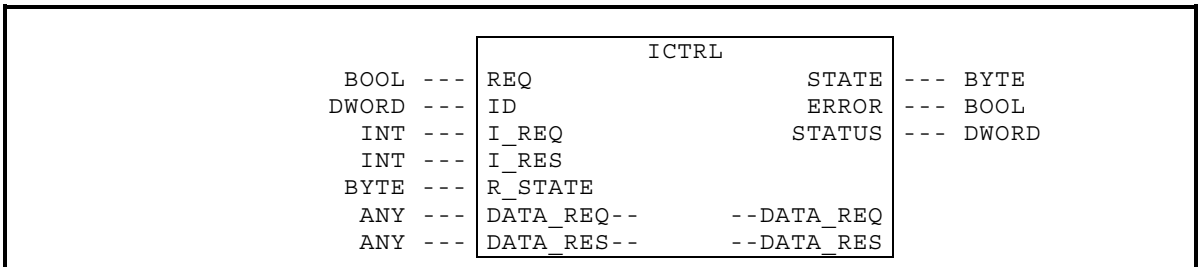
If the function block is called with REQ input = 0 a still executing request is cancelled.

The state of the request execution is given at the STATE output. It contains one of the following values:

Table 19 – States of interlocked control execution

Value of STATE output	Meaning
0	NOT_READY: Not ready to receive a new request
1	READY: Ready to receive a new request
2	REQUESTING: Request received and transferring
3	EXECUTING: Request executing
4	READY_WITHOUT_DATA: Request done at DP-Slave without result data record
5	READY_WITH_DATA: Request done at DP-Slave, result data record available
6	READING_RESULT: Reading result data record
7	READY: Function block completed
254	ABORTED: Request aborted by DP-Slave
255	CANCELLED: Request cancelled by function block

The DATA_RES output contains the result data record of the request if a result data record is available.




```
FUNCTION_BLOCK ICTRL  (* Start interlocked control *)
VAR_INPUT
  REQ     : BOOL;         (* Request *)
  ID      : DWORD;        (* Identifier of a slot of a DP-Slave*)
  I_REQ   : INT;          (* Index of the data record to request
                             interlocked control *)
  I_RES   : INT;          (* Index of the data record
                             for the result *)
  R_STATE : BYTE;         (* Remote job state byte out of the
                             user data of the slot *)
END_VAR

VAR_OUTPUT
  STATE   : BYTE;         (* State of the execution *)
  ERROR   : BOOL;         (* Error detected *)
  STATUS  : DWORD;        (* Last detected status *)
END_VAR

VAR_IN_OUT
  DATA_REQ: ANY;          (* Data record to request the job *)
  DATA_RES: ANY;          (* Data record for the result *)
END_VAR
```

Example:

```
VAR JOB1: ICRTL;
    START: BOOL;
    D3S4: DWORD;          (* Identifier of the slot *)
    REQ1: ARRAY[1..5] OF BYTE := [1, 4(0)]; (* Request buffer *)
    RSP1: ARRAY[1..3] OF BYTE               (* Response buffer *)

JOB1 (REQ:= START, ID:= D3S4,      (* Start interlocked control *)
      I_REQ:= 34, I_RES:= 34,      (* Use records #34 for data transfer *)
      R_STATE:= %IB16,             (* Use Input byte 16 for state *)
      DATA_REQ:= REQ1, DATA_RES:= RSP1);
```

Figure 24 – ICTRL function block

The following state diagram in the figure below describes the algorithm of the ICTRL function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the ICTRL function block outputs.

One byte of the user input data of the device is used for synchronisation of remote jobs as the remote job state. It is typically read cyclic by the device FB and given to the FB ICTRL via its input parameter R_STATE.

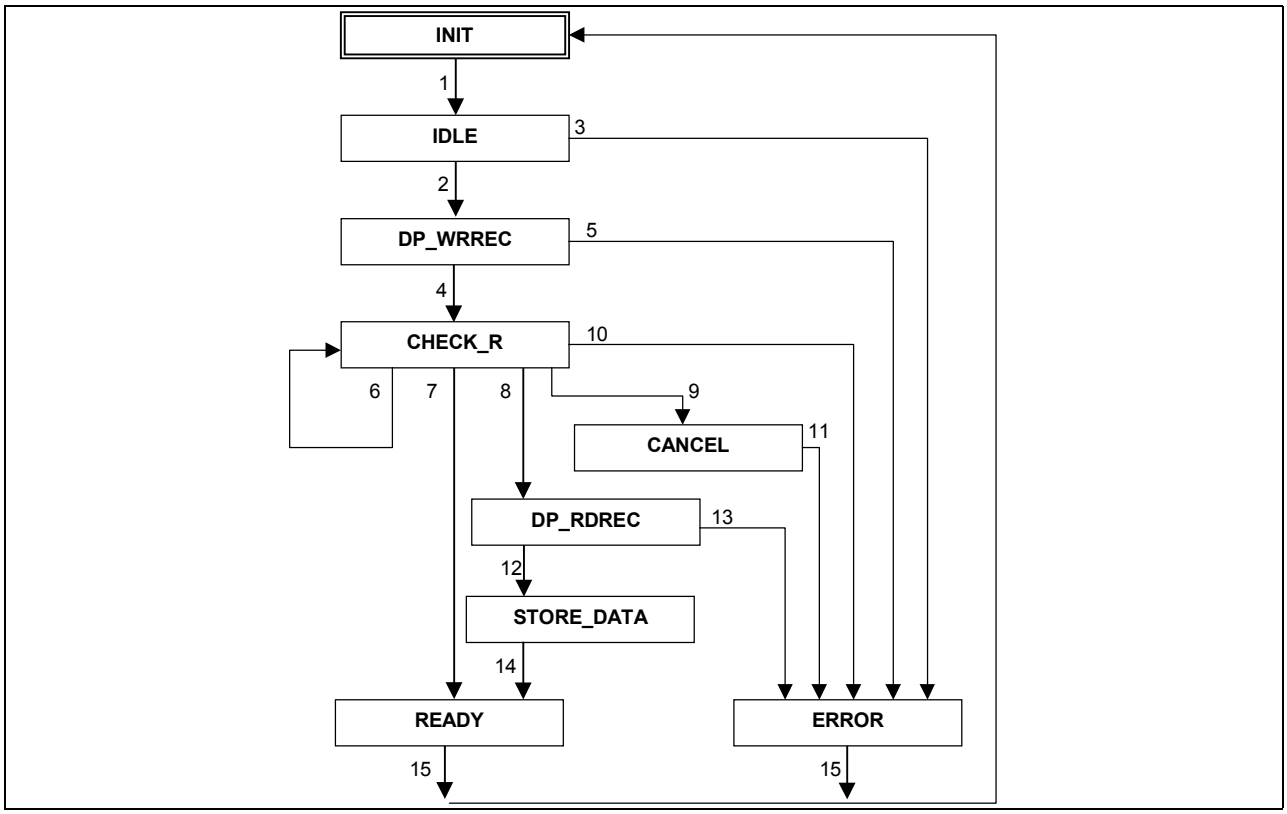


Figure 25 – State diagram of ICTRL function block

Table 20 - Transitions of the ICTRL state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1 and R_STATE = 1 (READY)
3	REQ = 1 and R_STATE <> 1 (READY)
4	Positive result of FB WRREC from remote communication partner: NDR=1; STATUS=0
5	REQ=0 or negative result of FB WRREC from remote communication partner: ERROR=1; STATUS<>0
6	Remote job state of input parameter R_STATE = 3 (EXECUTING)
7	Remote job state of input parameter R_STATE = 4 (READY_WITHOUT_DATA)
8	Remote job state of input parameter R_STATE = 5 (READY_WITH_DATA)
9	REQ=0
10	Remote job state of input parameter R_STATE <> 3, 4 or 5
11	Immediate
12	Positive result of FB RDREC from remote communication partner: NDR=1; STATUS=0 and result data are ready
13	Negative result of FB RDREC from remote communication partner: ERROR=1; STATUS<>0
14	Immediate
15	After next invocation of this instance

Table 21 - Action table for ICTRL state diagram

		FB outputs			
State	**Actions**	**STATE**	**ERROR**	**STATUS**	**DATA_RES**
INIT[1)]	Initialise outputs	0= NOT_READY	0	0	System null
IDLE	No actions	= R_STATE	0	---	---
WRREC	Request new interlocked control from remote communication partner: VAR IC1: WRREC; IC1(REQ:= REQ, ID:= ID INDEX:= I_REQ; RECORD:= DATA_REQ)	2= REQUESTING	---	---	---
CHECK_R	Check input parameter R_STATE	= R_STATE	---	---	---
CANCEL	Request new interlocked control from remote communication partner: DATA_REQ [1]:= 16#FF; IC1(REQ:= REQ, ID:= ID INDEX:= I_REQ; RECORD:= DATA_REQ)	255 = CANCELLED	---	---	---
RDREC	Read result from remote communication partner: VAR IC2: RDREC; IC2 (REQ:= REQ, ID:= ID INDEX:= I_RES);	6= READING _RESULT	---	---	System null
STORE_DATA	Deposit data	---	---	---	IC2.RECORD
READY		7= READY	0	0	---
ERROR	Indicate error	0	1	Error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

If the remote job function is implemented as an integrated functionality of a device-FB, it is possible to skip the state CHECK_R and poll the result of the remote job using RDREC after an appropriate time. In this case an application result has to be defined, that states that the result is not ready and still executing (STATE output = 3). The use of one byte of the user data of the device is not necessary, no byte of the user data has to be reserved for synchronisation of remote jobs.

4 Communication Function Blocks for DP-Master (Class 2)

4.1 General

A PLC may act as a DP-Master (Class 2).

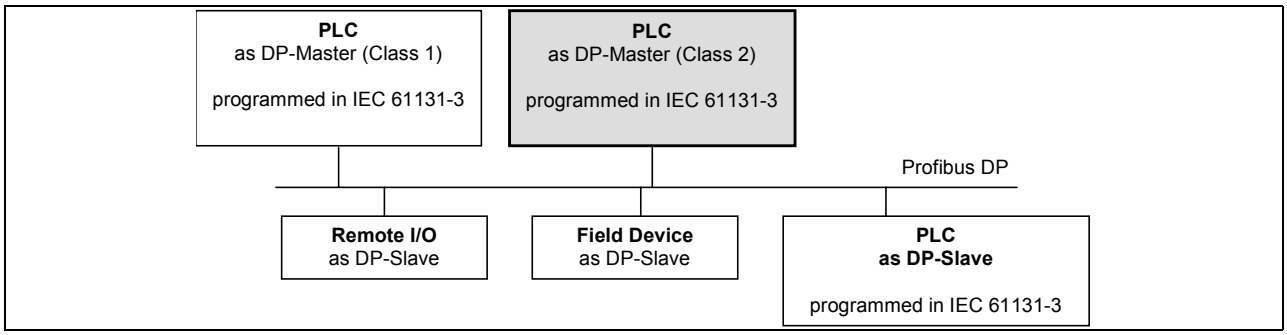


Figure 26 – Profibus system with a PLC as DP-Master (Class 2)

The following function blocks define the application program interface for a PLC acting as a DP-Master (Class 2):

- RDIN: Read input data of a DP-Slave
- RDOUT: Read output data of a DP-Slave
- RDREC: Read a process data record from a slot of a DP-Slave
- WRREC: Write a process data record to a slot of a DP-Slave
- RDIAG: Read diagnosis information from a DP-Slave
- CNCT: Manage a connection to a DP-Slave

4.2 Reading I/O data

4.2.1 Read Input Data (RDIN)

The communication function Read Input Data Record for a DP-Master (Class 2) uses the RDIN function block defined in this clause. One instance of a RDIN function block provides one instance of the PLC function Read Input Data. The function is invoked when the REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the input data is read from.

The MLEN parameter specifies the count of bytes which shall be read as a maximum. The variable given as INPUTS parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 244.

If the input data are read successfully, the VALID output indicates that the read data are stored in the IO output. The LEN output contains the length of the read Input data in byte.

If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.

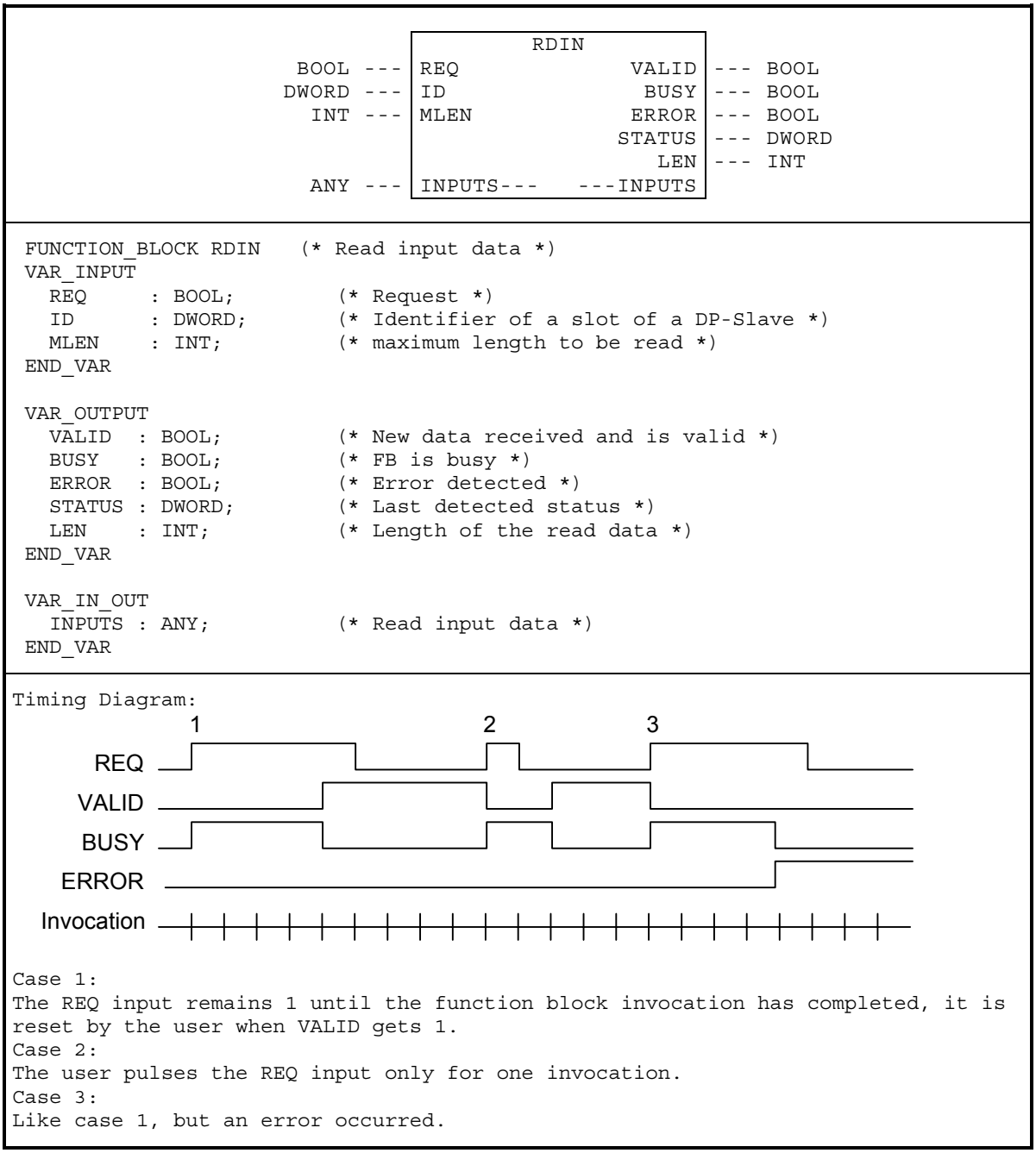


Figure 27 – RDIN function block

The following state diagram describes the algorithm of the RDIN function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RDIN function block outputs.

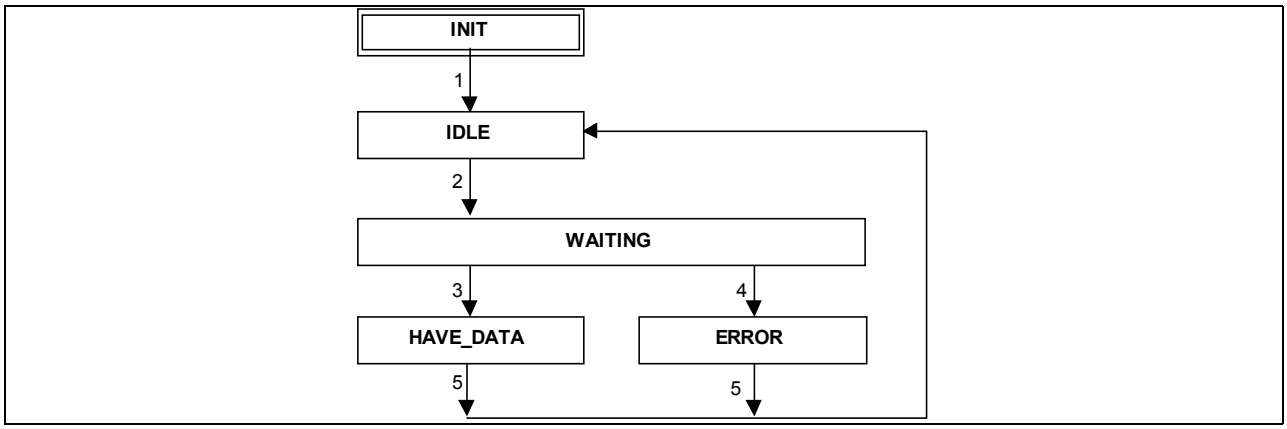


Figure 28 – State diagram of RDIN function block

The following table defines the transitions given in the state diagram above.

Table 22 - Transitions of the RDIN state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: ReadInput.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: ReadInput.Cnf(-) or Abort.Ind or local problems
5	Immediate

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, STATUS and they may have an effect on the parameter INPUTS, LEN.

Table 23 - Action table for RDIN state diagram

		FB outputs				
State	**Actions**	**VALID**	**BUSY**	**ERROR**	**STATUS**	**INPUTS, LEN**
INIT[1)]	Initialise outputs	0	0	0	0	System null
IDLE	No actions	---	---	---	---	---
WAITING	Evaluate FB input ID. Request variables from remote communication partner: ReadInput.Req with AREP= device id out of ID	0	1	0	-1	---
HAVE_DATA	Deposit data in parameter LEN and IO	1	0	0	0	New data
ERROR	Indicate error	0	0	1	New error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

4.2.2 Read Output Data (RDOUT)

The communication function Read Output Data for a DP-Master (Class 2) uses the RDOUT function block defined in this clause. One instance of a RDOUT func-

tion block provides one instance of the PLC function Read Output Data. The function is invoked when the REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the output data is read from.

The MLEN parameter specifies the count of bytes which shall be read as an maximum. The byte array given as OUTPUTS parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 244.

If the output data are read successfully, the VALID output indicates that the read data are stored in the OUTPUTS parameter. The variable passed to the OUTPUTS parameter shall be of appropriate size to receive the output data. An ARRAY[1..244] OF BYTE can hold the data in all cases. The LEN output contains the length of the read Output data in byte.

If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.

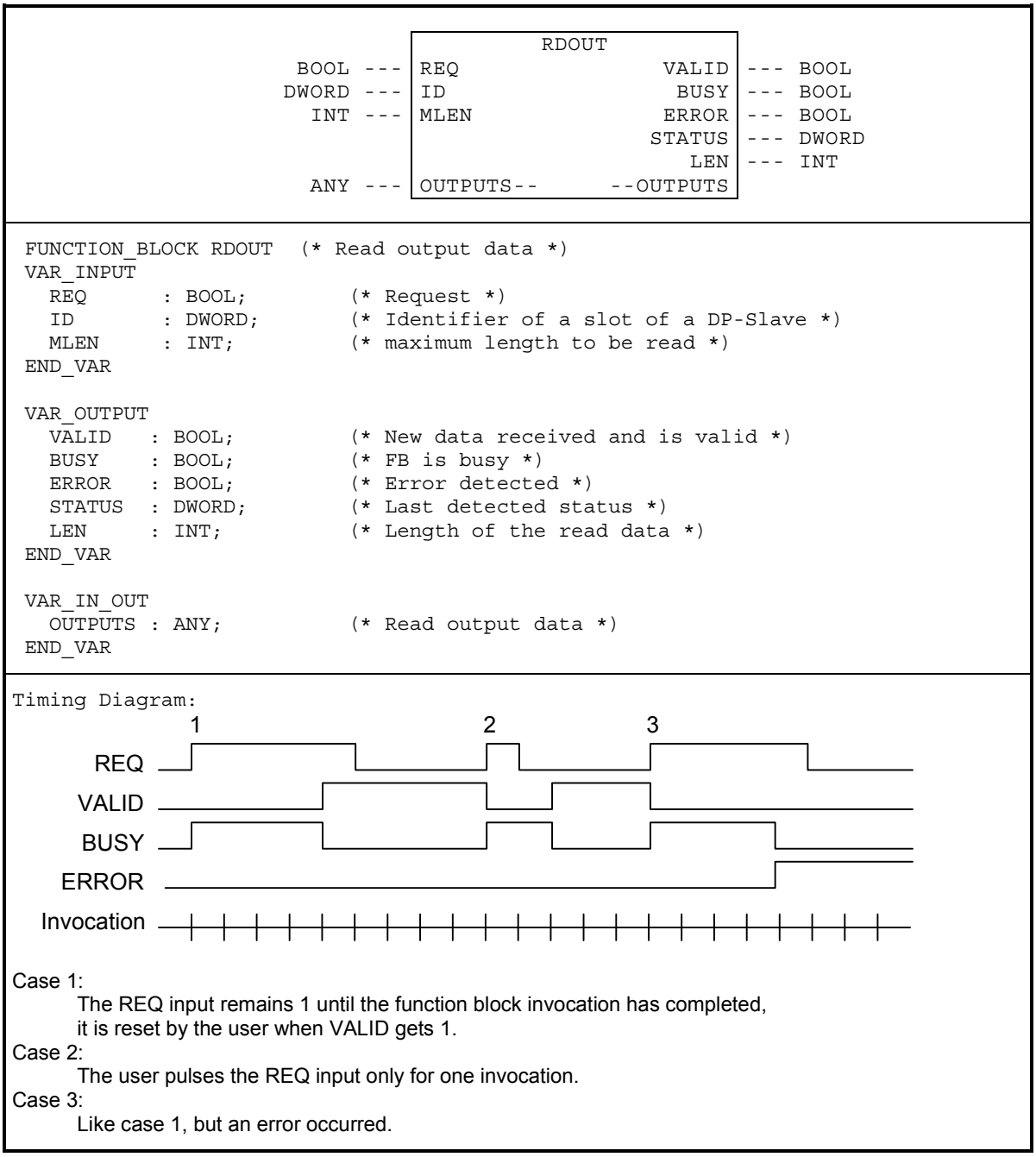


Figure 29 – RDOUT function block

The following state diagram describes the algorithm of the RDOUT function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RDOUT function block outputs.

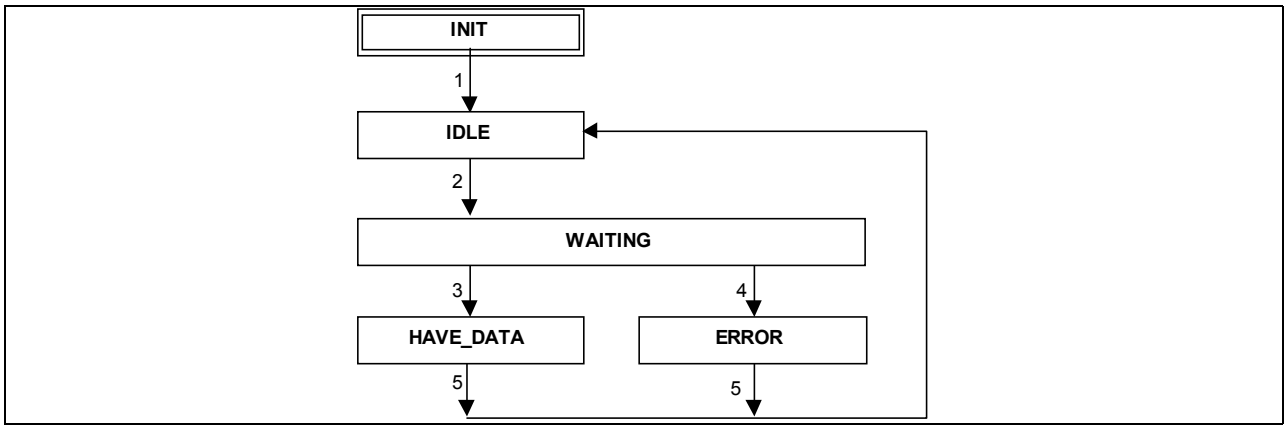


Figure 30 – State diagram of RDOUT function block

The following table defines the transitions given in the state diagram above.

Table 24 - Transitions of the RDOUT state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: ReadOutput.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: ReadOutput.Cnf(-) or Abort.Ind or local problems
5	Immediate

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, STATUS and they may have an effect on the parameter OUTPUTS, LEN.

Table 25 - Action table for RDOUT state diagram

		FB outputs				
State	**Actions**	**VALID**	**BUSY**	**ERROR**	**STATUS**	**IO, LEN**
INIT[1)]	Initialise outputs	0	0	0	0	System null
IDLE	No actions	---	---	---	---	---
WAITING	Evaluate FB input ID. Request variables from remote communication partner: ReadOutput.Req with AREP= device id out of ID	0	1	0	-1	---
HAVE_DATA	Deposit data in parameter LEN and IO	1	0	0	0	New data
ERROR	Indicate error	0	0	1	New error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

4.3 Exchange of process data records

The function blocks RDREC and WRREC defined in the previous chapters are defined for a PLC acting as a DP-Master (Class 1). These function blocks can also be used for communication as a DP-Master (Class 2).

4.4 Diagnosis

4.4.1 Read Diagnosis (RDIAG)

The communication function Read Diagnosis for a DP-Master (Class 2) uses the RDIAG function block as defined in clause 3.4.2. One instance of a RDIAG function block provides one instance of the PLC function Read Diagnosis.

The ID parameter identifies the slot of the DP-Slave the diagnosis is read from. A Connection to this DP-Slave shall be established before.

The following table defines the transitions given in the state diagram defined in clause 3.4.2, if this Comm FB is acting in the context of a DP-Master (Class 2).

Table 26 - Transitions of the RDIAG state diagram for DP-Master (Class 2)

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from DP-Master interface: ReadSlaveDiag.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: ReadSlaveDiag.Cnf(-) or other communication problems
5	Next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram defined in clause 3.4.2, if this Comm FB is acting in the context of a DP-Master (Class 2).

Table 27 - Action table for RDIAG state diagram for DP-Master (Class 2)

		FB outputs			
State	**Actions**	**VALID**	**BUSY**	**LEN, DINFO**	**ERROR, STATUS**
INIT[1)]	Initialise outputs	0	0	System null	0
IDLE	No actions	---	0	---	---
WAITING	Evaluate FB input ID. Get diagnosis ReadSlaveDiag.Req with AREP= slave id out of ID	0	1	---	---
HAVE_DATA	Deposit data in parameter LEN and DINFO	1	0	New data	0
ERROR	Indicate error	0	0	---	New error code

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

4.5 Connection Management (CNCT)

DP-Masters (Class 2) need connections to access a DP-Slave.

NOTE: A PLC system may manage connections by local means. In this case the following function blocks for the connection management have no functionality and always indicate success.

The communication function Connection Management for a DP-Master (Class 2) uses the CNCT function block defined in this clause. One instance of a CNCT function block provides one instance of the PLC function Connection Management. A connection shall be established to a DP-Slave which is connected at a DP system. The function is invoked when the REQ input is equal to 1.

The variable given D_ADDR input shall identify the DP system the destination DP-Slave is connected at. The ID parameter identifies the DP-Slave the connection shall be established to. The variable at the D_ADDR input shall be structured as defined in the following table.

Table 28 - Structure of the variable at D_ADDR input

Component name	Data type	Meaning
D_TYPE	BYTE	= 1: Components D.N_ADDR and D.MAC are not empty
D_LEN	BYTE	length of the substructure D
D	STRUCT	Destination address
API	BYTE	AP
SCL	BYTE	Access level
N_ADDR	ARRAY [1..6] OF BYTE	only if D_TYPE=1: Network address
MAC	ARRAY [1..x] OF BYTE	only if D_TYPE=1: MAC address where x = D_LEN-8

The connection is a peer-to-peer connection. One PLC can only establish one connection to the same DP-Slave.

If the connection is established successfully, the VALID output indicates that the connection can be used.

The connection remains connected as long as the function block is called with REQ=0 or an error is indicated. If a connection is established and the function block is called with REQ input = 0 the connection is disconnected.

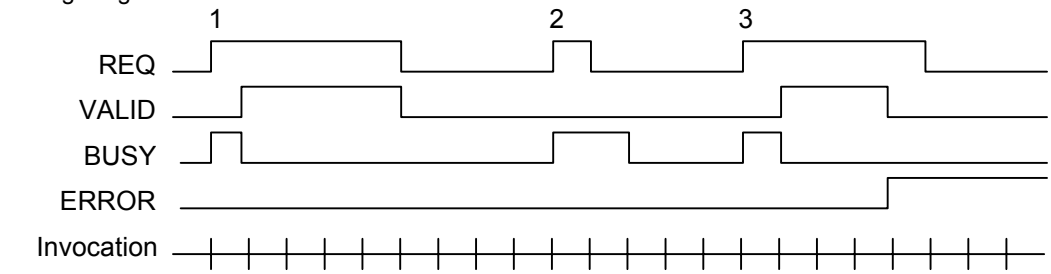
If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.

```
                          CNCT
          BOOL --- REQ          VALID --- BOOL
           ANY --- D_ADDR        BUSY --- BOOL
         DWORD --- ID           ERROR --- BOOL
                               STATUS --- DWORD
```

```
FUNCTION_BLOCK CNCT  (* Connection Management *)
VAR_INPUT
  REQ     : BOOL;       (* Request *)
  D_ADDR  : ANY;        (* Additional information for the destination address *)
  ID      : DWORD;      (* Identifier of a DP-Slave*)
END_VAR

VAR_OUTPUT
  VALID  : BOOL;        (* Connection is valid *)
  ERROR  : BOOL;        (* Error detected *)
  BUSY   : BOOL;        (* FB is busy *)
  STATUS : DWORD;       (* Last detected status *)
END_VAR
```

Timing Diagram:



Case 1:
The REQ input remains 1 until the connection is used,
it is reset by the user when the connection shall be disconnected.
Case 2:
The user aborts the establishment of a connection.
Case 3:
Like case 1, but an error occurred.

Figure 31 – CNCT function block

The following state diagram describes the algorithm of the CNCT function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the CNCT function block outputs.

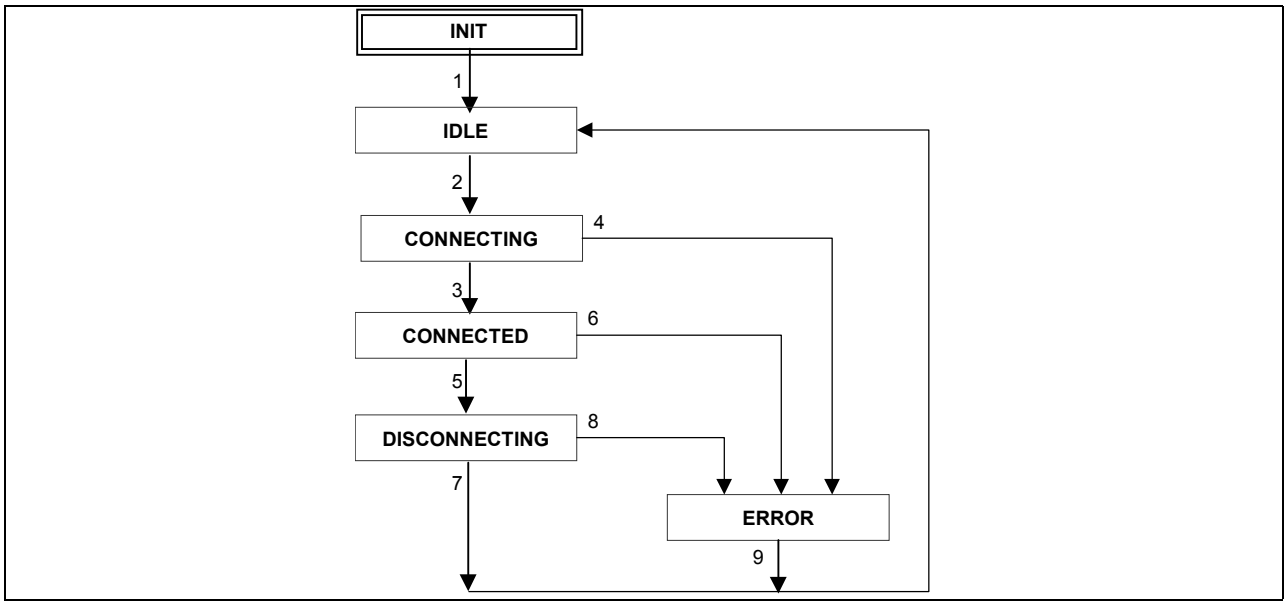


Figure 32 – State diagram of CNCT function block

The following table defines the transitions given in the state diagram above.

Table 29 - Transitions of the CNCT state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: Connect.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: Connect.Cnf(-) or Abort.Ind or local problems
5	REQ input = 0
6	Communication problems detected, connection aborted: Abort.Ind or local problems
7	Positive response from remote communication partner: Disconnect.Cnf(+)
8	Negative response from remote communication partner or other communication problems detected: Disconnect.Cnf(-) or Abort.Ind or local problems
9	Immediate

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, STATUS.

Table 30 - Action table for CNCT state diagram

		FB outputs			
State	**Actions**	**VALID**	**BUSY**	**ERROR**	**STATUS**
INIT[1)]	Initialise outputs	0	0	0	0
IDLE	No actions	---	---	---	---
CONNECTING	Evaluate FB input ID. Request to establish a point-to-point connection to the remote communication partner: Connect.Req with AREP= device id out of ID AddAddrParam.D-Len= D_LEN out of D_ADDR AddAddrParam.D-Addr.D-NetworkAddress = D	0	1	0	-1
CONNECTED	No actions	1	0	0	0
DISCONNECTING	Evaluate FB input ID. Request to close the connection to the remote communication partner: Disconnect.Req with AREP= device id out of ID	0	1	0	0
ERROR	Indicate error	0	0	1	New error code

--- indicates "unchanged" FB outputs.
1) INIT is the cold start state.

5 Communication Function Blocks for DP-Slaves

5.1 Model of a PLC as a DP-Slave

The Comm FB defined in the previous clauses are defined for PLC which is acting as a DP-Master (Class 1) or as a DP-Master (Class 2). A PLC may also be used acting as DP-Slave. The following Comm FB define the application interface to a DP system when the PLC is acting as a DP-Slave.

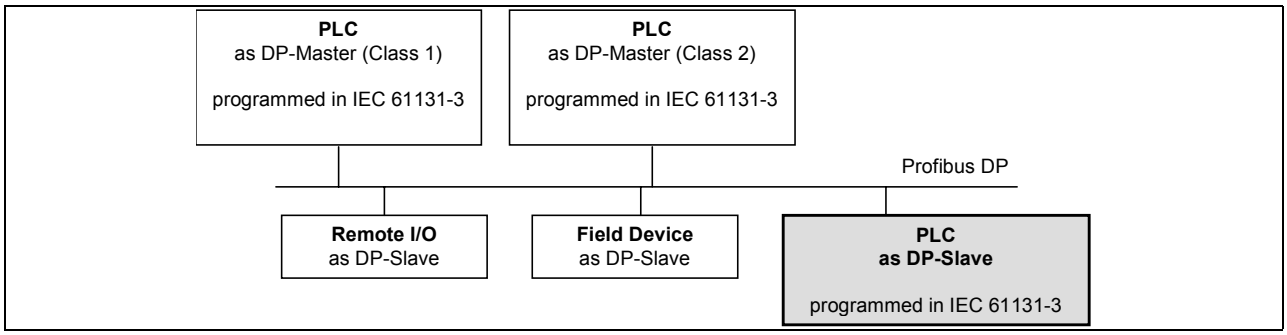


Figure 33 – Profibus system with a PLC as DP-Slave

A PLC as a DP-Slave may contain process control functions for one (or for more than one) parts of a plant or machinery. Each of these process control functions are typically implemented by function block instances or function calls using the programming languages of IEC 61131-3. In a DP-Slave one process control function shall be modelled as one (or more than one) slot.

It shall be possible, that the application program parts which implement one process control function are programmed independently from each other and from other program parts, e.g. one process control function only knows which slots it uses, and there shall be no knowledge necessary which slots are used by other process control functions. The function blocks defined as an application interface for DP-Slave in this chapter shall support this.

The application program interface to the DP system are Comm FB. The following function blocks provide this application program interface:

- RCVCO: Receives the (cyclic) output data of a DP-Master
- SBCCI: Subscribe (cyclic) input data of another DP-Slave
- PRVCI: Provides (publishes) the (cyclic) input data of the DP-Slave
- RCVREC: Receives a process data record from a DP-Master
- PRVREC: Receives a request and provides a process data record to a DP-Master
- SALRM: Request to send an alarm from the DP-Slave to the DP-Master (Class 1)
- SDIAG: Request to send diagnosis from the DP-Slave to a DP-Master

5.2 I/O Data Interface

5.2.1 General

The output data of a DP-Master (Class 1) to a DP-Slave are received by a RCVCO function block or may be mapped into the %I area of the application program of the DP-Slave, e.g. these DP-Master outputs are treated as inputs of the DP-Slave PLC.

The input data of a DP-Master (Class 1) from this DP-Slave are provided by a PRVCI function block or may be mapped into the %Q area of the application program of the DP-Slave, e.g. these DP-Master inputs are treated as outputs of the DP-Slave PLC.

Output data of another (publishing) DP-Slave to this DP-Slave may be subscribed and are received by a SBCCI function block or may be mapped into the %I area of the application program of the subscribing DP-Slave, e.g. outputs of a publishing DP-Slave are treated as inputs of the DP-Slave PLC.

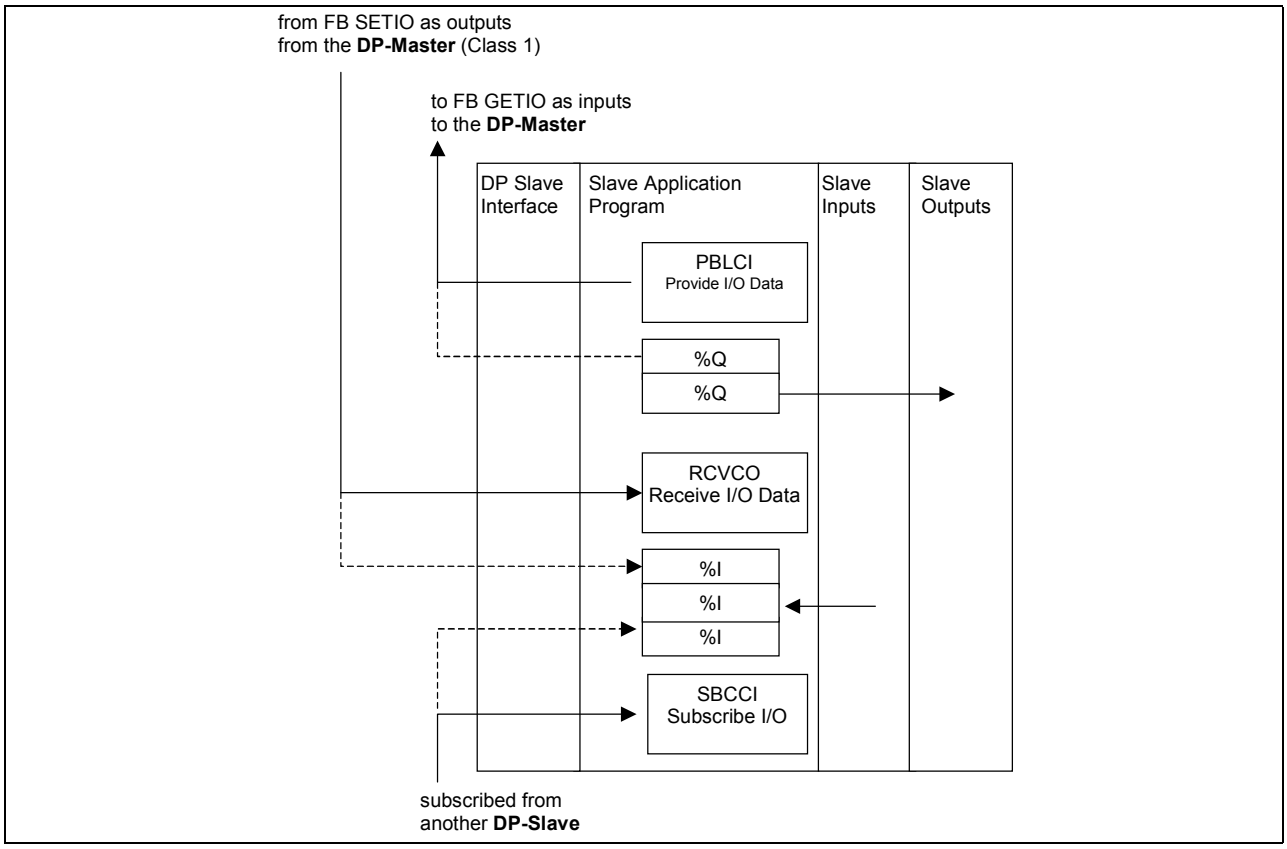


Figure 34 – PLC as a DP-Slave Using I/O Data

Enabling a Comm FB for receiving, subscribing and providing I/O data means, that the I/O data are transferred to or from the DP-Slave interface to the application program of the DP-Slave CPU. The RCVCO function block gets the output data of the addressed slot from the DP-Master out of the input data interface of the DP-Slave. The SBCCI function block subscribes and gets input data of the addressed DP-Slave from the data interface of the other DP-Slave. The PRVCI

function block provides the data of a slot to the output data interface of the DP-Slave to the DP-Master as inputs.

NOTE: The same output data of the DP-Slave CPU should not be written by different function block instances or be written via the %Q interface, because which values are transferred to the slave may be unpredictable.

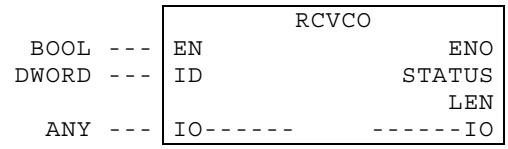
5.2.2 Receive Cyclic Output Data (RCVCO)

The communication function Receive Cyclic Output Data for a DP-Slave uses the RCVCO function block defined in this clause. One instance of a RCVCO function block provides one instance of the PLC function Receive Cyclic Output Data. The function is invoked by a 1 at the EN input.

The ID parameter identifies the slot of the DP-Slave the output data shall be received. The data are given by the DP-Master (Class 1) as its cyclic output data for this DP-Slave.

If the data are received successfully, the ENO output is set to 1 and the received I/O data are stored in the variable at the IO parameter. The variable passed to the IO parameter shall be of appropriate size to receive the diagnosis data. An ARRAY[1..244] OF BYTE can hold the data in all cases. The LEN output contains the length of the received I/O data in byte.

If an error occurred, the ENO output is set to 0 and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK RCVCO      (* Receive Cyclic Output data *)
VAR_INPUT
  EN       : BOOL;        (* Enable *)
  ID       : DWORD ;      (* Identification of a slot of the DP-Slave *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;         (* Flag *)
  STATUS  : DWORD;        (* Last detected status *)
  LEN     : INT;          (* Length of the received input data *)
END_VAR

VAR_IN_OUT
  IO      : ANY;          (* Received I/O data *)
END_VAR
```

Example:

```
VAR RECEIV1:  RCVCO;
    D3S4:     DWORD;
    BUFFER16: ARRAY [1..16] OF BYTE;

RECEIV1 (ID:=D3S4, IO:= BUFFER16);
```

The call RECEIV1 receives 16 bytes of the output data of the DP-Master(Class 1) which belongs to the slot of the DP-Slave identified by the D3S4 handle. The data are transferred to the variable BUFFER16, the count of the received bytes is in output LEN.

Figure 35 – RCVCO function block

The following state diagram describes the algorithm of the RCVCO function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RCVCO function block outputs.

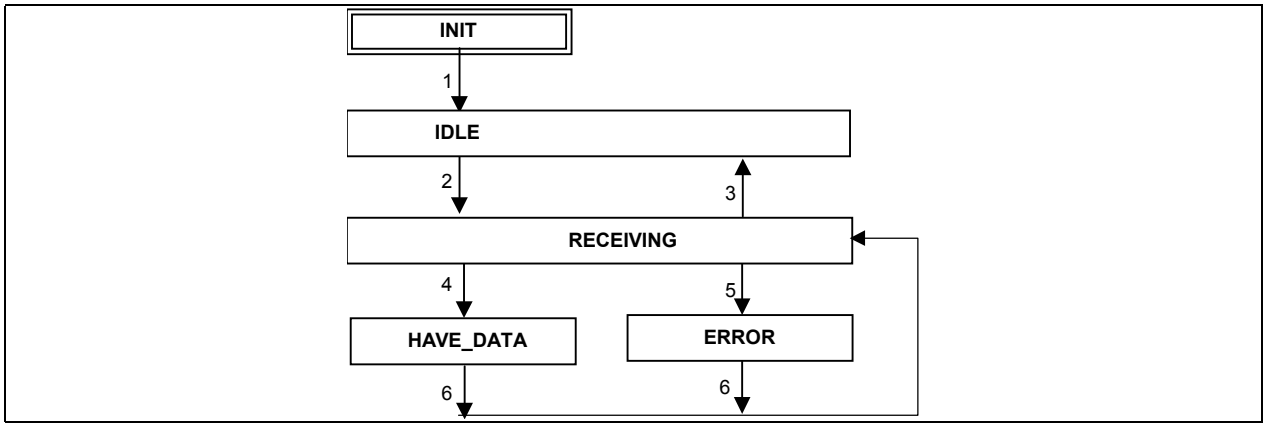


Figure 36 – State diagram of RCVCO function block

The following table defines the transitions given in the state diagram above.

Table 31 - Transitions of the RCVCO state diagram

Transition	Condition
1	Initialisation done
2	EN=1
3	EN=0
4	Next invocation of this instance and valid I/O data
5	Next invocation of this instance and no valid I/O data
6	Immediate

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters ENO and STATUS and they may have an effect on the parameter IO and LEN.

Table 32 - Action table for RCVCO state diagram

		FB outputs		
State	**Actions**	**ENO**	**STATUS**	**IO, LEN**
INIT[1)]	Initialise outputs	0	0	System null
IDLE	No actions	0	0	--- ???
RECEIVING	Evaluate FB input SLOT. Get I/O data for slot from DP-Slave interface	---	---	---
HAVE_DATA	Deposit received I/O data of the slot in IO output and set LEN output	1	0	New data
ERROR	Indicate error	0	New error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

NOTE: Receiving the I/O data in RECEIVING state is performed continuously by the DP-Slave interface. The actual results are transferred at the next invocation in the HAVE_DATA or ERROR state.

5.2.3 Subscribe Cyclic Input Data (SBCCI)

The communication function Subscribe Cyclic Input Data for a DP-Slave uses the SBCCI function block defined in this clause. One instance of a SBCCI function block provides one instance of the PLC function Subscribe Cyclic Input Data. The function is invoked by a 1 of the EN input.

This Comm FB subscribes and gets the input data of the slot of the DP-Slave identified with the ID input. The OFFSET and LEN inputs specify the data area inside the cyclic input data the other DP-Slave sends to the DP-Master and are subscribed by this DP-Slave using the SBCCI function block. The OFFSET input counts from 0.

If the data are received successfully, the ENO output is set to 1 and the subscribed I/O data are stored in the variable given at the IO parameter. The variable passed to the IO parameter shall be of appropriate size to receive the diagnosis data. An ARRAY[1..244] OF BYTE can hold the data in all cases.

If an error occurred, the ENO output is set to 0 and the STATUS output contains the error code. The STATUS values are defined in table 3.


```
                          SBCCI
          BOOL --- EN                ENO --- BOOL
         DWORD --- ID              STATUS --- DWORD
           INT --- OFFSET
           INT --- LEN
           ANY --- IO------     ------IO
```

```
FUNCTION_BLOCK SBCCI      (* Subscribe Cyclic Input data *)
VAR_INPUT
  EN      : BOOL;         (* Enable *)
  ID      : DWORD;        (* Identification of the DP-Slave *)
  OFFSET  : INT;          (* Offset of the area subscribed *)
  LEN     : INT;          (* Length of the subscribed data *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;         (* Flag *)
  STATUS  : DWORD;        (* Last detected status *)
END_VAR

VAR_IN_OUT
  IO      : ANY;          (* Subscribed I/O data *)
END_VAR
```

Example:

```
VAR SUBSCR1: SBCCI;
    D1S2   : DWORD;
    BUFFER1: ARRAY [1..6] OF BYTE;

SUBSCR1 (ID:=D1S2, IO:= BUFFER1);
```

The call SUBSCR1 subscribes the published output data of the DP-Slave identified with the D1S2 handle. The published data are transferred to the variable BUFFER1.

Figure 37 – SBCCI function block

The following state diagram describes the algorithm of the SBCCI function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the SBCCI function block outputs.

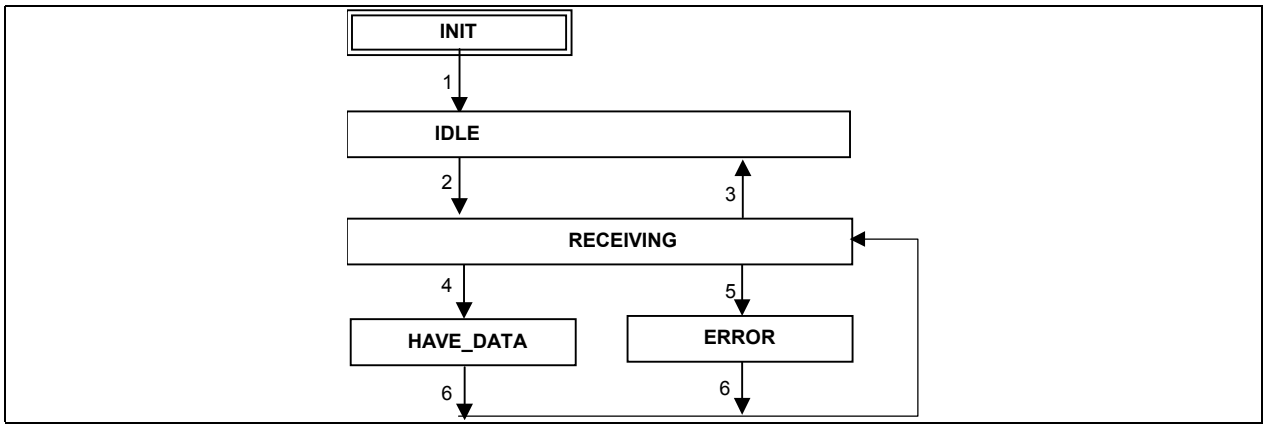


Figure 38 – State diagram of SBCCI function block

The following table defines the transitions given in the state diagram above.

Table 33 - Transitions of the SBCCI state diagram

Transition	Condition
1	Initialisation done
2	EN=1
3	EN=0
4	Valid I/O data
5	No valid I/O data
6	Next invocation

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters ENO and STATUS and they may have an effect on the parameter INPUTS.

Table 34 - Action table for SBCCI state diagram

		FB outputs		
State	**Actions**	**ENO**	**STATUS**	**IO**
INIT[1)]	Initialise outputs	0	0	System null
IDLE	No actions	0	0	---
RECEIVING	Evaluate FB input ID. Get subscribed I/O data from DP-Slave interface	---	---	---
HAVE_DATA	Deposit received I/O data of the publisher in IO parameter, set LEN output	1	0	New data
ERROR	Indicate error	0	New error code	---

--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

NOTE: Subscribing the I/O data in RECEIVING state is performed continuously by the DP-Slave interface. The actual results are transferred at the next invocation in the HAVE_DATA or ERROR state.

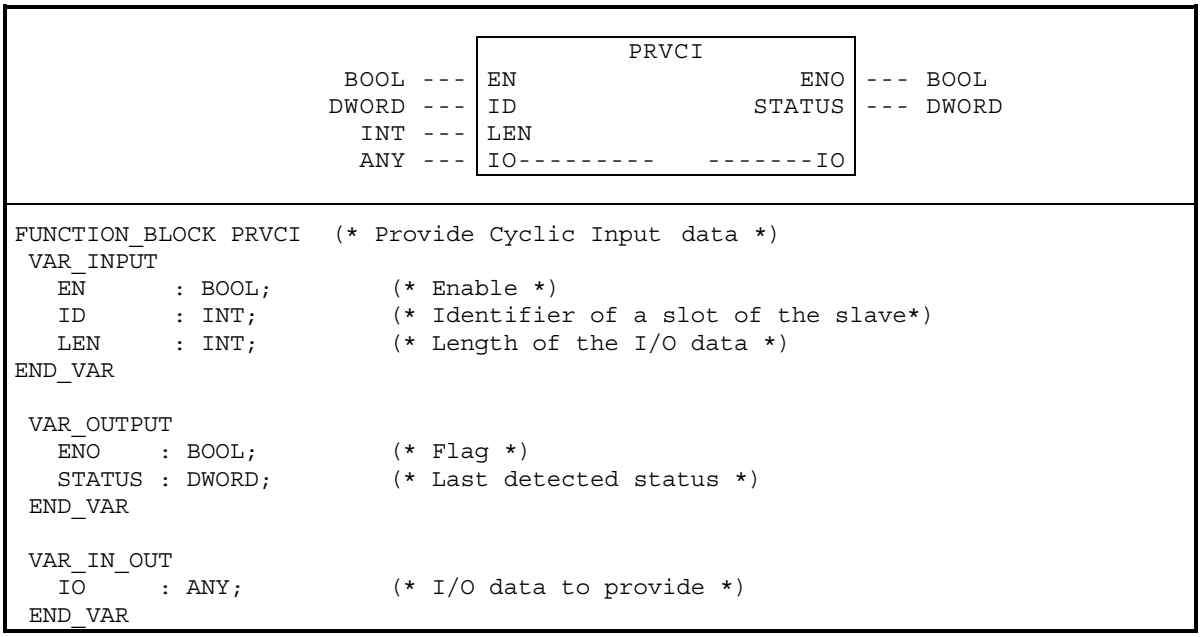
5.2.4 Provide Cyclic Input Data (PRVCI)

The communication function Provide Cyclic Input Data of a DP-Slave uses the PRVCI function block defined in this clause. One instance of a PRVCI function block provides one instance of the PLC function Provide Provide Cyclic Data. The function is invoked by a 1 of the EN input.

The ID parameter identifies the slot of the slave the I/O data is provided for. The variable given at the IO input shall contain the I/O data that shall be provided as the input data of the slot of the DP-Slave to the DP-Master. The variable passed to the IO parameter shall be of appropriate size to contain the output data. An ARRAY[1..244] OF BYTE can hold the data in all cases. The LEN input contains the length of the IO data in byte.

If the I/O data are provides successfully, the ENO output is set to 1.

If an error occurred, the ENO output is set to 0 and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK PRVCI  (* Provide Cyclic Input  data *)
 VAR_INPUT
   EN      : BOOL;         (* Enable *)
   ID      : INT;          (* Identifier of a slot of the slave*)
   LEN     : INT;          (* Length of the I/O data *)
END_VAR

 VAR_OUTPUT
   ENO    : BOOL;          (* Flag *)
   STATUS : DWORD;         (* Last detected status *)
 END_VAR

 VAR_IN_OUT
   IO     : ANY;           (* I/O data to provide *)
 END_VAR
```

Figure 39 – PRVCI function block

The following state diagram describes the algorithm of the PRVCI function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the PRVCI function block outputs.

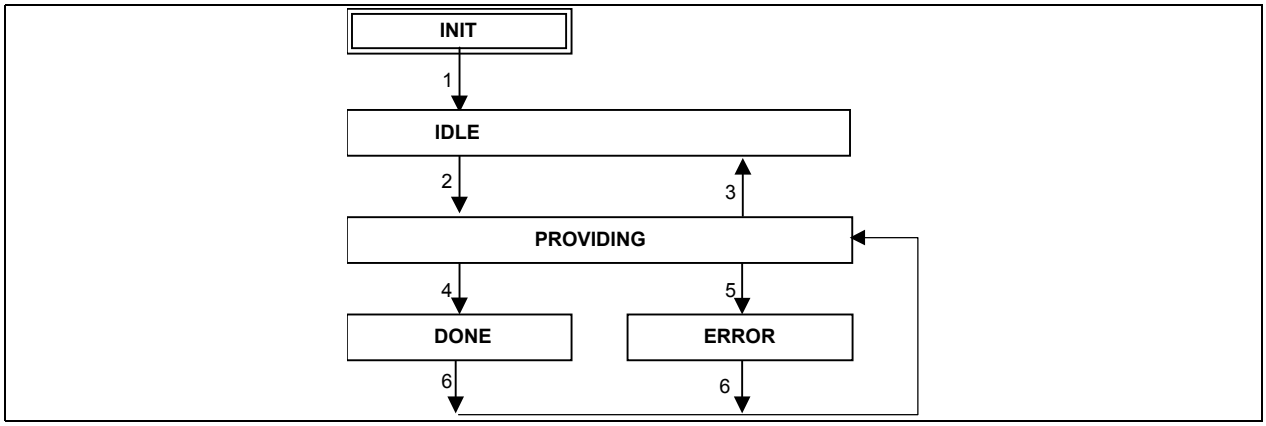


Figure 40 – State diagram of PRVCI function block

The following table defines the transitions given in the state diagram above.

Table 35 - Transitions of the PRVCI state diagram

Transition	Condition
1	Initialisation done
2	EN=1
3	EN=0
4	No communication problems detected
5	Communication problems detected
6	Next invocation

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters ENO and STATUS.

Table 36 - Action table for PRVCI state diagram

		FB outputs	
State	**Actions**	**ENO**	**STATUS**
INIT[1)]	Initialise outputs	0	0
IDLE	No actions	0	0
PROVIDING	Evaluate FB input ID. Transfer I/O data to DP-Slave interface as output data of the slot	---	---
DONE	No actions	1	0
ERROR	Indicate error	0	New error code
--- indicates "unchanged" FB outputs. 1) INIT is the cold start state.			

NOTE: Providing the I/O data of a DP-Slave to a DP-Master in PROVIDING state is performed continuously. The actual results are transferred at the next invocation in the DONE or ERROR state.

5.3 Process Data Record Interface

5.3.1 General

A DP-Slave can receive a process data record from a DP-Master (Class 1) or (Class 2). The DP-Master may use (if its a PLC) the WRREC function block. The PLC application program is informed about this using the RCVREC function block and can process the process data record.

A DP-Slave can receive a request to provide a process data record to a DP-Master (Class 1) or (Class 2). The DP-Master may use (if its a PLC) the RDREC function block. The PLC application program is informed about this request and can provide the requested process data record using the PRVREC function block.

5.3.2 Receive Process Data Record (RCVREC)

The communication function Receive Process Data Record for a DP-Slave uses the RCVREC function block defined in this clause. One instance of a RCVREC function block provides one instance of the PLC function Receive Process Data Record.

The function is invoked by EN=1. The MODE input controls the functionality of the RCVREC function block.

MODE	Meaning
0	Check for request: If the DP-Slave interface has received a process data record, only the outputs NEW, SLOT, INDEX and RLEN are set. Multiple calls of this function block with MODE=0 returns the outputs for the same request.
1	Receive all process data records: If the DP-Slave interface has received a process data record, the function block outputs are updated and the data record is transferred to the RECORD parameter. The service is responded positively.
2	Receive process data records for one slot: If the DP-Slave interface has received a process data record for the slot the number of which is given in input FSLOT, the function block outputs are updated and the data record is transferred to the RECORD parameter. The service is responded positively.
4	Negative response: After checking the request to receive a process data record, this function block refuses to accept this record and sends a negative response to the DP-Master. The error reason is given with the inputs CODE1 and CODE2.

NOTE 1: This function blocks contains the methods to check, receive and acknowledge a process data record. All aspects of receiving a process data record may use one function block instance, the different methods are distinguished using the MODE input.

The MLEN parameter specifies the count of bytes which shall be received as a maximum. The byte array given as RECORD parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 240.

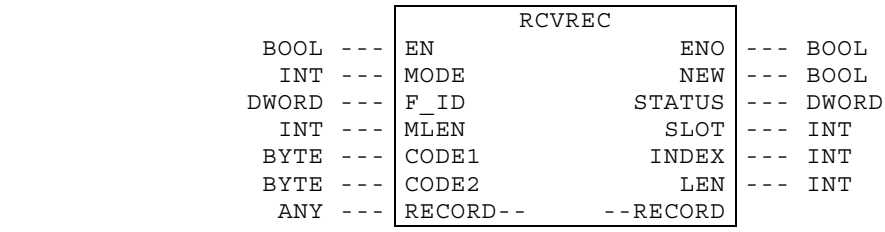
NOTE 2: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

If a data record is received (with MODE=1 or MODE=2), the NEW output indicates that the data record is stored in the variable given at the RECORD parameter. The variable passed to the RECORD parameter shall be of appropriate size to receive the process data record. An ARRAY[1..240] OF BYTE can hold the data in all cases. The LEN output contains the length of the data record in byte.

If the function block refuses to accept the data record, the CODE1 input sets the Error Code 1, and the CODE2 input sets the Error Code 2 of the negative response.

NOTE 3: The application program of the DP-Slave shall acknowledge the received request, otherwise the DP-Master will get a timeout error and will deactivate the DP interface of the DP-Slave.

If an error occurred, the ENO=0 indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK RCVREC  (* Receive process data record *)
VAR_INPUT
  EN      : BOOL;     (* Enable *)
  MODE    : INT;      (* Function specifier *)
  F_ID    : DWORD;    (* Slot identification to filter *)
                         (* the process data records to receive *)
  MLEN    : INT;      (* Maximum length of a data record to receive *)
  CODE1   : BYTE;     (* Reason for negative response *)
  CODE2   : BYTE;     (* Reason for negative response *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;     (* Function enabled *)
  NEW     : BOOL;     (* New data record received *)
  STATUS  : DWORD;    (* DP-Slave interface status *)
  SLOT    : INT;      (* Slot number the record is received for *)
  INDEX   : BOOL;     (* Index of the received process data record *)
  LEN     : INT;      (* Length of the received data record *)
END_VAR

VAR_IN_OUT
  RECORD : ANY;       (* Received data record *)
END_VAR
```

Example 1: Receive a process data record for all slots of the DP-Slave

```
VAR R1: ARRAY [1..240] OF BYTE;
    RREC1: RCVREC;

RREC1 (MODE:=1, MLEN:=240, RECORD:=R1);
IF NEW=1 THEN (* process record, the index is in RREC1.INDEX *)
..
```

Example 2: Check the received process data records, process and acknowledge conditionally

```
VAR R2: ARRAY [1..240] OF BYTE;
    RREC2: RCVREC;

RREC2 (MODE:=0); (* Check for a new record *)
IF NEW=1 THEN (* new process data record available *)
   IF RREC2.SLOT=12 and RREC2.INDEX=1 THEN
      RREC2 (MODE:=1, RECORD:=R2); (* get this record *)
         (* process data record for slot 12 and index 1 *)
   ELSEIF RREC2.SLOT=13 and RREC2.INDEX=2 THEN
      RREC2 (MODE:=1, RECORD:=R2); (* get this record *)
         (* process data record for slot 13 and index 2 *)
   :
   ELSE RREC2 (MODE:=4); (* give negative response *)
   END_IF;
```

Figure 41 – RCVREC function block

The following state diagram describes the algorithm of the RCVREC function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the RCVREC function block outputs.

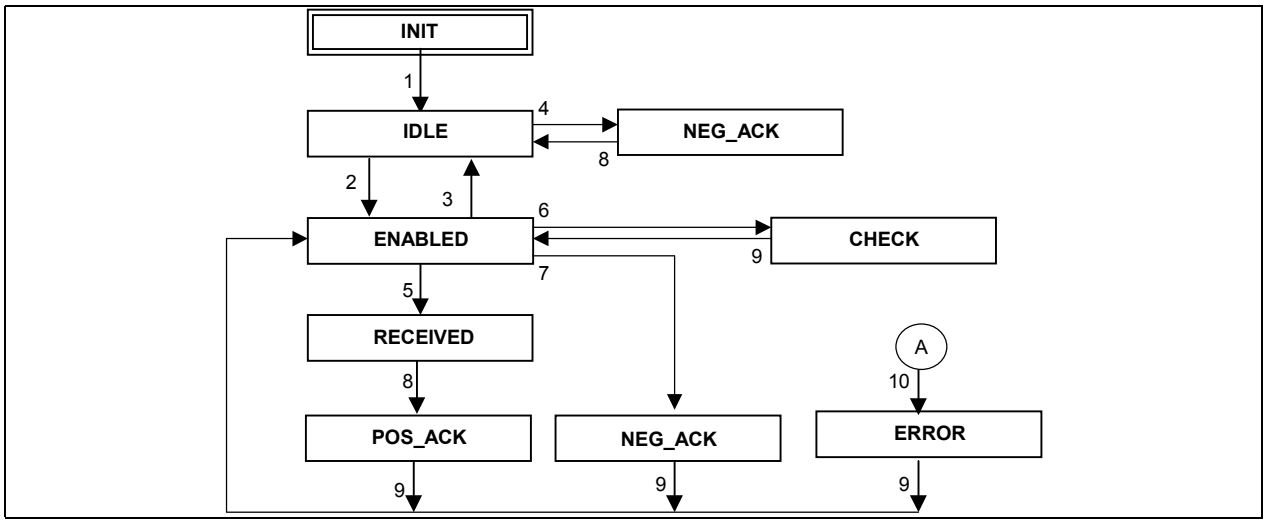


Figure 42 – State diagram of RCVREC function block

The ERROR state may be entered by the states ENABLED, CHECK, RECEIVED, POS_ACK or NEG_ACK if a communication error is detected.

The following table defines the transitions given in the state diagram above.

Table 37 - Transitions of the RCVREC state diagram

Transition	Condition
1	Initialisation done
2	EN = 1
3	EN = 0
4	Unacknowledged indication from DP-Master (Class 1) or (Class 2): ProcessData.Write.Ind
5	(MODE=1 or (MODE=2 and FSLOT=ProcessData.Write.Ind.SlotNumber)) and indication from DP-Master (Class 1) or (Class 2): ProcessData.Write.Ind
6	MODE=0 and unacknowledged indication from DP-Master (Class 1) or (Class 2)
7	MODE=4 and unacknowledged indication from DP-Master (Class 1) or (Class 2)
8	Immediate
9	Next invocation
10	Communication error detected

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters.

Table 38 - Action table for RCVREC state diagram

		FB outputs				
State	**Actions**	**ENO**	**NEW**	**SLOT, INDEX, LEN**	**RECORD**	**STATUS**
INIT[1)]	Initialise outputs	0	0	System null	---	0
IDLE	No actions	0	---	---	---	0
ENABLED	No actions	1	0	0	---	---
CHECK	Update outputs			New data	---	---
RECEIVED	Deposit data in parameter SLOT, INDEX, LEN and RECORD	1	1	New data	New record	---
POS_ACK	Positive response to DP-Master: ProcessData.Write.rsp(+) with Length= MIN(MLEN,LEN)	1	---	---	---	---
NEG_ACK	Negative response to DP-Master: ProcessData.Write.rsp(-)with Error Decode= 16#80 Error Code 1= CODE1 Error Code 2= CODE2	1	0	---	---	---
ERROR	Update status output	0	0	---	---	New status

--- indicates "unchanged" FB outputs.
1) INIT is the cold start state.

5.3.3 Provide Process Data Record (PRVREC)

The communication function Provide Process Data Record for a DP-Slave uses the PRVREC function block shown in figure below.

One instance of a PRVREC function block provides one instance of the PLC function Provide Process Data Record.

The function is invoked by EN=1. The MODE input controls the functionality of the PRVREC function block.

MODE	Meaning
0	Check for request: If the DP-Slave interface has received a request to provide a process data record, only the outputs NEW, SLOT, INDEX and RLEN are set. Multiple calls of this function block with MODE=0 returns the outputs for the same request.
1	Receive all requests: If the DP-Slave interface has received a request, the function block outputs are updated.
2	Receive requests for one slot: If the DP-Slave interface has received a request for the slot the number of which is given in FSLOT input, the function block outputs are updated.
3	Positive response: After checking or receiving the request to provide a process data record, this function block provides the requested process data record with its RECORD parameter and sends a positive response to the DP-Master.
4	Negative response: After checking or receiving the request to provide a process data record, this function block refuses to provide this record and sends a negative response to the DP-Master. The error reason is given with the inputs CODE1 and CODE2.

NOTE 1: This function blocks contains the methods to check, receive and respond a request for a process data record. All aspects of providing a process data record may use one function block instance, the different methods are distinguished using the MODE input.

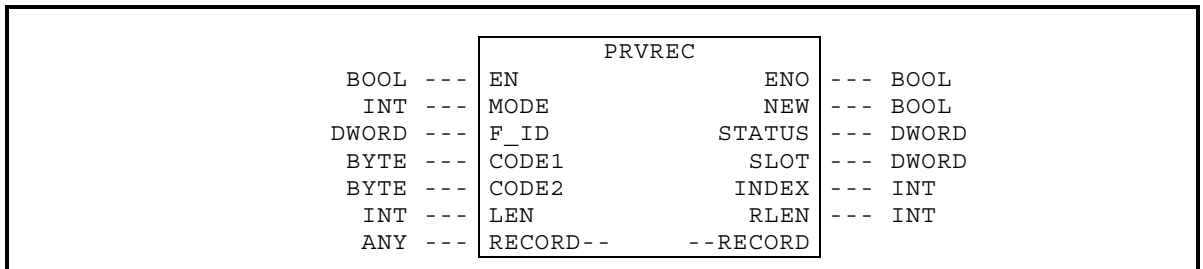
The MLEN parameter specifies the count of bytes which shall be provided as a maximum. The byte array given as RECORD parameter shall be at least of MLEN byte. Possible value range of the MLEN input is 0 .. 240.

NOTE 2: An array declaration with zero elements is not supported in IEC 61131-3, therefor the minimum length shall be 1 byte even if the record length is zero. The actual length is given with the LEN parameter.

If a data record is received (with MODE=1 or MODE=2), the NEW output indicates that the data record is stored in the variable given at the RECORD parameter. The variable passed to the RECORD parameter shall be of appropriate size to contain the process data record. An ARRAY[1..240] OF BYTE can hold the data in all cases. The LEN output contains the length of the data record in byte.

If the function block refuses to accept the data record, the CODE1 input sets the Error Code1, and the CODE2 input sets the Error Code 2 of the negative response.

If an error occurred, the ENO=0 indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK PRVREC (* Provide process data record *)
VAR_INPUT
  EN      : BOOL;     (* Enable *)
  MODE    : INT;      (* Function specifier *)
  F_ID    : DWORD;    (* Slot identification to filter the requests *)
                         (* to provide process data records *)
  CODE1   : BYTE;     (* Reason for negative response *)
  CODE2   : BYTE;     (* Reason for negative response *)
  LEN     : INT;      (* Length of a data record to provide *)
END_VAR

VAR_OUTPUT
  ENO     : BOOL;     (* Function enabled *)
  NEW     : BOOL;     (* New data record requested *)
  STATUS  : DWORD;    (* DP-Slave interface status *)
  SLOT    : INT;      (* Slot number the record is requested for *)
  INDEX   : INT;      (* Index of the requested process data record *)
  RLEN    : INT;      (* Length of the requested data record *)
END_VAR

VAR_IN_OUT
  RECORD : ANY;       (* Provided data record *)
END_VAR
```

Example 1: Provide a process data record

```
VAR P1: ARRAY [1..240] OF BYTE;
    L1: INT;
    PREC1: PRVREC;

PREC1 (EN:=1, MODE:=1);
IF NEW=1 THEN
  (* provide record, the slot number is in PREC1.SLOT,
     the index is in PREC1.INDEX, store the record in P1
     and its length in L1 *)
  PREC1 (EN:=1, MODE:=3, LEN:=L1; RECORD:=P1);
..
```

Example 2: Check the received requests, process and acknowledge conditionally

```
VAR P2: ARRAY [1..240] OF BYTE;
    L2: INT;
    PREC2: PRVREC;

PREC2 (EN:=1, MODE:=0);                (* check for a new request *)
IF NEW=1 THEN (* new request for a process data record available *)
   IF PREC2.SLOT=12 and PREC2.INDEX=1 THEN
     (* build record for slot 12 and index 1 and store in P2 *)
      PREC2 (MODE:=3, RECORD:=P2);     (* provide this record *)

   ELSEIF PREC2.SLOT=13 and PREC2.INDEX=2 THEN
     (* build record for slot 13 and index 2 and store in P2 *)
      PREC2 (MODE:=3, RECORD:=R2);     (* get this record *)
   :
   ELSE PREC2 (MODE:=4);               (* give negative response *)
   END_IF;
```

Figure 43 – PRVREC function block

The following state diagram describes the algorithm of the PRVREC function blocks. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the PRVREC function block outputs.

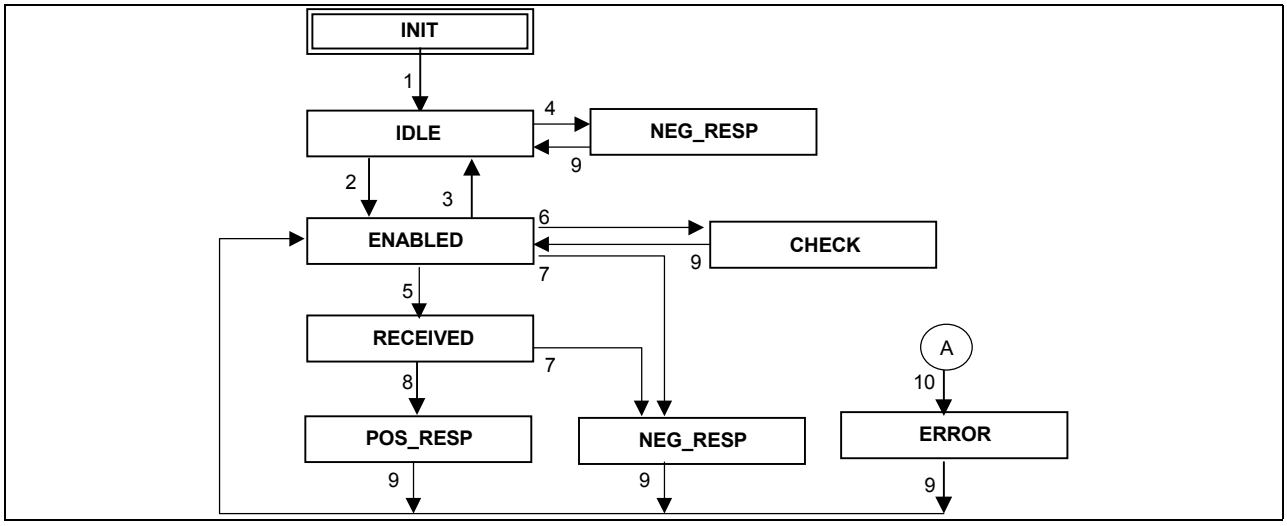


Figure 44 – State diagram of PRVREC function block

The ERROR state may be entered by the states ENABLED, CHECK, RECEIVED, POS_RESP or NEG_RESP if a communication error is detected.

The following table defines the transitions given in the state diagram above.

Table 39 - Transitions of the PRVREC state diagram

Transition	Condition
1	Initialisation done
2	EN = 1
3	EN = 0
4	Indication from DP-Master (Class 1) or (Class 2): ProcessData.Read.Ind
5	(MODE=1 or (MODE=2 and FSLOT=ProcessData.Read.Ind.SlotNumber)) and indication from DP-Master (Class 1) or (Class 2): ProcessData.Read.Ind
6	MODE=0 and unresponded indication from DP-Master (Class 1) or (Class 2): ProcessData.Read.Ind
7	MODE=4 and unresponded indication from DP-Master (Class 1) or (Class 2): ProcessData.Read.Ind
8	MODE=3 and unresponded indication from DP-Master (Class 1) or (Class 2): ProcessData.Read.Ind
9	Next invocation

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters.

Table 40 - Action table for PRVREC state diagram

		FB outputs			
State	**Actions**	**ENO**	**NEW**	**SLOT, INDEX, RLEN**	**STATUS**
INIT[1)]	Initialise outputs	0	0	System null	
IDLE	No actions	0	---	---	
ENABLED	No actions	1	0	0	
CHECK	No actions	1	1	New data	
RECEIVED	Deposit data in parameter SLOT, INDEX, and RLEN	1	1	New data	
POS_RESP	Positive response to DP-Master: ProcessData.Read.rsp(+) with Length= LEN Data= RECORD	1	---	---	
NEG_RESP	Negative response to DP-Master: ProcessData.Read.rsp(-) with Error Decode= 16#80 Error Code 1=CODE1 Error Code 2= CODE2	1	0	---	
ERROR	Update status	0	0	---	New Status

--- indicates "unchanged" FB outputs.
1) INIT is the cold start state.

5.4 Alarm Handling and Diagnosis

A DP-Slave can generate alarms to its associated DP-Master (Class 1) to inform it e.g. about certain process events or other events to state the some limitations of the capabilities of the DP-Slave for diagnostic reasons.

The DP-Slave shall inform its DP-Master (Class 1) when it is in a state which prevents it to perform the intended process control function. A DP-Master (Class 2) shall be able to read this information. The PLC application program of the DP-Slave shall provide adequate diagnosis information when it recognises such a state.

5.4.1 Send Alarm (SALRM)

The communication function Send Alarm for a DP-Slave uses the SALRM function block defined in this clause. One instance of a SALRM function block provides one instance of the PLC function Send Alarm. The function is invoked when the REQ input is equal to 1.

The ID parameter identifies the slot of the DP-Slave the alarm is generated for. The ATYPE input shall contain the alarm type as specified in IEC 61158-6. The ASPEC input shall contain the alarm specifier as defined in IEC 61158-6. The LEN input contains the length in byte of the additional alarm information stored in the AINFO parameter.

The Variable given as AINFO parameter shall be at least of LEN byte. Possible value range of the LEN input is 0 .. 59.

If the alarm is transmitted successfully, the VALID output indicates that the alarm was received by the DP-Master (Class 1).

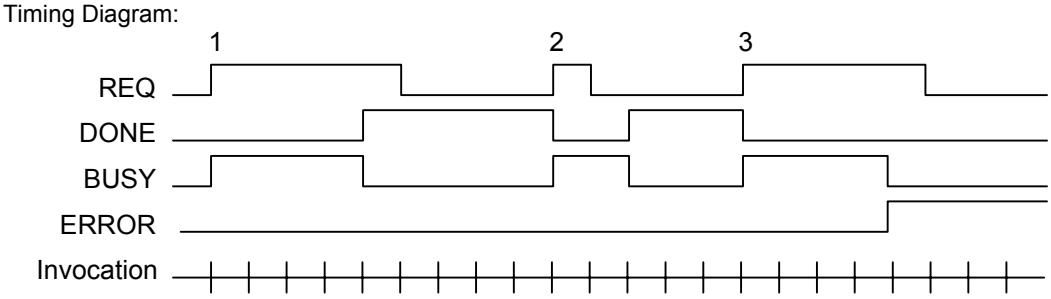
If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.

```
                          SALRM
          BOOL --- REQ              DONE --- BOOL
         DWORD --- ID               BUSY --- BOOL
           INT --- ATYPE           ERROR --- BOOL
           INT --- ASPEC         STATUS --- DWORD
           INT --- LEN
           ANY --- AINFO---    ---AINFO
```

```
FUNCTION_BLOCK SALRM  (* Generate alarm *)
VAR_INPUT
  REQ     : BOOL;        (* Request *)
  ID      : INT;         (* Handle of the slot of the alarm source *)
  ATYPE   : INT;         (* Type of the alarm *)
  ASPEC   : INT;         (* Specifier of the alarm *)
  LEN     : INT;         (* Length to be read *)
END_VAR

VAR_OUTPUT
  DONE   : BOOL;         (* Function done *)
  ERROR  : BOOL;         (* Error detected *)
  BUSY   : BOOL;         (* FB is busy *)
  STATUS : DWORD;        (* Last detected status *)
END_ VAR
VAR_IN_OUT
  AINFO  : ANY           (* Additional alarm information *)
END_VAR
```

Timing Diagram:



Case 1:
The REQ input remains 1 until the function block invocation has completed, it is reset by the user when DONE gets 1.

Case 2:
The user pulses the REQ input only for one invocation. The request is not aborted.

Case 3:
Like case 1, but an error occurred.

Figure 45 – SALRM function block

The following state diagram describes the algorithm of the SALRM function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the SALRM function block outputs.

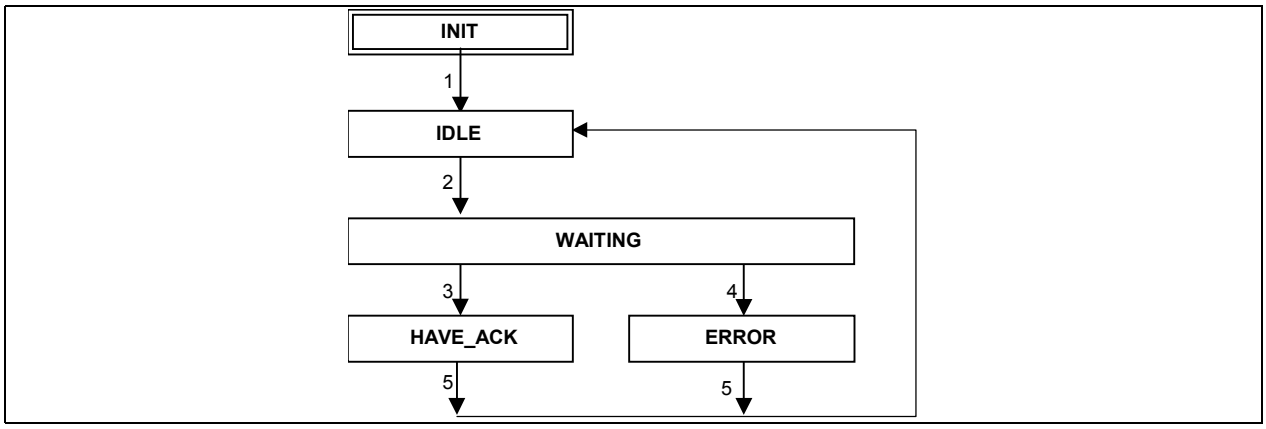


Figure 46 – State diagram of SALRM function block

The following table defines the transitions given in the state diagram above.

Table 41 - Transitions of the SALRM state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Positive response from remote communication partner: AlarmNotification.Cnf(+)
4	Negative response from remote communication partner or other communication problems detected: AlarmNotification.Cnf(-) or Abort.Ind or local problems
5	Next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, and STATUS.

Table 42 - Action table for SALRM state diagram

		FB outputs			
State	**Actions**	**DONE**	**BUSY**	**ERROR**	**STATUS**
INIT[1)]	Initialise outputs	0	0	0	0
IDLE	No actions	---	---	---	---
WAITING	Evaluate FB inputs. Request alarm notification: AlarmNotification.Req with AREP= ... Slot number= SLOT Alarm Type= ATYPE Seq Nr= increment last Seq Nr specific to the DP-Slave Alarm Specifier= ASPEC Add Ack= false Alarm Data= AINFO	0	1	0	-1 (is busy)
HAVE_ACK	No actions	1	0	0	0
ERROR	Indicate error	0	0	1	New error code

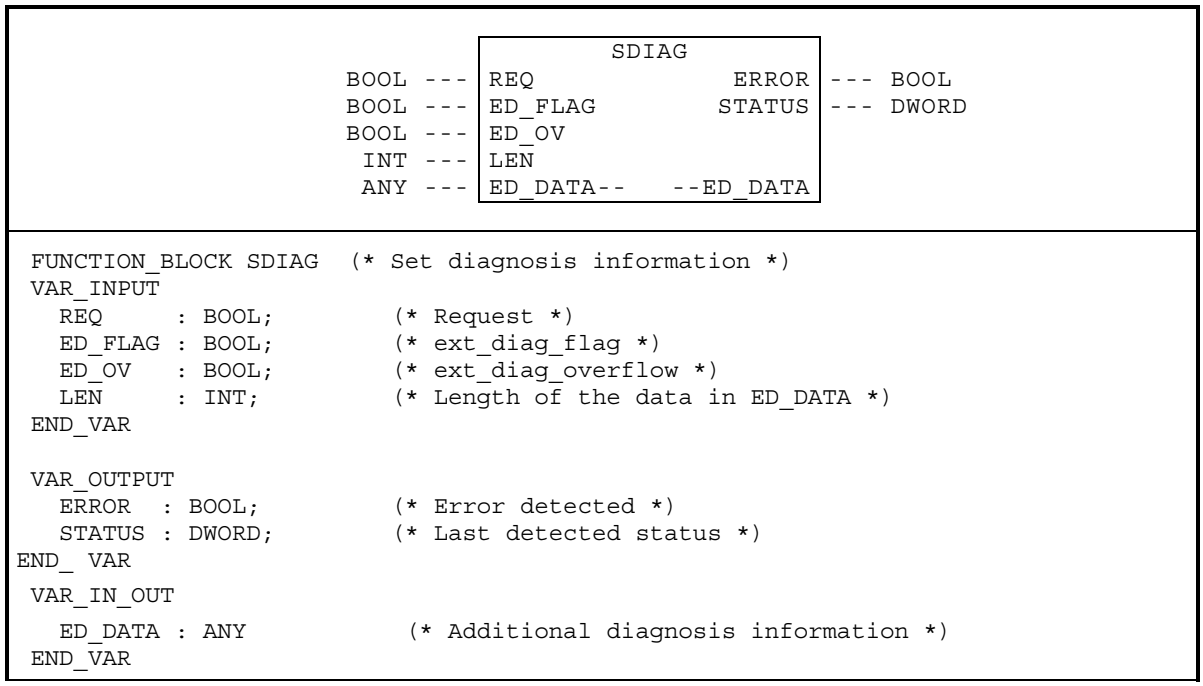
--- indicates "unchanged" FB outputs.
[1)] INIT is the cold start state.

5.4.2 Generate Diagnosis Information (SDIAG)

The communication function Generate Diagnosis Information for a DP-Slave uses the SDIAG function block defined in this clause. One instance of a SDIAG function block provides one instance of the PLC function Generate Diagnosis Information. The function is invoked when the REQ input is equal to 1.

The ED_FLAG input provides the ext_diag_flag information, the ED_OV provides the ext_diag_overflow information. The LEN input contains the length in byte of the additional diagnosis information stored in the ED_DATA parameter. The Variable given as ED_DATA parameter shall be at least of LEN byte. Possible value range of the LEN input is 0 .. 59.

If an error occurred, the ERROR output indicates an error and the STATUS output contains the error code. The STATUS values are defined in table 3.




```
FUNCTION_BLOCK SDIAG  (* Set diagnosis information *)
VAR_INPUT
  REQ     : BOOL;         (* Request *)
  ED_FLAG : BOOL;         (* ext_diag_flag *)
  ED_OV   : BOOL;         (* ext_diag_overflow *)
  LEN     : INT;          (* Length of the data in ED_DATA *)
END_VAR

VAR_OUTPUT
  ERROR  : BOOL;          (* Error detected *)
  STATUS : DWORD;         (* Last detected status *)
END_ VAR
VAR_IN_OUT
  ED_DATA : ANY           (* Additional diagnosis information *)
END_VAR
```

Figure 47 – SDIAG function block

The following state diagram describes the algorithm of the SDIAG function block. The following tables describe the transitions of this state diagram and the actions to be performed within the states and the settings of the SDIAG function block outputs.

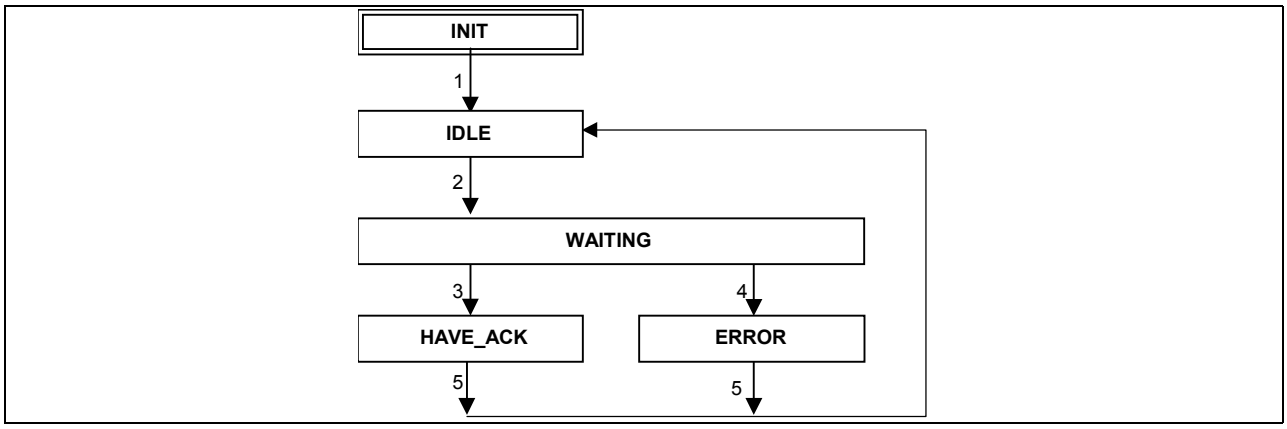


Figure 48 – State diagram of SDIAG function block

The following table defines the transitions given in the state diagram above.

Table 43 - Transitions of the SDIAG state diagram

Transition	Condition
1	Initialisation done
2	REQ = 1
3	Response from remote communication partner: SetSlaveDiag.Cnf
4	Communication problems detected: Abort.Ind or local problems
5	Next invocation of this instance

The following table defines the actions which are associated to the states given in the state diagram above. The actions set the output parameters VALID, BUSY, ERROR, and STATUS.

Table 44 - Action table for SDIAG state diagram

		FB outputs	
State	**Actions**	**ERROR**	**STATUS**
INIT[1)]	Initialise outputs	0	0
IDLE	No actions	---	---
WAITING	Evaluate FB inputs. Set diagnosis information: SetSlaveDiag.Req with AREP= ... Ext_Diag_Flag= ED_FLAG Ext_Diag_Overflow= ED_OV Length of Ext_Diag_Data= LEN Ext_Diag_Data= ED_DATA	0	---
HAVE_ACK	No actions	0	0
ERROR	Indicate error	1	New error code
--- indicates "unchanged" FB outputs. 1) INIT is the cold start state.			

6 Guidelines for application of Communication Function Blocks

6.1 Comm FB and Proxy FB

This clause of the specification provides some tutorial information for the user of the Comm FB defined in the previous clauses. The Comm FB can be used in two levels of PLC applications written in IEC 61131-3 programming languages as shown in the following figure:

1. In a DP Master the Comm FB can be used in application programs to communicate ***directly*** with DP-Slaves. This use is mainly offered to experts which are familiar with the communication specific functionality.

2. The Comm FB can also applied ***inside*** a so-called Proxy FB to achieve for its user a "hidden" communication with the DP-Slaves. The Proxy FB can represent the technological functionality of the DP-Slave completely or partly. In this case an expert for the DP communication and for the specific field device functionality provides a predefined standardised interface to the field device to the user of the Proxy FB.

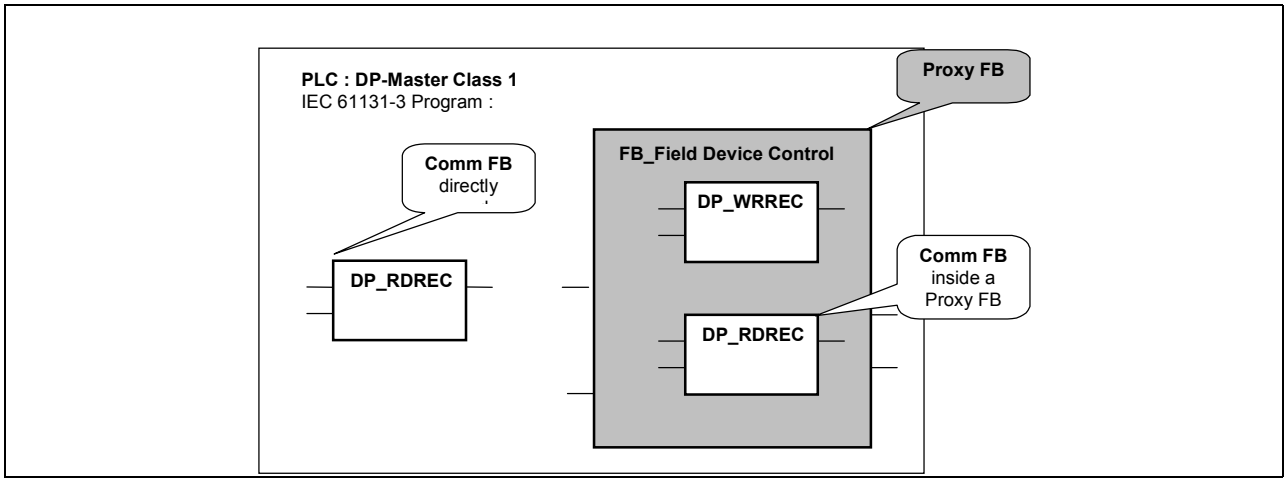


Figure 49 – Usage of Comm FB and Proxy FB in the PLC program

6.2 Mapping Technological Functionality to Proxy FB

There are two main concepts for mapping technological functionality to Proxy FB as illustrated in the following figure:

1. In the typical usage *one* ***single*** function block instance represents the device or the logical object in the application program. This is a *Proxy FB*.

   In this usage the invocation (call) of the FB passes *all* FB inputs via the actual parameters into the function block instance. This kind of FB has only one "method", which is performed by the single FB algorithm dependant on the history of the last FB invocation and the new actual parameter set.
   However various input data can cause different behaviour of the function block. For example one input can determine a specific "mode" or "method"

of the FB execution. In the following figure the so-called Proxy FB illustrates this usage.

2. In another usage a *set of **different*** function block types and instances represent the various "methods" applied to one device or logical object.
   This device or object needs not to be represented by a user accessible FB but may be a virtual object managing the internal device data, e.g. a separate instance or a common data structure. Each invocation of these various "method"-FB may have their specific set of parameters and may causes a different method on the device. Each of these invocations references the object representing the device for co-ordination and synchronisation. In the following figure the function blocks Methods1_FB, Methods2_ illustrate this usage.

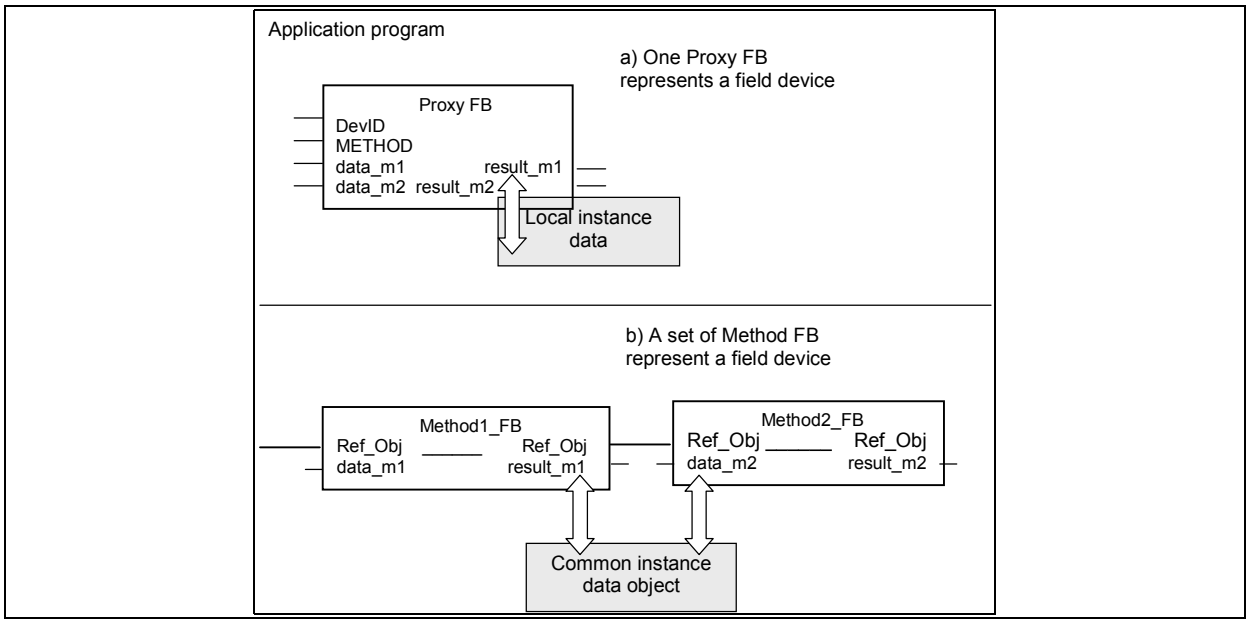


Figure 50 – Concepts of FB application
a) One function block as proxy
b) Set of function blocks as methods

6.3 Using Device IO

6.3.1 Integrated and External Device IO

The following example of a minimal PID controller (MiniPID Field Device) illustrates the two possible attachments of IO to a DP-Slave as field device 1. The PID controller receives cyclically the setpoint SP value via the DP interface from the DP-Master (Class 1) as a host and calculates the OUT value depending on the actual process value PV. The process data OUT and PV are *locally* attached the field device and presented to the DP interface of the field device.

The parameters PAR of the PID controller can be set and modified via a process data record as a data structure PAR. The current version of the parameters is for example contained in the data record PAR as byte 0..1 and is managed only in the host.

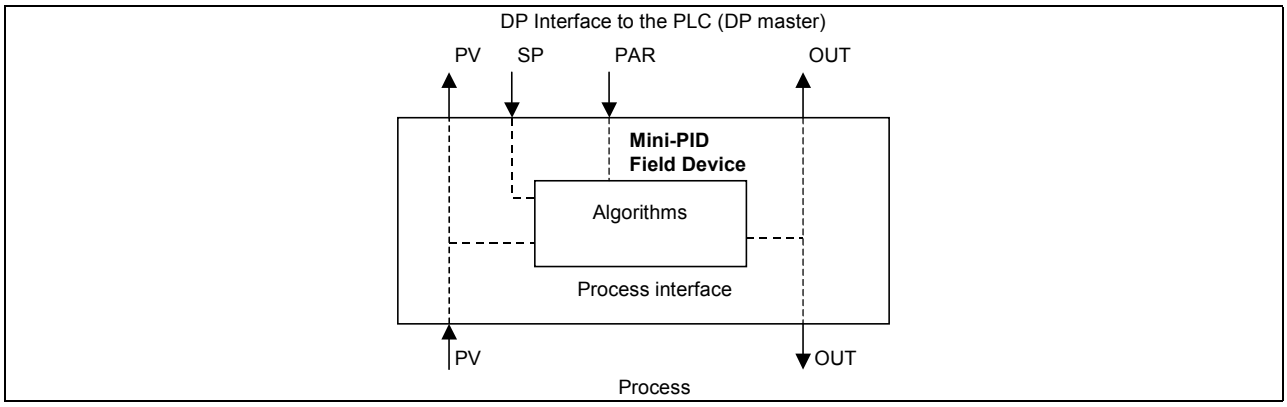


Figure 51 – Field device as DP-Slave with local IO

2. A other possible application is shown in the following figure. The Mini-PID field device puts and gets the process data PV and OUT by other field devices AI and AO via the PLC (host). In this case the following interface structure is applied.

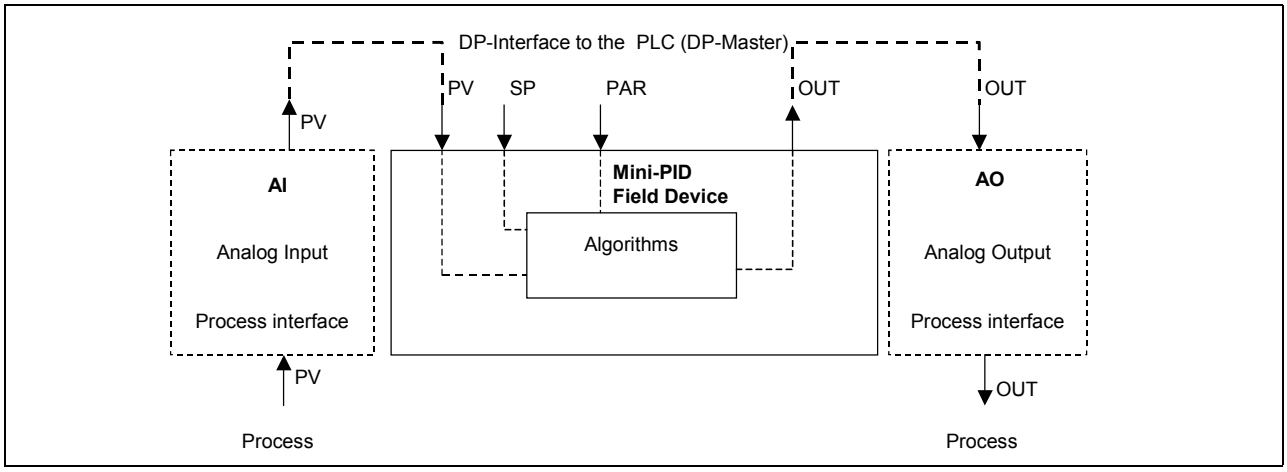


Figure 52 – Field device as DP-Slave with external IO

6.3.2 Proxy FB for a device with local IO

For the field device above shown with the local IO a simplified Proxy FB usable in the PLC (host) program is given in the next figure. The parameter ID identifies the field device, SP and PAR are input parameters for the PID, PV and OUT output the actual process values to the PLC program. REQ_PAR and VPAR serve for the actualisation of the data record PAR. Details are shown in the example program below in ST language

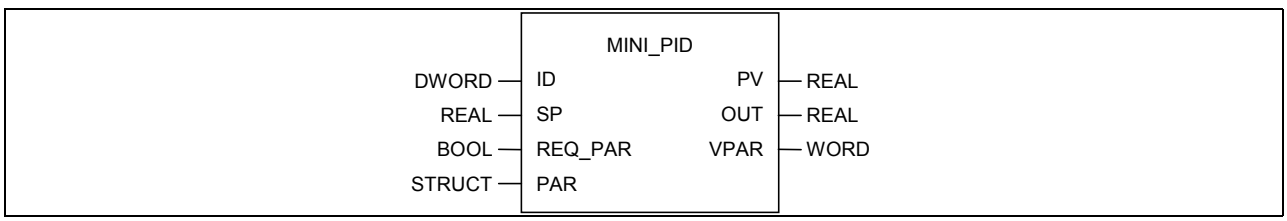


Figure 53 – Proxy FB MINI_PID with local IO

For each PID field device (DP-Slave) a Proxy FB MINI-PID is instanciated and the corresponding parameter ID is given to identify the device.

REQ_PAR=1 initiates the parameterisation, i.e. the data of PAR are sent to the device as data record 1. In VPAR the acknowledgement of the successful parameterisation is given as a version number in the VPAR variable.

The example program below shows the algorithm of the Proxy FB in IEC 61131 language Structured Text (ST).

```
FUNCTION_BLOCK MINI_PID;
VAR_IN
      ID: DWORD;
      SP: REAL;
      REQ_PAR: BOOL;
      PAR: STRUCT;
            VPAR: WORD;
            PARS: ARRAY [1..20] OF BYTE;
            END_STRUCT;
END_VAR;
VAR_OUT;
      PV, OUT: REAL;
      VPAR: WORD;
      END_VAR;
VAR
      OLD_REQ: BOOL;
      INS: STRUCT;
            PV, OUT: REAL;
            VPAR:WORD;
            END_STRUCT;
      RD_INS: GETIO;     (* Instance for reading the input data of slave  *)
      WR_OUTS:SETIO;     (* Instance for writing the output data of slave *)
      WRPAR:  WRREC;     (* Instance for writing parameters of slave      *)
END_VAR;

BEGIN
RD_INS(ID:= ID);
INS := RD_INS.INPUTS; (* transfers an array of bytes to a structure *)
WRPAR(REQ:= REQ_PAR; ID:= ID; MLEN:= 20; RECORD:= PAR.PARS);

IF    WRPAR.DONE THEN    (* parameterising successfully finished *)
      VPAR:= PAR.VPAR;
      :
      END_IF;
IF    WRPAR.ERROR THEN   (* parameterising failed *)
      :
      END_IF;
WR_OUTS(ID:= ID; OUTPUTS:= SP; LEN := 10);
END;
```

Alternatively the algorithm shown in the following figure may be used to parameterise the field device. This algorithm uses the BUSY output to avoid to call the WRPAR instance and to poll the result of the parameterisation.

```
IF (REQ = true) AND (OLD_REQ = false)
THEN IF NOT WRPAR.BUSY
     THEN  (* first call *)
          WRPAR(REQ:= true; ID:= ID; LEN:= 20; RECORD:= PAR.PARS);
     ELSE
          :      (* error handler: REQ_PAR on busy parameterisation *)
     END_IF;
END_IF;
IF WRPAR.BUSY THEN WRPAR (); (* next invocations with the same parameters *)
END_IF;
IF WRPAR.DONE THEN            (* parameterising successfully finished *)
    VPAR:= PAR.VPAR;
    :
    END_IF;
IF WRPAR.ERROR THEN           (* parameterising failed *)
    :
    END_IF;
OLD_REQ:= REQ;
```

Die output BUSY of the WRREC can be used for reducing the invocations and to achieve a simple error handling upon REQ_PAR=1 during running parameterising.

6.3.3 Proxy FB for a field device with IO via the process image

If the cyclic data from and to the field device is mapped via the *process image* the actual process variable PV is used as an *input* of the Proxy FB. The setpoint OUT has to be written in the process output image.

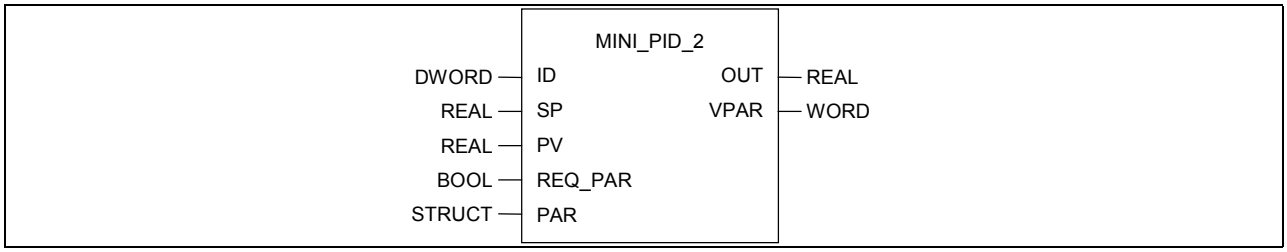


Figure 54 – Proxy FB MINI_PID_2 with IO via the process image

In this case the FB input SP and the FB output OUT have to be the corresponding addresses in the process image.

Using this FB the inputs and /or outputs of the PID functionality may be integrated into a PID field device or the PID functionnality may get its input and outputs from an other field device.

The example program below shows the algorithm of the Proxy FB MINI_PID_2 in IEC 61131 language Structured Text (ST).

```
FUNCTION_BLOCK MINI_PID_2;
VAR_IN
  ID: DWORD;
  SP, PV: REAL;
  REQ_PAR: BOOL;
  PAR: STRUCT;
        VPAR: WORD;
        PARS: ARRAY [1..20] OF BYTE;
        END_STRUCT;
END_VAR;
VAR_OUT;
  OUT: REAL;
  VPAR: WORD;
END_VAR;
VAR
  OLD_REQ: BOOL;
  INS: STRUCT;
        PV, OUT: REAL;
        VPAR:WORD;
        END_STRUCT;
  OUTS: REAL;
  RD_INS:      GETIO;
  WR_OUTS:     SETIO;
  WRPAR:       WRREC;
  :
END_VAR;

BEGIN
WRPAR(REQ:= REQ_PAR; ID:= ID; LEN:= 20; RECORD:= PAR.PARS);
  (* permanent invocations for updating of outputs *)
IF WRPAR.DONE THEN VPAR:= PAR.VPAR;
                        (* parameterising successfully finished *)
  END_IF;
IF WRPAR.ERROR THEN ...(* parameterising failed *)
END_IF;
END
```

6.3.4 Some Recommendations

1. The usage of the BUSY output is useful for the application of the asynchronuously executed function blocks.

1. Comparing Proxy FB using an integrated IO interface or using IO via process image shows some differences:

   Proxy FB with integrated IO interface (e.g. FB MINI_PID):

   - The interface of the Proxy FB represents the complete DP interface of its technological functiuonality.

   - The Field Device can be represented by one Proxy FB.

   - The Field Device is identified only by one parameter ID.

   Proxy FB with IO via process image (e.g. FB MINI_PID_2):

- The same FB interface may be used with different configurations:

  The IO are integrated in the Field Device the Proxy FB represents.
  The IO come from a different device.

The FB itself may contain the technological functionality or parts of it.

- In the case of a Field Device with integrated IO the interface of this field device is splitted to the Proxy FB and the process image.

6.4 Scheduling of Function Blocks

Accordíng IEC 61131-3 the function block instance can be

- *invoked* (called) in the body of another Program Organisation Unit (POU) like a program or a function block which is in the next (higher) level of the hierarchy and/or

- *scheduled* by an associated Task which is defined according IEC 61131-3 on a periodic basis or upon the occurence of specified "events" and conditions.

That means if a funktion block instance does not have an association to an IEC 61131-3 task the execution of its algorithm is a part of the execution of the invoking POU.

In the figure below PROGRAM_1 is scheduled periodically with a task (100 ms cycle) and it invokes the included function block instances M_2 like subroutines.

Otherwise if a function block instance is associated with a task it shall be under the *exclusive* control of the task, independant of the rules of evaluation of the program organisation unit (FB or program) in which the function block instance is declared.

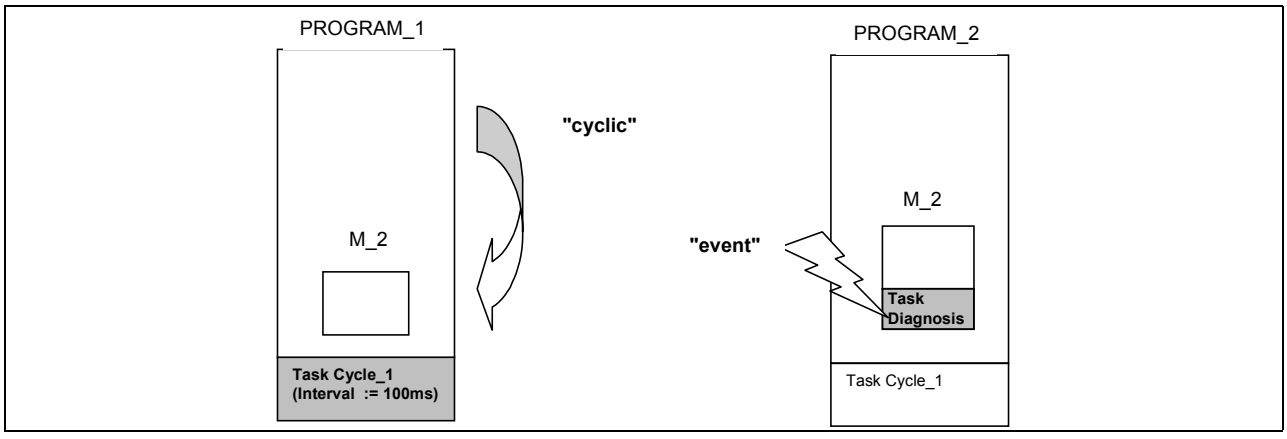


Figure 55 – Scheduling of a function block
a) in a task scheduled program (cyclic) - b) by a direct task association (event)

Proxy Function Blocks representing a field device may need for performance reason like response time to be *scheduled* in multiple tasks. Therefore the PLC Programming System and the PLC Runtime System may support one of the following solutions:

- The graphical and/or textual representation of the *same* function block instance invoked in *multiple* programs as illustrated in the figure below.
  In this case the function block instance M_2 is scheduled as well as cyclically

by the invocation in PROGRAM_1 and event driven upon an occurrence of an diagnosis event in PROGRAM_2.

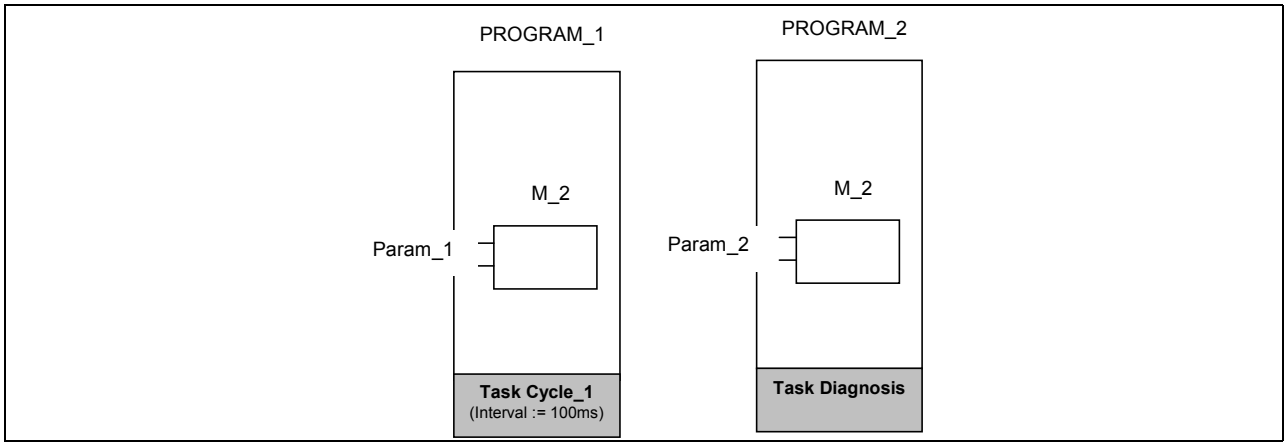


Figure 56 – Multiple scheduling of a FB instance by invokations in different programs

- The graphical and/or textual representation of *multiple* invokations of the same function block instance with different explicit task associations in the *same* program.
  In this case the function block instance M_2 is scheduled as well as by a cyclic task and by a diagnosis task.

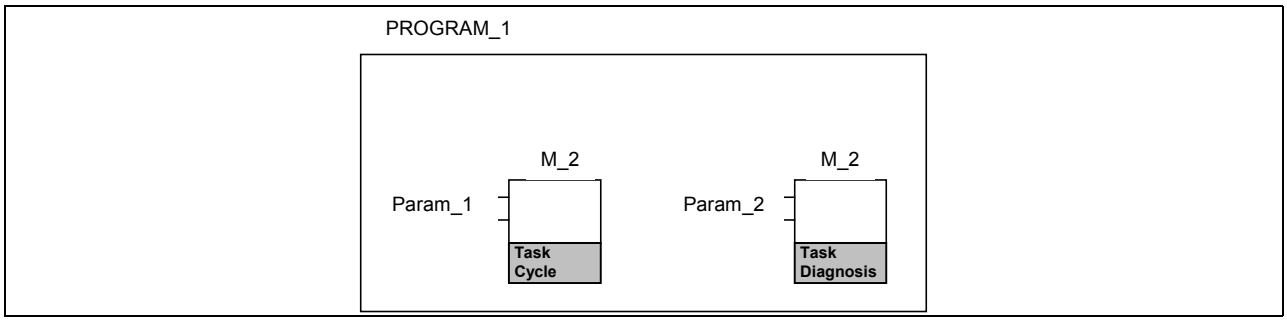


Figure 57 – Multiple scheduling of a FB instance by different invokations in the same programs

Some provision may be necessary for both concepts of multiple scheduling of function blocks:

- Specific invokation parameters can be used for the distinction which task has actually scheduled the FB; i.e. the indication of the actual invokation by a specific parameter can be used from inside the FB to apply the suitable "method".

- Since the multiple scheduling may cause an interrupt of the current function block execution the so-called reentrant programming technique provides the necessary data consistency; i.e. the concurrent access to common instance data has to be mutually exclusive using e.g. semaphores.

Annex A - Compliance Table

The following table lists all Comm Functions and Comm FB defined in this specification.
A manufacturer which claim to be compliant with this PNO specification shall provide a list in the format of this table and shall identify all compliant Com Functions and Function Blocks.

Table A.1 - Compliance table

No	Comm FB Comm FCT	Explanation	implementation specific additional information	com-pliant (Y/N)
		Address Functions		
1	ID	Function for conversion of a physical address to the handle		
2	ADDR	Function for conversion of a handle to the physical address		
3	SLOT	Function for addressing a slot of a DP-Slave		
4	NSLOT	Function for addressing the next slot of a DP-Slave		
		Comm FB for DP-Master (Class 1)		
5	GETIO	Get I/O Data		
6	SETIO	Set I/O Data		
7	RDREC	Read Process Data Record		
8	WRREC	Write Process Data Record		
9	RALRM	Receive Alarm		
10	RDIAG	Read Diagnosis		
11	SYCFR	Synchronise or freeze I/O		
12	ICTRL	"Interlocked Control"		
		Comm FB for DP-Master (Class 2)		
13	RDIN	Read input data of a DP-Slave		
14	RDOUT	Read output data of a DP-Slave		
15	RDREC	Read a process data record from a slot of a DP-Slave		
16	WRREC	Write a process data record to a slot of a DP-Slave		
17	RDIAG	Read diagnosis information from a DP-Slave		
18	CNCT	Manage a connection to a DP-Slave		
		Comm FB for DP-Slave		
19	RCVCO.	Receives the output data of a DP-Master		
20	SBCCI	Subscribe input data of another DP-Slave		
21	PRVCI	Provides (publishes) the input data of the DP-Slave		
22	RCVREC	Receives a process data record from a DP-Master		
23	PRVREC	Receives a request and provides a process data record		
24	SALRM	Request to send an alarm to the DP-Master (Class 1)		
25	SDIAG	Request to send diagnosis to a DP-Master		

In the following table the permitted "*Implementation dependant features"* shall be listed.

Table A.2 - Implementation dependant features

Clause	Feature	Implementation chosen
3.2	Maximum length of I/O data of one DP-Slave supported by a DP-Master (Class 1)	
3.2	Maximum length of I/O data of one slot of a DP-Slave supported by a DP-Master (Class 1)	
3.2	Maximum length of consistent I/O data of one slot of a DP-Slave supported by a DP-Master (Class 1)	
4.2	Maximum length of I/O data of one DP-Slave supported by a DP-Master (Class 2)	
4.2	Maximum length of I/O data of one slot of a DP-Slave supported by a DP-Master (Class 2)	
4.2	Maximum length of consistent I/O data of one slot of a DP-Slave supported by a DP-Master (Class 2)	

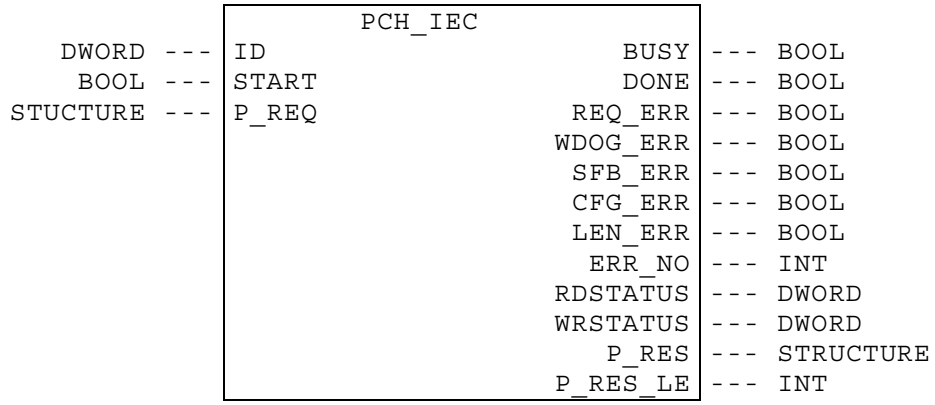
Annex B – Application example „DPV1 – Parameter Channel“

The function block “PCH_IEC” can be used for data exchange between a PLC and a drive which supports the DPV1 parameter channel (for detailed information see PROFIBUS guideline - Order no. 3.171). Both devices have to support the functionality of IEC 61131-3.
To determine the handle of the driver out of its physical identification the block “ID” is used. The necessary input variables are supplied via input variables of “PCH_IEC”. If no slot exists at the given physical address “CFC_ERR” is set and no data are transmitted until “ID” signalizes that a slot exists.

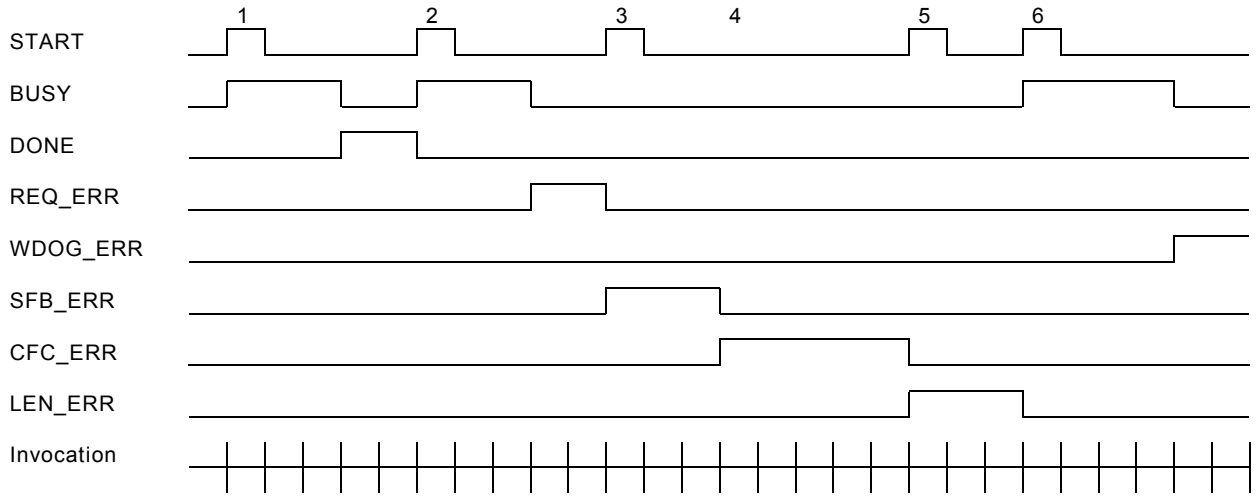
The user has to enter the request in the input structure (“P_REQ”). With a rising edge at the input “START” the new request is accepted. Before transmitting the new request it will be checked whether it’s data are plausible. This means:

- the number of parameter is between 1 through 37 — if not: ERR_NO = 1 and LEN_ERR = 1
- the number of elements is between 0 through 117 — if not: ERR_NO = 2 and LEN_ERR = 1
- the format for parameter change is byte, word or dword — if not: ERR_NO = 3 and LEN_ERR = 1
- the request ID is 0x01 or 0x02 — if not: ERR_NO = 4 and LEN_ERR = 1
- the length of the telegram is less or equal 240 bytes — if not: ERR_NO = 5 and LEN_ERR = 1

After this the telegram is transmitted to the drive with the communication function WRREC and the block “PCH_IEC” is waiting for the response which will be received with the communication function RDREC. If this function block signalizes that new data are received the response ID is compared with the request ID whether it’s the right answer to the sent request.
The response is displayed at the output structure (“P_RES”). From there the user can pick up the data for further usage.



Timing diagram



The START input has to be reset by the user when the block signals BUSY, SFB_ERR, CFC_ERR or LEN_ERR. A new request can only be started by setting START if one of the following outputs is set: DONE, REQ_ERR, WDOG_ERR, SFB_ERR, CFC_ERR or LEN_ERR. If SFB_ERR, CFC_ERR or LEN_ERR is set at least for one cycle START has to be reset before setting it again.

Case 1:
Shows the time characteristic of a request which is finished without error.

Case 2:
Shows the time characteristic of a request which is finished with an error.

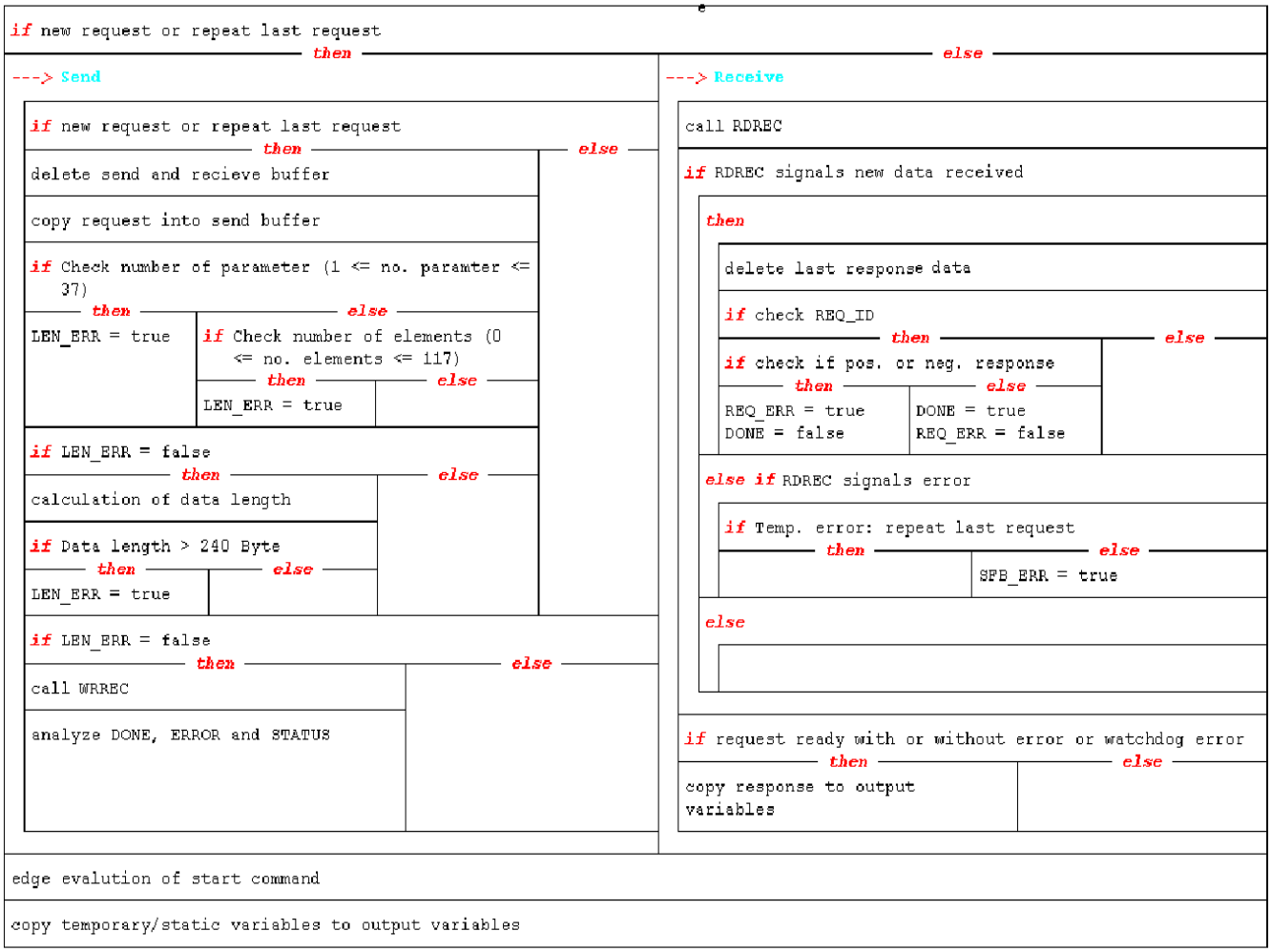
Case 3:
Shows the time characteristic if a SFB_ERR occurs.

Case 4:
A CFC_ERR can always occur without starting a new job. It occurs if no handle for the given slot exits.

Case 5:
Shows the time characteristic if a LEN_ERR occurs.

Case 6:
Shows the time characteristic if a WDOG_ERR occurs.

Structure chart



In the following program listing the headlines out of the structure chart are **bold**.

```
FUNCTION_BLOCK PCH_IEC

VAR_INPUT
    ID              :DWORD;          //  ID of the slot of a DP-Slave
    START           :BOOL;           //  Accept start impulse for request
    P_REQ           :STRUCT          //  Parameter request
                     DATA_000    : BYTE;
                     …
                     DATA_239    : BYTE;
                    END_STRUCT;
END_VAR
VAR_OUTPUT
    BUSY            :BOOL;     //  Request busy
    DONE            :BOOL;     //  Request completed without error
    REQ_ERR         :BOOL;     //  Request completed with error
    WDOG_ERR        :BOOL;  //  Watchdog error, no plausible response data exist
    SFB_ERR         :BOOL;     //  WRREC / RDREC signals error
    CFG_ERR         :BOOL;     //  Slave incorrectly configured or not at all
    LEN_ERR         :BOOL;     //  Length calculation error
    ERR_NO          :INT;      //  Explanation to LEN_ERR
    RDSTATUS        :DWORD;    //  STATUS of DP_RDREC
    WRSTATUS        :DWORD;    //  STATUS of DP_WRREC
    P_RES           :STRUCT    //  Parameter response
                     DATA_000    : BYTE;
                      …
                     DATA_239    : BYTE;
                    END_STRUCT;
    P_RES_LE        :INT;      //  Number of received data
END_VAR

//Static Variables
VAR
    sb_SEND                 : BOOL;  // Transmission active or request repetition
    sb_SFB_ERR              : BOOL;  // SFB error
    sb_BUSY                 : BOOL;  // request still active
    sb_START_OLD            : BOOL;  // Start flag last cycle
    sb_DONE                 : BOOL;  // Data exchange ready without error
    sb_REQ_ERR              : BOOL;  // Data exchange ready with error
    sb_WDOG_ERR             : BOOL;  // Watchdog monitoring exceeded
    sb_WR_ERROR             : BOOL;  // SFC error DP_WRREC
    sb_RD_ERROR             : BOOL;  // SFC error DP_RDREC
    sb_LEN_ERR              : BOOL;  // Length of request is wrong
    sb_WR_REQ               : BOOL;  // Write request
    sb_RD_REQ               : BOOL;  // Read request
    si_HM_RET_VAL           : INT;   // Return value of DP_RDREC and DP_WRREC
    si_WDOG_RETRY           : INT;   // Counter for request retry
                                        for Watchdog monitoring
    si_NODATA_RETRY         : INT;   // Counter for request retry
                                        if none Response data exist
    si_NODATA_CYCLE         : INT;   // Cycle counter, if no response data exists
    si_WDOG_RETRY_NO        : INT := 05;  // Number of request retries,
                                      when no plausible response data exists
    si_NODATA_RETRY_NO      : INT := 05;  // Number of request retries,
                                    when no response data exist
    si_NODATA_CYCLE_NO      : INT := 2500;        // Number of cycles
                                    when no response data exist, before request retry
    si_LEN                  : INT;                // message length
    si_NO_PARA              : INT;                // number of parameter
    si_FORMAT               : INT;                // data type of values
    si_ERR_NO               : INT;                // Explanation to LEN_ERR
    sd_DP_WRREC_RET_VAL     : DWORD;              // Status of DP_WRREC
    sd_DP_RDREC_RET_VAL     : DWORD;              // Status of DP_RDREC
    sa_SEND_BUF             : ARRAY [0..239] of Byte;        // Send buffer
    sa_RECV_BUF             : ARRAY [0..239] of Byte;        // Receive buffer
    DP_WRREC                : WRREC;
    DP_RDREC                : RDREC;
END_VAR
```



```
//Temporary Variables

VAR_TEMP
    pb_CFG_ERR              : BOOL;     // CFG_ERR
    pb_WR_DONE              : BOOL;     // DP_WRREC ready without error
    pb_WR_BUSY              : BOOL;     // DP_WRREC busy
    pb_RD_VALID             : BOOL;     // DP_RDREC new data received
    pb_RD_BUSY              : BOOL;     // DP_RDREC busy
    pw_STATUS               : WORD;     // needed to determinate error code
    pi_INDEX                : INT;      // Loop counter
    pd_STATUS               : DWORD;    // needed to determinate error code
    pd_RET_VAL_DP_ID        : DWORD;    // Return value of DP_ID
END_VAR

//Code Section

pd_RET_VAL_DP_ID := ID

IF sb_BUSY
    THEN
        //wait for response

        IF si_NODATA_CYCLE < si_NODATA_CYCLE_NO
        THEN
            //Increase number of cycle
            si_NODATA_CYCLE := si_NODATA_CYCLE + 1;
        ELSE;
            IF si_HM_RET_VAL < 0
            THEN
                IF si_NODATA_RETRY < si_NODATA_RETRY_NO
                THEN
                    //increase number of repeated requests
                    si_NODATA_RETRY := si_NODATA_RETRY + 1;

                    //initialize number of cycle
                    si_NODATA_CYCLE := 0;

                    //repeat last request
                    sb_SEND := TRUE;
                ELSE;
                    //SFC - error
                    sb_SFB_ERR := TRUE;

                    //reset request repeat
                    sb_SEND := FALSE;

                    //request not active
                    sb_BUSY := FALSE;
                END_IF;  //END_IF of IF si_NODATA_RETRY < si_NODATA_RETRY
            ELSE;
                //not used
            END_IF;

            //END_IF of IF si_HM_RET_VAL < 0
        END_IF; //END_IF of IF si_NODATA_CYCLE < si_NODATA_CYCLE_NO
    ELSE;
        //initialize number of cycle and number of request repeats
        si_NODATA_CYCLE := 0;
        si_NODATA_RETRY := 0;
        si_WDOG_RETRY := 0;
    END_IF; //END_IF of IF BUSY

    //Reset temp. variables
    sb_RD_ERROR := FALSE;
    sb_WR_ERROR := FALSE;

    IF (NOT sb_BUSY AND START AND NOT sb_START_OLD) OR (sb_BUSY AND sb_SEND)
    THEN
```



```
//Send

  IF START AND NOT sb_START_OLD AND NOT sb_SEND
  THEN
//delete send and receive buffer

      FOR pi_INDEX :=0 to 239
       DO
          sa_SEND_BUF[pi_INDEX] := 16#0;
          sa_RECV_BUF[pi_INDEX] := 16#0;
      END_FOR;

//copy request into send buffer

      sa_SEND_BUF[0] := P_REQ.DATA_000;
      …
      sa_SEND_BUF[239] := P_REQ.DATA_239;

      //Reset internal variables
      sb_DONE := FALSE;
      sb_REQ_ERR := FALSE;
      sb_WDOG_ERR := FALSE;
      sb_SFB_ERR := FALSE;
      sd_DP_WRREC_RET_VAL := 16#0;
      sd_DP_RDREC_RET_VAL := 16#0;
      sb_LEN_ERR := FALSE;
      si_ERR_NO := 0;

      //Set start command
      sb_WR_REQ := TRUE;

      //check important numbers in request

      //Check number of parameter (1 <= no. parameter <= 37)

      IF (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[3])) < 1)
            OR (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[3])) > 37)
      THEN
          sb_LEN_ERR := TRUE;
          si_ERR_NO := 1;
      ELSE;
          FOR pi_INDEX := 0
             TO (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[3]))-1)
           DO
          //Check number of elements (0 <= no. elements <= 117)

              IF (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[5+
                   (pi_INDEX * 6)])) < 0) OR (WORD_TO_INT
                   (BYTE_TO_WORD (sa_SEND_BUF[5+(pi_INDEX * 6)])) > 117)
              THEN
                  sb_LEN_ERR:= TRUE;
                  si_ERR_NO := 2;
              ELSE;
                  //not used
              END_IF;
          END_FOR;
       END_IF;

      IF NOT sb_LEN_ERR
      THEN
      //calculation of data length

          //Calculation of data length for parameter request

          IF sa_SEND_BUF[1] = 16#01
          THEN
     // Length = parameter-header + parameteraddress * number of parameter

              si_LEN := 4 + (6 * (WORD_TO_INT
                         (BYTE_TO_WORD (sa_SEND_BUF[3]))));
```

```
                //Calculation of data length for parameter change

                ELSIF sa_SEND_BUF[1] = 16#02
                THEN
                    // Length = parameter-header + (parameteraddress +
parametervalue-header + format * number of values)1 + ... + (parameteraddress +
parametervalue-header + format * number of values)n

                   si_NO_PARA := WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[3]));
                         // number of parameter
                   pi_INDEX := 1;       // loop counter
                   si_LEN := 4 + 6 * si_NO_PARA;
                         // startaddress of 1. parameter value

                    REPEAT
                      CASE (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[si_LEN]))) OF

                            //format: byte
                            65:
                                si_FORMAT := 1;

                            //format: word
                            66:
                                si_FORMAT := 2;

                            //format: dword
                            67:
                                si_FORMAT := 4;
                            ELSE :
                                sb_LEN_ERR := TRUE;
                                si_ERR_NO := 3;
                                exit;    // quit loop
                        END_CASE;

                            si_LEN := si_LEN + (si_FORMAT *
                       (WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[si_LEN + 1]))))+2;
                            pi_INDEX := pi_INDEX + 1;

                    UNTIL pi_INDEX > si_NO_PARA END_REPEAT;

                ELSE
                    sb_LEN_ERR := TRUE;
                    si_ERR_NO := 4;
                END_IF;        //END_IF of IF sa_SEND_BUF[1] = 16#01

            //Data length > 240 Byte

                IF si_LEN > 240
                THEN
                    sb_LEN_ERR := TRUE;
                    si_ERR_NO := 5;
                ELSE;
                    //not used
                END_IF;        //END_IF of IF si_LEN > 240
            ELSE;
                //not used
            END_IF             //END_IF of IF NOT sb_LEN_ERR
        ELSE;
            //not used
        END_IF;      //END_IF of IF START AND NOT sb_START_OLD AND NOT sb_SEND

        IF NOT sb_LEN_ERR
        THEN

      //call DP_WRREC

            DP_WRREC (REQ       := sb_WR_REQ,
                      ID        := pd_RET_VAL_DP_ID,
                      INDEX     := WORD_TO_INT (BYTE_TO_WORD (sa_SEND_BUF[0])),
```



```
                LEN       := si_LEN,
                RECORD    := sa_SEND_BUF);

        DP_WRREC.DONE      := pb_WR_DONE;
        DP_WRREC.BUSY      := pb_WR_BUSY;
        DP_WRREC.ERROR     := pb_WR_ERROR;
        DP_WRREC.STATUS    := sd_DP_WRREC_RET_VAL;

    //analyze DONE, ERROR and STATUS

        //Data transfere ready without error

        IF pb_WR_DONE
        THEN
            sb_BUSY := TRUE;
            sb_SEND := FALSE;

            //Reset start command
            sb_WR_REQ := FALSE;

        //Data transfer ready with error

        ELSIF sb_WR_ERROR
        THEN
            pd_STATUS := SHR (IN:=sd_DP_WRREC_RET_VAL, N:=8);
            pw_STATUS := DWORD_TO_WORD (pd_STATUS);
            si_HM_RET_VAL := WORD_TO_INT (pw_STATUS);

            //Temp. error: repeat last request

            IF (pw_STATUS = 16#80A7) or (pw_STATUS = 16#80C2)
            THEN
                sb_SEND := TRUE;
                sb_BUSY := TRUE;
                sb_WR_ERROR := FALSE;
                sd_DP_WRREC_RET_VAL := 16#0;
            ELSE;
                sb_BUSY := FALSE;
                sb_SEND := FALSE;
                sb_WR_ERROR := TRUE;

                //Reset start command
                sb_WR_REQ := FALSE;
            END_IF;

        ELSE
            sb_SEND := TRUE;
            sb_BUSY := TRUE;

            //Reset start command
            sb_WR_REQ := FALSE;
        END_IF;            //END_IF of IF pb_WR_DONE
    ELSE;
        sb_BUSY := FALSE;
        sb_DONE := FALSE;
        sb_SEND := FALSE;
    END_IF;                //END_IF of IF NOT sb_LEN_ERR
ELSE;
  //Receive

    IF sb_BUSY
    THEN
        IF si_WDOG_RETRY < si_WDOG_RETRY_NO
        THEN
            //Set start command
            sb_RD_REQ := TRUE;

        //call DP_RDREC

            DP_RDREC (REQ       := sb_RD_REQ,
```

```
                ID        := pd_RET_VAL_DP_ID,
                INDEX     := WORD_TO_INT
                             (BYTE_TO_WORD (sa_SEND_BUF[0])),
                MLEN      := 240,
                RECORD    := sa_RECV_BUF);

    DP_RDREC.VALID     := pb_RD_VALID;
    DP_RDREC.BUSY      := pb_RD_BUSY;
    DP_RDREC.ERROR     := pb_RD_ERROR;
    DP_RDREC.STATUS    := sd_DP_RDREC_RET_VAL;
    DP_RDREC.LEN       := P_RES_LE;

//DP_RDREC signals new data received

    IF pb_RD_VALID
    THEN
//delete last response data

        P_RES.DATA_000 := 16#0;
        …
        P_RES.DATA_239 := 16#0;

        //Reset start command
        sb_WR_REQ := FALSE;

//check REQ ID

        IF sa_SEND_BUF[0] = sa_RECV_BUF[0]
        THEN

//check if pos. or neg. response

            IF sa_RECV_BUF[1] = 16#81 OR sa_RECV_BUF[1] = 16#82
            THEN
                sb_REQ_ERR := TRUE;
                sb_DONE := FALSE;
            ELSE;
                sb_REQ_ERR := FALSE;
                sb_DONE := TRUE;
            END_IF;
        ELSE;
            //not used
            ;
        END_IF;    //END_IF of IF sa_SEND_BUF[0] := sa_RECV_BUF[0]

        //request ready with or without error

        IF sb_DONE OR sb_REQ_ERR
        THEN
            //job no longer active
            sb_BUSY := FALSE;
        ELSE;
    //increase number of repeats while no valid response is received
            si_WDOG_RETRY := si_WDOG_RETRY + 1;

            //initialize number of cycle
            si_NODATA_CYCLE := 0;

            //Repeat request
            sb_SEND := TRUE;
        END_IF;    //END_IF of IF DONE OR REQ_ERR

//DP_RDREC signals error

    ELSIF sb_RD_ERROR
    THEN
        pd_STATUS := SHR (IN:=sd_DP_RDREC_RET_VAL, N:=8);
        pw_STATUS := DWORD_TO_WORD (pd_STATUS);
        si_HM_RET_VAL := WORD_TO_INT (pw_STATUS);
```

```
                    //Temp. error: repeat last request

                    IF (pw_STATUS = 16#80A7) or (pw_STATUS = 16#80C2)
                    THEN
                        sb_SEND := false;
                        sb_BUSY := TRUE;
                        sb_RD_ERROR := FALSE;
                        sd_DP_RDREC_RET_VAL := 0;
                    ELSE;
                        sb_BUSY := FALSE;
                        sb_SEND := FALSE;
                        sb_RD_ERROR := TRUE;
                    END_IF;

                ELSE
                    //not used
                    ;
                END_IF;  //END_IF of IF pb_RD_VALID
            ELSE;
                //sign watchdog - error
                sb_WDOG_ERR := TRUE;

                //job not active anymore
                sb_BUSY := FALSE;
            END_IF;      //END_IF of IF si_WDOG_RETRY < si_WDOG_RETRY_NO

      //request ready with or without error or watchdog error

            IF sb_WDOG_ERR OR sb_DONE OR sb_REQ_ERR
            THEN
      //copy response to output variables

                P_RES.DATA_000 := sa_RECV_BUF[0];
                …
                P_RES.DATA_239 := sa_RECV_BUF[239];
            ELSE;
                //not used
                ;
            END_IF;
        ELSE;
            //not used
        END_IF;    //END_IF of IF BUSY
    END_IF;


//edge evaluation of start command
sb_START_OLD := START;

//copy temporary/static variables to output variables

CFG_ERR := pb_CFG_ERR;
BUSY := sb_BUSY;
DONE := sb_DONE;
REQ_ERR := sb_REQ_ERR;
WDOG_ERR := sb_WDOG_ERR;
SFB_ERR := sb_SFB_ERR OR sb_WR_ERROR OR sb_RD_ERROR;
RDSTATUS := sd_DP_RDREC_RET_VAL;
WRSTATUS := sd_DP_WRREC_RET_VAL;
LEN_ERR := sb_LEN_ERR;
ERR_NO := si_ERR_NO;

END_FUNCTION_BLOCK
```

PROFIBUS Nutzerorganisation e.V.
Haid-und-Neu-Str. 7
D-76131 Karlsruhe

Phone: ++ 721 / 96 58 590
Fax: ++ 721 / 96 58 589
PROFIBUS_International@compuserve.com

www.profibus.com