502 077 02

P-NET CONTROLLER

PD 5000

Manual

GB *August 2001*

PROCES-DATA A/S, Navervej 8, DK-8600 Silkeborg, Denmark, Phone +45 87 200 300, Fax + 45 87 200 301

Contents **Page**

1 General information.

The PD 5000-Series of controllers has been developed as the 2nd generation of P-NET fieldbus controllers (masters), for use as distributed computing elements within either highly complex or simple process control systems.

Together with possible additional controllers and PC's, the PD 5000 is used in conjunction with other distributed input/output modules (slaves), such as digital, analogue, flow, and weight nodes, to form a complete control system, which all communicate via the P-NET fieldbus (EN 50170 Vol. 1).

Since the Controller offers dual P-NET ports together with an RS232 connection, it can be used to link two P-NET sections (multi-net), or as a gateway to other communications media, including PC's, modem's etc. Although the PD 5000 can be utilised independently as a powerful processing element, the PD 5010 and PD 5015 enable a controller to be used as an operator input and display interface. Furthermore, the PD 5020 enables the controller to be used with a VGA monitor, and a standard PC keyboard and a mouse, as a Supervisory Control System.

The PD 5000-Series can be ordered in different configurations:

As a PD 5000, a controller without keyboard/display.

As a PD 5020 controller, where the PD 5000 controller is equipped with a video card for a VGA monitor (640 * 480 pixels). The PD 5020 has connections for a standard PS/2 mouse and keyboard, 128 k Bytes of EPROM, 1.5 M Byte of FLASH and 2.5 M Byte of RAM with lithium battery back-up.

As a PD 5010 or PD 5015 controller, where the PD 5000 controller is equipped with a keyboard/display unit. The PD 5010 controller has a backlit graphics LCD display with 256 * 64 pixels, and a 48 key sealed membrane click-switch keyboard. The PD 5015 controller has a backlit graphics LCD display with 240 * 128 pixels, and a 44 key sealed membrane click-switch keyboard.

SYSTEM DESCRIPTION

Programming

The Controller is programmed in Process-Pascal, which is an extension of standard Pascal, allowing easy declaration and utilisation of P-NET variables and objects. Programmes are developed on a standard PC, compiled and downloaded directly via a PC/P-NET interface card, or via the RS232 interface. Program code can be downloaded to FLASH memory or battery backed RAM. Prior to delivery, a boot program is loaded into the EPROM.

The Operating System in a PD 5000 controller may be located in EPROM or FLASH, enabling the Operating System to be upgraded without changing the EPROM.

The P-NET channel structure is introduced in the PD 5000 controller, and the first 256 ($100) Softwire numbers are organised into standard channels.

The PD 5000 is programmed in Process-Pascal, version 3.1 or higher. The description of the facilities and the channels and variables in this manual apply to Process-Pascal version 4.0.

Memory
The PD 5000 has on-board memory consisting of 512 Kbytes of FLASH and 512 Kbytes of lithium battery backed RAM. In addition, a 128 Kbyte EPROM is provided, which holds the Operating System. The use of the RAM Extension Board (PD 5090), extends each of the FLASH and RAM memory capacities to 1 Mbyte.
The lithium battery can be exchanged while the controller is running.

Channels
In keeping with the standardised, object orientated channel structure of other P-NET devices, the PD 5000 incorporates a structure of 16 Channels, which deal with the configuration and use of Communication Ports, Program control, and the digital alarm output. As with all P-NET devices, the Service channel (channel 0) provides the ability, amongst other facilities, to identify the device and to provide the means to set the node address.

Communications Interfaces
The Controller has three serial communications interfaces, each configurable for PNETmode, DatamodeIn, DatamodeOut or DatamodeInOut. Two of these provide galvanically isolated RS485, multi-master, multi-net interfaces with the P-NET fieldbus, and run at the standardised speed of 76.8 Kbaud or at 9.6 Kbaud. In non-PNET modes, the communication channels are used to provide an interface to other communication protocols using RS485. In non-PNET modes, the Baud rates are adjustable between 1.2 and 76.8 Kbaud.

The third is a standard RS232 interface, which has an adjustable Baud rate of between 1200 and 9600. This communications channel is used to provide an interface to printers, modems, barcode readers etc., as well as other communications protocols using this medium.

Digital Channel
The PD 5000 has a built in digital channel, controlling the operation of the “Alarm Output” available at the rear of the controller.

2 Channel overview

The first 16 channels, channel number 0..F are designated as follows:

Channel number	Identifier	Channel type
0	Service	Service channel
1	Port1	Communication channel, port 1, RS485
2	Port2	Communication channel, port 2, RS485
3	Port3	Communication channel, port 3, RS232
4	ResCh4	Reserved
5	GatewayCh	Gateway channel, port 5
6	AlarmCh	Digital I/O channel, alarm output
7	ResCh7	Reserved for future expansions
8	OpSysCh	Program channel, operating system
9	PPProgCh	Program channel, Process-Pascal
A	KeyMouseCh	Keyboard/Mouse channel
B	DisplayCh	Display channel
C	ResChC	Reserved for future expansion
D	ResChD	Reserved for future expansion
E	ResChE	Reserved for future expansion
F	ResChF	Reserved for future expansion

Variables using Battery RAM memory type are left unchanged following a normal reset or loss of power, but are cleared after a master reset.

Variables using PROM Read only type memory, are declared as constants in the Process-Pascal program, and are thus stored along with this program, in EPROM, FLASH or RAM.

Register and port overview

The first softwire numbers following the standard channels are designated as follows:

Softwire no.	Identifier	Type
$100	InterfaceErrorBuffer	BUFFER[10] of InterfaceErrorRecord
$101	ControllerCode	ControllerCodeRec
$102	ActualPowerdownTime	LongInteger
$103	NodeList	Array[1..10] of NodelistElement
$104	DefaultPen	PenInformationType
$105	PDBoxDefinition	Array[0..0] of Word
$106	ExtTimeDate	TimeDateType
$107..$11F	Reserved	

The PD 5000 controller is equipped with 5 ports, having the following numbers and characteristics:

Port no.	Characteristic
1	Standard multimaster P-NET, RS485
2	Standard multimaster P-NET, RS485
3	Low speed P-NET, RS232
4	SPI, virtual port, used for on-board real-time clock (RTC) and EEPROM
5	Gateway, virtual port (refer to gateway channel)

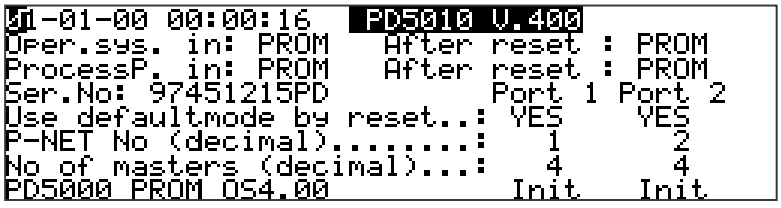
3 Getting started

3.1 Boot PROM program, Master reset

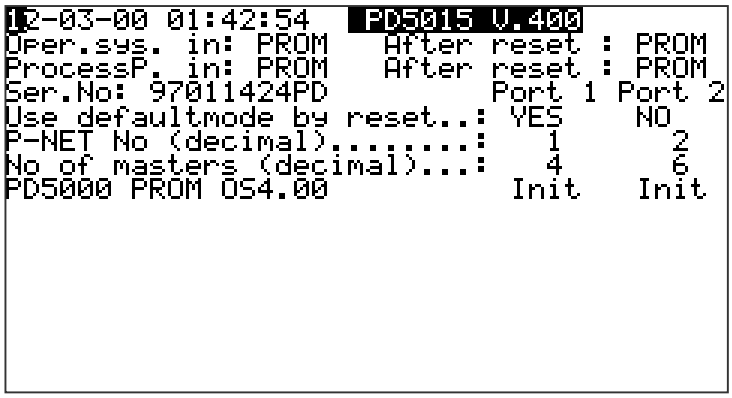
Following delivery, and after a Master reset, the controller will run the Process-Pascal program that is stored in the boot PROM memory.

Pressing and holding the reset button of the controller for about 10 seconds (or more), performs a Master reset.

When the controller is powered up after a Master reset, the display will show the following for a PD 5010:



When using a PD 5015, the display will show the following after a master reset:



When using a PD 5020, a similar display will be shown on the attached VGA screen.

On these screens, the P-NET node address for the controller, and the number of masters, can be keyed in for port 1 and 2. The program for this is placed in the boot PROM, and is thus the same in all controllers. However, the keyboard program in the application program may be different, perhaps with the arrow keys and so on placed elsewhere than that expected by the boot PROM program. Therefore, a customised the keyboard overlay MIGHT NOT be directly compatible with the boot PROM program.

After keying in the P-NET node address and the number of masters, a '1' should be entered in the 'Init' field. This will cause the controller to initialise the P-NET system for the selected port, according to the new values.

While the boot program is running, the controller is ready for download at any time.

If a program has already been downloaded to the flash memory, this program can be selected to run instead of the boot program. Enter a '1' in the 'PROM' area and perform a normal reset.

NOTE: It is recommended that the operating system is configured to run in FLASH memory. The performance of the controller is almost increased by a factor of two, when the operating system runs in FLASH memory.

When the controller has been reset, it will start the Process-Pascal program in the flash memory.

3.2 Downloading programs

The operating system may be downloaded to flash memory, using the OpSysCh channel.

A Process-Pascal program may be downloaded to flash memory or to the CMOS RAM memory, using the PPProgCh channel.

The Process-Pascal program is downloaded independently of the operating system. This means that it is possible to update the operating system to a new version, without changing the boot PROM. Therefore, the program space in flash memory must be sufficient for both the Process-Pascal program and the operating system. The operating system occupies approx. 63 K bytes of program memory.
Please refer to the VIGO Users Manual (Ref.No. 502086) for further information about how to use the Download tool under Windows 95/98 or NT.

3.3 CMOS RAM

The PD 5000 controller holds 512 K bytes of CMOS RAM with battery backup.

The operating system in the controller uses 8 K bytes of the RAM memory, hence 504 K bytes are free to be used by the user Process-Pascal program. The RAM memory is used for user data, variables, stack memory for tasks etc. for the Process-Pascal program, and may also be used for Process-Pascal code. However, normally the Process-Pascal code will be placed in flash memory.

After a Master reset the contents of the CMOS RAM are lost. The entire RAM is cleared to zeroes, but after that, the Process-Pascal program in the boot PROM will be running, using about 1.5 K bytes of RAM. The contents of this area of RAM are thus undefined following a Master reset.

3.4 Programming

The PD 5000 controller is programmed in Process-Pascal, which is based on standard PASCAL with some extensions, such as multi-tasking, built-in facilities for accessing external variables via P-NET, and standard procedures for writing on the display.

The Process-Pascal source code is edited by means of a standard editor. The source code is then compiled, by means of the Process-Pascal compiler. Depending on the selected options in the compiler setup, the compilation of the code results in the generation of a number of new files, of the form >xxx.LST<, >xxx.MAP<, >xxx.SMB<, >xxx.DEB<, >xxx.ERR< and >xxx.COD< files.

The >xxx.LST< file is a list file, holding the entire program and includes line numbers etc. The list file also contains any error messages, all of which are indicated with a ”^”. Finally, the list file holds information on compile time, as well as the data and code size for the program.
The >xxx.MAP< file contains a list of all the global variables and constants in the program. This includes the Softwire number, and the type and size of the variables and constants. The Softwire number of a variable or constant is defined as a logical address, which can, for instance, be used to access the variable or constant from other controllers via P-NET. The Process-Pascal compiler generates what is called a Softwire List, which is a table containing information about all the global variables and constants defined in the program. The Softwire List is part of the Process-Pascal code. The Softwire number acts as a pointer to an element in the Softwire List, defining one global simple or complex variable or constant.
The >xxx.SMB< file is a sub-MIB file, which can be amalgamated with a MIB file in VIGO. The SMB file contains all information about the variables, tasks, properties for backup, visibility etc.
The >xxx.DEB< file is a debug file, and contains additional information required by the Process-Pascal Debugger when debugging a program.
The >xxx.ERR< file is an error file, and contains a list of errors that may have occurred during compilation.
The >xxx.COD< file holds the Process-Pascal code. The file contents are in binary format. The Process-Pascal compiler does not compile the source code into machine code, but into an intermediate code (P-code), which is then interpreted by the interpreter/operating system in the controller. This dramatically reduces the size of the COD file.

The Process-Pascal language is described in a separate manual. This manual mainly contains information specific to the PD 5000 controller.

The Process-Pascal compiler suite is shipped with a number of additional programs and files, some of which contain basic variable, constant and procedure declarations. The following paragraphs contain a description of the result of the declarations made in the System file (PD5000.sys), which forms the Channel structure and other softwire numbers.

4 Service channel (channel 0)

Variables in Service channel (channel 0). Channel identifier: **Service**

SWNo	Identifier	Memory type	Read out	Type
0	NumberOfSWNo	PROM Read Only		Integer
1	DeviceID	PROM Read Only	- - - - - - - -	Record
2				
3	Reset	BatteryRAM	Hex	Byte
4	PnetSerialNo	Special function	- - - - - - - -	Record
5				
6	TimeDate	BatteryRAM	- - - - - - - -	Record
7	FreeRunTimer	RAM Read Only	Decimal	LongInteger
8				
9	ModuleConfig	BatteryRAM RPW	- - - - - - - -	Record
A				
B	Mailfilter	BatteryRAM	- - - - - - - -	String[9]
C	Mailbox	BatteryRAM	- - - - - - - -	Buffer
D	WriteEnable	BatteryRAM	Binary	Boolean
E	ChType	PROM Read Only	- - - - - - - -	Record
F	CommonError	BatteryRAM	- - - - - - - -	Record

SWNo 0: NumberOfSWNo
This variable holds the highest SWNo in the module. The value for this constant is calculated and patched by the compiler.

SWNo 1: DeviceID
The purpose of this record is to be able to identify the device. The record includes a registered manufacturer number, the type number of the module and a string, identifying the manufacturer.

```
Record
    DeviceNumber      : Word;       (* Offset = 0 *)
    ProgramVersion    : Word;       (* Offset = 2 *)
    ManufacturerNo    : Word;       (* Offset = 4 *)
    Manufacturer      : String[20]; (* Offset = 6 *)
end
```

The field values in the *DeviceID* record are shown below:

DeviceNumber	= 50xx	(xx = 00/10/15/20)
ProgramVersion	= 400	(version 4.00)
ManufacturerNo	= 1	
Manufacturer	= PROCES-DATA DK	

SWNo 3: Reset
By writing $FF to the *Reset* variable, the module performs a reset, and *ExternalReset* in *CommonError* SWNo $F is set TRUE.

SWNo 4: PnetSerialNo
This variable is a record having the following structure:

```
Record
        PnetNo          : Byte;         (* Offset = 0 *)
        NoOfMasters     : Byte;         (* Offset = 1 *)
        SerialNo        : String[20];   (* Offset = 2 *)
end
```

The serial number is used for service purposes and as a 'key' for setting the controller's P-NET Node Addresses and the number of masters. The Node Address is set for the port, through which this variable is accessed.

A special function is included for identifying a module connected to a network containing many other modules, having the same or unknown node addresses, and to enable a change of the node address and number of masters via the P-NET.

Setting a new node address and number of masters via the P-NET is performed by writing the required node address and number of masters together with the serial number of the module in question, into the *PnetSerialNo* at node address $7E (calling all modules). All modules on the P-NET will receive the message, but only the module with the transmitted serial number will store the P-NET node address and the number of masters. If the transmitted number of masters is = 0, the value is NOT stored in *NoOfMasters*.
An attempt to write data to node address $7E will give no reply. Consequently the calling master must disable the generation of a transmission error when addressing this node.

In the module, the *SerialNo* = "XXXXXXXXPD", is set by PROCES-DATA, and cannot be changed. The eight X`s indicate the serial number, and PD is the initials of PROCES-DATA.

SWNo 6 TimeDate
This variable holds the data from the real-time clock, in the form of time and date. The variable has the following structure:

```
Record
        Second          : Byte; (0 - 59)
        Minute          : Byte; (0 - 59)
        Hour            : Byte; (0 - 23)
        Day             : Byte; (1 - 7 Sunday = 1 )
        Date            : Byte; (1 - 28/29/30/31)
        Month           : Byte; (1 - 12, January = 1)
        Year            : Byte; (0 -99 (modulus 100 of the year)
        Code            : Byte;
End
```

The operating system in the Controller synchronises the *DateTime* with the real-time clock chip every time the Second changes to 0. The user should avoid setting the time to 'just before midnight', i.e. 23:59:SS, since in this case, the time will not reset to 00:00:00, but will revert to the previous time setting. If the time is set to 23:58:SS, the time will reset to 00:00:00 correctly.

The *Code* byte is not used.

SWNo 7: FreeRunTimer
FreeRunTimer is a timer, to which internal events are synchronised. The timer is of type LongInteger with a 1/256 Second resolution.

SWNo 9: ModuleConfig

```
Record
        Enablebit       : Bit8;         (* Offset = 0 *)
        Functions       : Byte;         (* Offset = 1 *)
        Ref_A           : Byte;         (* Offset = 2 *)
        Ref_B           : Byte;         (* Offset = 3 *)
end
```

No functions are related to *ModuleConfig* in this module.

SWNo B: MailFilter
The *MailFilter* is used in connection with the *MailBox* (see below).

MailFilter : String*[9];*

The value of the *MailFilter* can be any combination of characters in a string with a length of up to 9 characters.

There are no automatic functions implemented in related to the use of *MailFilter.*

The configuration for the filter functions, the meaning of the messages, the action on all or specific messages, is totally dependent on the application program, and is not a part of the P-NET standard.

SWNo C: MailBox
The *MailBox* variable is a general-purpose message system for the P-NET. Mail in the system could be an alarm, a message for an operator or an event to be logged on a printer. In a situation where it is required that the application program performs some special activity when the *MailBox* is accessed, the programmer may arrange to generate an interrupt under these circumstances, in order to start a dedicated task.

The basic message system consists of a mail buffer - the *MailBox*, and a mail filter - the *MailFilter* (see above).

MailBox : Buffer[10] Of String*[89];*

SWNo $D: WriteEnable
WriteEnable does not apply to any variables in the PD 5000 Controller.

SWNo E: ChType
Each channel within an interface module is described in an individual *ChType* variable. This is a Record, consisting of a unique number for the channel type and a TRUE boolean value for each of the registers which are represented within a channel. The register number in a channel corresponds to the index number in the boolean array. In addition to these fields, various other fields may be found within the record, which are dependent on the channel type.

```
Record
        ChannelType     : Word;         (* Offset = 0 *)
        Exist           : Bit16;        (* Offset = 2 *)
        Functions       : Bit16;        (* Offset = 4 *)
end
```

For the service channel, *ChType* has the following value:

ChannelType = 1

Exist =

15								7							0
1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	1

Functions =

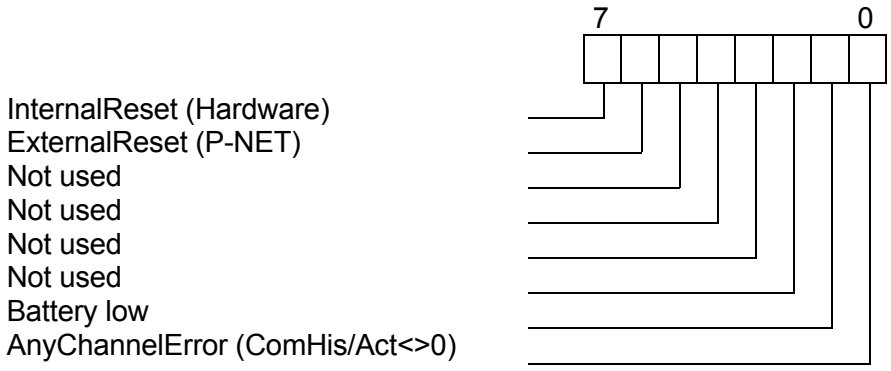
15								7							0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

SWNo F: CommonError
The *CommonError* variable holds error status information for all Channels.

This variable is a record of the following type:

```
Record
        ChError:
                Record
                        His:Array[0..7] of Boolean;     (* Offset = 0 *)
                        Act:Array[0..7] of Boolean;     (* Offset = 2 *)
                End;
        ComHis16:Array [0..15] of Boolean;              (* Offset = 4 *)
        ComAct16:Array [0..15] of Boolean;              (* Offset = 6 *)
End
```

The 8 bits in *ChError.His* and *ChError.Act* have the following meaning:



Bit 7 InternalReset is set TRUE if a reset is caused by a power failure, reset button, by writing $FF to *Service.Reset* from Process-Pascal, or by any other internal error causing a reset. Clearing the *Service.Reset* (softwire 3) clears the bit in .Act.

Bit 6 ExternalReset is set TRUE if a reset is caused by writing $FF to SWNo 3 - Reset, via P-NET. Clearing the *Service.Reset* (softwire 3) clears the bit in .Act.

Bit 1 Battery low is set if the Lithium battery for RAM backup and RealTimeClock is low. When the battery is low, RAM contents are lost if power is turned off (the controller will perform a masterreset). Exchange the battery as soon as possible.

Bit 0 AnyChannelError = 1 means that an error exists in one or more channels.

In the PD 5000 controller, a range of different causes can produce a reset. The reason for the last reset can be read in the *ResetCode*. *ResetCode* is declared in the system file and can also be accessed via P-NET. The reset code is updated each time the controller performs a reset, and is NOT cleared by readout. See further details in chapter 12.

5 Communication channel (Channel 1 - 4)

The PD 5000 controller provides three communication ports, and each of them has a related channel for the communication parameters and exchange of data. Channel 4 is also defined as a communication channel, although no external physical communication port is provided. Port 4 is used to access the real-time clock from the operating system. However, this port must be declared as a communication port, because it must be initialised by the operating system. The real-time clock can be accessed via the Service channel.

Variables in a Communication channel — Channel identifier: **Port_x**

SWNo	Identifier	Memory type	Read out	Type
x0	ActualMode	BatteryRAM	- - - - - - - -	Record
x1	OutputBuffer	RAMReadWrite	- - - - - - - -	Buffer
x2	InputBuffer	RAMReadWrite	- - - - - - - -	Buffer
x3				
x4				
x5				
x6				
x7				
x8				
x9	ChConfig	BatteryRAM	- - - - - - - -	Record
xA				
xB				
xC	DefaultMode	PROM ReadOnly	- - - - - - - -	Record
xD				
xE	ChType	PROM ReadOnly	- - - - - - - -	Record
xF	CHError	RAM ReadOnly	Binary	Record

SWNo $x0: ActualMode
This variable is a record of the following type:

```
Record
   PNETNo       : Byte;
   NoOFMasters  : Byte;
   Protocol     : ProtocolType;
   ElStd        : ElStdType;
   BaudRate     : LongInteger;
   SumCheck     : Boolean; (* False: 1 byte, True: 2 byte*)
   Parity       : ParityType;
   Manual       : Boolean; (* False: Auto, True:Manual *)
   DisableNIL   : Boolean; (* False: NIL inserted, True:NIL not inserted *)
   DataOut      : Boolean; (* False: String, True:Block *)
   DataIn       : Boolean; (* False: String, True:Block *)
   StopChar     : Byte;
end
```

PNETNo defines the P-NET number of the port. All units connected to P-NET must have a unique P-NET number, a P-NET node address. *Port_x.PNETNo* indicates the node address for P-NET on Port_x and must be in the range from 1 to 125.

NoOfMasters defines the number of masters for the port. Since communication ports can act within a multi-master net, the operating system must know the number of masters on each net. A master is a unit which can instigate access to P-NET, i.e. it can start a transmission to another unit and afterwards expects an answer to that transmission, unlike a slave unit, which can only return answers to a master. Only controllers can be masters, and only if the controller has the *Port_x.PNETNo* set to a value that is less than or equal to *Port_x.NoOfMasters*. If *Port_x.PNETNo* is greater than *Port_x.NoOfMasters*, the controller can only act as a slave module, in which case it must not initiate any transmissions. *Port_x.NoOfMasters* indicates the number of masters on the net and must be in the range from 1 to 32.

The values of these parameters may also be changed via the Service channel, by accessing the variable called *Service.PnetSerialNo*. When this method is used, the values are always transferred to the EEPROM, regardless of the state of *Service.WriteEnable*.

Protocol is declared as an enumeration type and defines the protocol mode for the Port. *Protocol* can take the following values:

Disabled	(the communication port is disabled)
PNETMode	(P-NET standard mode)
DatamodeIn	(data-mode input, reads bytes or ASCII characters)
DatamodeOut	(data-mode output, sends bytes or ASCII characters)
DatamodeInOut	(data-mode input+output (full duplex on RS232))

In datamode, it is only the actual data that is received/transmitted.

ElStd can hold the following values: RS485, IS16, RS232
If the electrical standard of the port is RS232, the normal rules for RS232 handshake signals are observed. That is, CTS is activated whenever the port is ready to receive data in *DatamodeIn*, and deactivated if it is not ready.

BaudRate selects the baud rate for the communication port. If an illegal baud rate is selected, i.e. the baud rate is not implemented, an error code is generated. *Baudrate* can hold the following values:

For port 1 and 2, in P-NET mode:
76800, 9600
For port 1 and 2, in Datamode:
76800, 38400, 19200, 9600, 4800, 2400, 1200

For port 3, in P-NET mode or Datamode:
9600, 4800, 2400, 1200

SumCheck defines whether the P-NET frames use a one byte sum check (Default error detection) or a two byte sum check (Extended error detection). A one byte sum check is selected when *SumCheck* is set to False.

Parity selects the parity for the data on the port. Parity can hold the following values: *NoParity*, *AddressData*.

Manual is used in conjunction with the P-NET CHIP, and defines whether handshake signals for RS232 in DataMode are controlled automatically or manually (from the Process-Pascal program). Please refer to the P-NET CHIP manual for further information.

DisableNIL is used to define whether NIL should be inserted in callback frames as the last source address, according to the P-NET Standard. The default value for *DisableNIL* is False, which means that a NIL is inserted. If *DisableNIL* is True, then a NIL is NOT inserted. This facility is necessary to maintain compatibility with older versions of P-NET devices, eg. PD 3000.

DataOut is used to define whether a port that is set to Protocol datamode out, is in string or block mode. In string mode, the number of bytes defined in the first byte (the current length of the string) is sent. The length itself is not transmitted. In block mode, the entire buffer element from *OutputBuffer* is sent. String mode is selected when *DataOut* is False.

DataIn is used to define whether a port that is set to protocol datamode in, is in string or block mode. In string mode an element is transferred to the *InputBuffer* when the character defined in ***StopChar*** is received, or until the number of bytes in an element minus 1 is received. The length of the string is inserted in the first byte of the buffer element. In block mode, an element is transferred to the *InputBuffer* when the entire number of bytes in an element is received.

The Process-Pascal standard procedure *InitPort(x)* will initialise the port according to the port configuration given in *ActualMode*. The ports are also initialised following a reset of the device. The port configuration following a reset is either the values from *ActualMode* or from *DefaultMode*, depending on the channel configuration (*ChConfig.EnableBit[0]*). Furthermore, a port can be initialised as a result of a write operation to *Service.PnetSerialNo* with a *SerialNo* that matches the device. In this case, the port will be initialised according to the received *PNETNo* and *NoOfMasters*.

SWNo $x0: OutputBuffer
If the port is in DatamodeOut or DatamodeInOut, data in this buffer will be sent directly to the port.

If the port is set to STRING mode (*ActualMode.DataOut* is False), then the bufferelements must be of type String. Data is transmitted when there is at least one element in the buffer. Only the number of bytes corresponding to the actual length of the string is sent. The string length (1st byte) is NOT sent.

When the port is set to STRING mode, the buffer takes the following form:

Buffer[10] of String*[255]*

If the port is set to BLOCK mode (*ActualMode.DataOut* is to True), then the data is transmitted when there is at least one element in the buffer. All the bytes contained in one element are sent.
The *OutputBuffer* is initialised in the task called InitTask.

SWNo $x1: InputBuffer
If the port is in DatamodeIn or DatamodeInOut, data received at the port is transferred to this buffer.

If the port is set to STRING mode (*ActualMode.DataIn* is False), then the bufferelements must be of type String. Data is transferred to the buffer when the maximum number of bytes that the element can hold has been received, or when the stopcharacter (*ActualMode.StopChar*) is received. The stopcharacter may, for example, be $0D, the ASCII code CR. The string length (1st byte) is calculated and inserted - NOT received.

When the port is set to STRING mode, the buffer takes the following form:

Buffer[10] of String*[255]*

If the port is set to BLOCK mode (*ActualMode.DataIn* is True), then the data is transferred to the buffer, when the maximum number of bytes that one element can hold, is transferred.

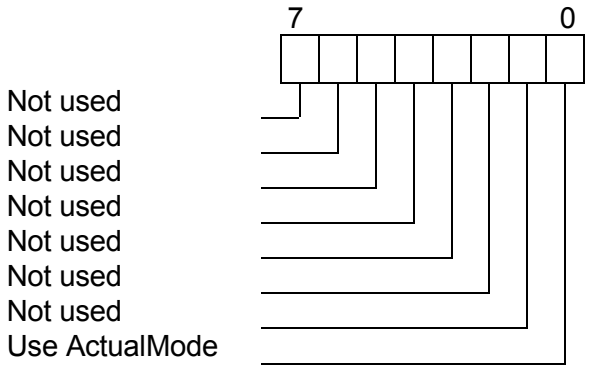
When an element is transferred to the *InputBuffer*, a Softwire interrupt can be generated. However, this facility is NOT part of the Communication channel standard.

If the electrical standard of the port is RS232, the normal rules for RS232 handshake signals are observed. That is, CTS is activated whenever the port is ready to receive data in DatamodeIn, and deactivated if it is not ready (not configured for DatamodeIn or *InputBuffer* full).
The *InputBuffer* is initialised in the task called InitTask.

SWNo $x9: ChConfig
The run-time configuration for the port is defined in *ActualMode*. Only the port configuration following a reset is defined in this variable.

```
Record
        Enablebit       : Bit8;         (* Offset = 0 *)
        Functions       : Byte;         (* Offset = 1 *)
        Ref_A           : Byte;         (* Offset = 2 *)
        Ref_B           : Byte;         (* Offset = 3 *)
end
```

Enablebit:



Only *EnableBit[0]* is used , having the following meaning:

False: Copy DefaultMode to ActualMode after Reset
True: Don't copy DefaultMode, use ActualMode after Reset

Functions, *Ref_A* and *Ref_B* are not used.

SWNo $xC: DefaultMode

DefaultMode is a record of the same type as *ActualMode*. If *ChConfig.EnableBit[0]* is False, then *DefaultMode* is copied to *ActualMode* on reset, before the port is initialised.

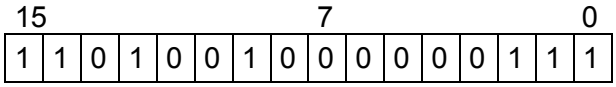
SWNo $xE: ChType

```
Record
        ChannelType: Word;          (* Offset = 0 *)
        Exist: Bit16;               (* Offset = 2 *)
        Functions: Bit16;           (* Offset = 4 *)
end
```

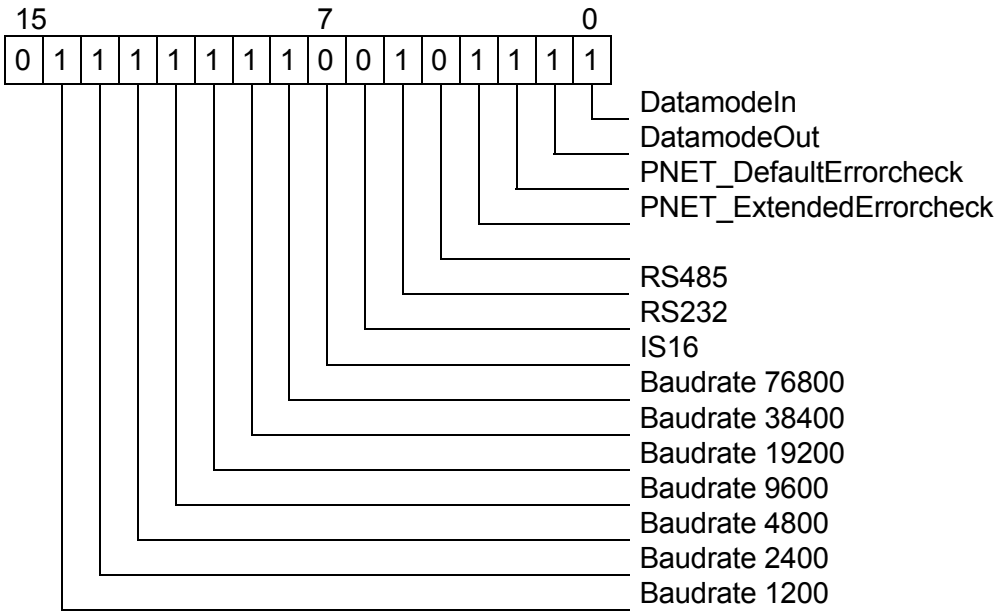
For the communication channels, *ChType* has the following value:

ChannelType = $8011

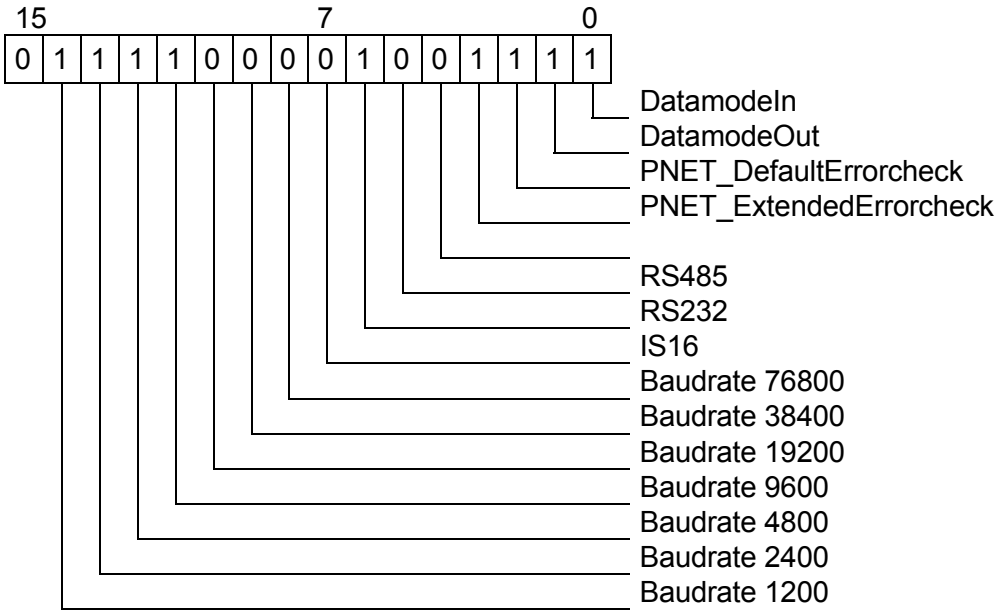
Exist =



Functions = (for ports 1 and 2)



Functions = (for port 3)



SWNo xF: ChError

```
Record
        His:Array[0..7] of Boolean;     (* Offset = 0 *)
        Act:Array[0..7] of Boolean;     (* Offset = 2 *)
End
```

No errors can be reported in the Communication channels.

6 Gateway channel (channel 5)

Variables in Gateway channel. Channel identifier: **Gateway**

SWNo	Identifier	Memory type	Read out	Type
50	GatewayRecord	BatteryRAM	- - - - - - - -	Record
51	GatewayInterrupt	PROM Read only	Decimal	Byte
52				
53				
54				
55				
56				
57				
58				
59	ChConfig	BatteryRAM	- - - - - - - -	Record
5A				
5B				
5C				
5D				
5E	ChType	PROM Read Only	- - - - - - - -	Record
5F	CHError	RAM Read Only	Binary	Record

The Gateway channel operates on a virtual communication port, port 5. The function of the Gateway channel is to interconnect P-NET and other communication protocols, or to activate, for example, time consuming procedures within other controllers.

When a request is made from P-NET (or from the Process-Pascal program in this controller), to Port 5, the complete P-NET request is transferred to the *GatewayRecord*, and an interrupt can be generated. The interrupt invokes a Process-Pascal program, which performs all communication (or time consuming calculations) through one of the physical ports 1 to 3, and returns the response to the *GatewayRecord*. When the response is returned, the operating system takes over, and returns the answer as if it was a normal P-NET transmission.

When the request is made via P-NET to the gateway port, the operating system returns an AnswerComesLater response onto the P-NET. Normally, the calling master waits approx. 2 seconds for the answer.

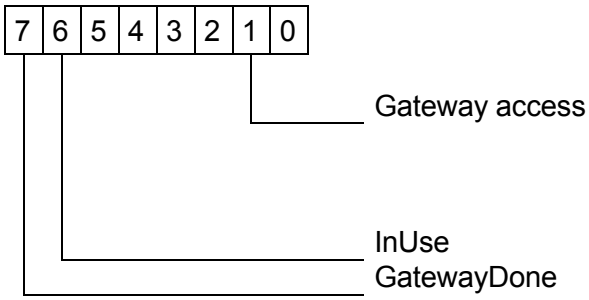
Refer to chapter PD GATEWAY in the Process-Pascal manual, version 4, for further details about the Gateway facility.

SWNo 50: GatewayRecord

```
Record
        NodeAddress     : String[25]
        Control_Status  : Byte;
        InfoLength      : Byte;
        Info            : Array[1..63] of Byte;
        Flags           : Array[0..7] of Boolean;
end
```

The fields *NodeAddress*, *Control_Status*, *InfoLength* and Info, correspond to the same fields that are described in the P-NET standard. The operating system handles all NodeAddress conversion. The task of the Process-Pascal program is to return Control/Status, InfoLength and Info according to the results of the NON P-NET transmission (or time consuming calculation). After this, the program must set the *GatewayDone* bit in the Flags field to True (*Flags[7]:= True*), to activate the operating system, which then returns the answer to the P-NET master.

The Flags variable has the following meaning:



Before an interrupt is generated to invoke the Process-Pascal program, *InUse* is set TRUE by the operating system. This is to prevent others from using the Gateway channel until this operation is Done.

When the Process-Pascal program has returned the response to *GatewayRecord,* it must set *GatewayDone* to True. The Process-Pascal program should NEVER write to *InUse*. The operating system routines that are activated by the *GatewayDone* bit will clear *InUse*, when the operation is completed.

SWNo 51: GatewayInterrupt
This variable denotes the number of the interrupt, which will invoke the Process-Pascal program. The number must be declared as a constant in the Process-Pascal program.

SWNo 59: ChConfig

```
Record
        Enablebit       : Bit8;         (* Offset = 0 *)
        Functions       : Byte;         (* Offset = 1 *)
        Ref_A           : Byte;         (* Offset = 2 *)
        Ref_B           : Byte;         (* Offset = 3 *)
end
```

The fields in this record are available for the Process-Pascal program that controls the Gateway communication.

SWNo $5E: ChType

```
Record
        ChannelType: Word;          (* Offset = 0 *)
        Exist: Bit16;               (* Offset = 2 *)
end
```

For the Gateway channel, *ChType* has the following value:

ChannelType = $8007

Exist =

15								7							0
1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	1

SWNo 5F: CHError

```
Record
        His:Array[0..7] of Boolean;     (* Offset = 0 *)
        Act:Array[0..7] of Boolean;     (* Offset = 2 *)
End
```

No errors can be reported in the Communication channels.

7 Alarm output (channel 6)

This channel controls the alarm output of the controller. The channel is constructed as a standard digital I/O channel, but the corresponding functionality is not implemented. Below is described how to activate the alarm output and how to set the alarm to be a pulse output.

Variables in digital I/O channel 6. Channel identifier: **AlarmOutput**

SWNo	Identifier	Memory type	Read out	Type	SI Unit
60	FlagReg	BatteryRAM	Binary	Bit8	- - -
61	OutTimer	BatteryRAM	Decimal	Real	s
62	Counter	BatteryRAM	Decimal	Longinteger	- - -
63					
64					
65					
66					
67					
68					
69	ChConfig	BatteryRAM	- - - - - -	Record	- - -
6A					
6B					
6C	UserLIArray	BatteryRAM	- - - - - -	Array2LI	- - -
6D	Maintenance	BatteryRAM	- - - - - -	Record	- - - - -
6E	ChType	PROM Read Only	- - - - - -	Record	-
6F	ChError	RAM Read Only	Binary	Record	- - -

SWNo 60: FlagReg
No functions are implemented on the *FlagReg* variable. The state of the alarm output can be read in *FlagReg[7]*.
Calling the standard procedure *AlarmHornOnOff* with a boolean parameter will set the alarm output according to the boolean value.

SWNo 61: OutTimer [s]
The *OutTimer* is not used in the Alarm channel.

SWNo 62: Counter
The *Counter* is used as a timer register by the operating system. The format of this register corresponds to the FreRunTimer in the Service channel. The alarm output starts pulsing if *UserLiArray[1]* is a value other than zero. *UserLiArray[1]* indicates the OFF time and *UserLiArray[0]* indicates the ON time.

SWNo 69: ChConfig
No functions are related to the *ChConfig* variable.

SWNo 6C: UserLIArray
The alarm output can act as a simple output that can be set ON or OFF. It can also act as a pulse output.

The pulse output function is selected by setting *UserLIArray[1]* to a value other than zero. The output will then be set true for a period equal to the value of *UserLIArray[0]*. When the *Counter* reaches zero, the output will be set false for a period equal to the value of *UserLIArray[1]*, and so on. The output is reset if the *UserLIArray[1]* is set to zero.

The ON and OFF periods for the pulse output can be set by using the standard procedure *AlarmPulseOn(OnTime, OffTime)*

SWNo 6D: Maintenance
The *Maintenance* variable is used for service management and maintenance purposes, and holds the last date of service and an indication of the type of service.

```
Record
        Date            : Byte;         (* Offset = 0 *)
        Month           : Byte;         (* Offset = 1 *)
        Year            : Byte;         (* Offset = 2 *)
        Category        : Byte;         (* Offset = 3 *)
end
```

SWNo 6E: ChType

```
Record
        ChannelType: Word;              (* Offset = 0 *)
        Exist: Bit16;                   (* Offset = 2 *)
        Functions: Bit16;               (* Offset = 4 *)
        FeedBack: Bit16;                (* Offset = 6 *)
end
```

ChType has the following value:

ChannelType = 2
Exist =

15								7							0
1	1	1	1	0	0	1	0	0	0	0	0	0	1	1	1

Functions and *Feedback* are not used for the alarm output channel.

SWNo 6F: CHError
No errors can be reported in this channel.

8 OpSystem program channel (channel 8)

The OpSystem program channel provides the ability to download and run the operating system for the PD 5000 controller without changing the EPROM. The operating system can be downloaded to FLASH, but can be run in EPROM or FLASH. Selection of the EPROM or FLASH operating system, is performed from a library.

A number, *LibraryIndex*, which is found in *LibraryControl*, provides and index to the programs in the library. The total number of programs, which can be stored in the library-memory, is stated in *MaxLibraryIndex*, which is found in *LibraryStatus*.

Registers in Program Channel — Channel identifier: **OpSysCh**

SWNo.	Identifier	Memory Type	Read Out	Type
80	ProgramControl	BatteryRAM	- - - - - - -	Record
81	ProgramStatus	RAM Read Only	- - - - - - -	Record
82	ProgramID	Read Only	- - - - - - -	Record
83				
84				
85	SystemPointer	PROM Read Only	Hex	LongInteger
86				
87	MemoryInfo	Read Only	- - - - - - -	Record
88	IDAndCode	Special function	- - - - - - -	Record
89	ChConfig	BatteryRAM	- - - - - - -	Record
8A	LibraryControl	BatteryRAM	- - - - - - -	Record
8B	LibraryStatus	RAM Read Only	- - - - - - -	Record
8C	LibraryProgramID	Read Only	- - - - - - -	Record
8D	Maintenance	BatteryRAM	- - - - - - -	Record
8E	ChType	PROM Read Only	- - - - - - -	Record
8F	ChError	RAM Read Only	- - - - - - -	Record

SWNo. 80: ProgramControl
ProgramControl is used to set and change the state of the current program, which has been selected and invoked via the Program Channel. The selected program number is inserted and indicated as a part of *ProgramControl*. Commands can be sent to *ProgramControl* to stop, start or reset the program.

```
Record
        Command         : Byte;
        ProgramToSelect : Word;
        ErrorStatus     : Bit32;
End
```

Command is used to send a command to the Program Channel to change the state of the current program. A list of possible commands is given below. The commands and the corresponding numbers conform to the Request Instructions used by MMS.

	Command	**Purpose**
38	SelectProgram	Selects a program from the library, resulting in State = Idle. (MMS = CreateProgramInvocation)
39	UnSelectProgram	Sets SelectedProgram to 0, resulting in State changing to Non-selected. (MMS =DeleteProgramInvication)
40	Start	Starts the selected program (MMS=Start) if the program is OK.
41	Stop	Stops the selected program. (MMS=Stop)
42	Resume	Continues program execution in the selected program. (MMS=Resume)
43	Reset	Resets the selected program. (MMS=Reset)
44	Kill	Stops the selected program instantly and sets the state to unrunable. (MMS=Kill)

The Program Channel automatically sets *Command* to 0 after writing to the variable. *State* is updated immediately to one of the corresponding temporary states, *Starting*, *Stopping*, *Resuming* or *Resetting* each time a command is sent to the *Command* variable. By the change in state, it is possible to read the variable *ProgramStatus.State* to check whether the *Command* was executed successfully or that the operation failed.

ProgramToSelect holds the library index for the program to select or the already selected program. *ProgramToSelect* is copied from *ChConfig* following a module reset or power up. It can hold the values 1 for EPROM or 2 for FLASH.

ErrorStatus indicates each type of error in a program by means of a Boolean value. An error may be cleared in this register by writing FALSE to the corresponding error bit. The *ErrorStatus* indicates run-time errors in the running operating system.

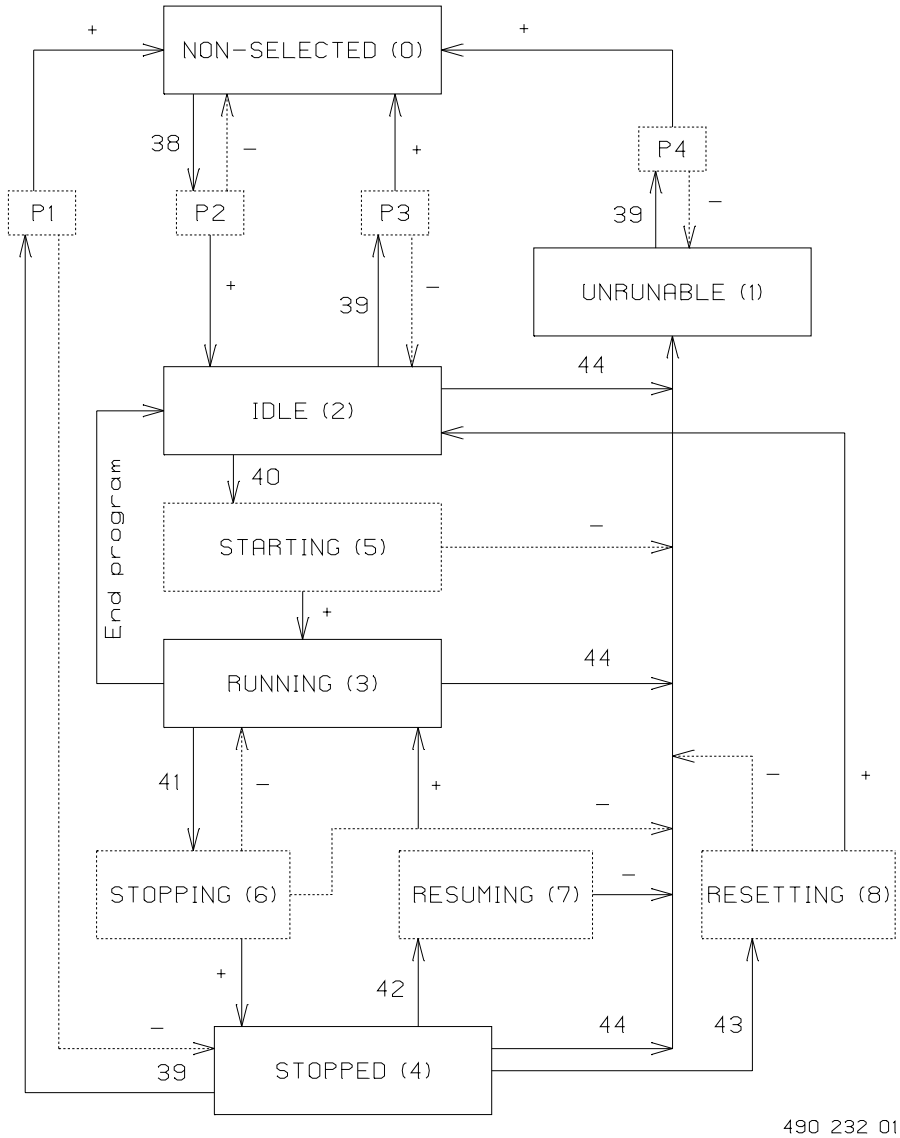
SWNo. 81: ProgramStatus

ProgramStatus summarises the state and error condition of the selected program. The library index for the selected program is also indicated.
ProgramStatus is a record of the following type:

```
Record
        State           : Byte;
        SelectedProgram : Word;
        ErrorStatus     : Bit32;
End
```

State indicates the current state ofr the selected program, eg. *stopping*, *stopped*, *running*, *idle*, *non-selected* etc. A state diagram and a list of possible states are given below. The states and the corresponding numbers conform to the states for Program Invocation Management used by MMS.

PROGRAM INVOCATION STATE DIAGRAM



\+ State transition succeeded - State transition failed.

State	Explanation

0	Non-selected	No program selected. (MMS = Non-selected)
1	Unrunable	The program can not run. (MMS = Unrunable)
2	Idle	The program is stopped and reset. (MMS = Idle)
3	Running	The program is running. (MMS = Running)
4	Stopped	The program is stopped. (MMS = Stopped)
5	Starting	The program is changing state from idle to running. (MMS = Starting)
6	Stopping	The program is changing state from running to stopping. (MMS = Stopping)
7	Resuming	The program is changing state from stopped to running. (MMS = Resuming)
8	Resetting	The program is changing state from stopped to idle. (MMS = Resetting)

SelectedProgram holds the library index for the selected program. *SelectedProgram* is 0 if *State* is *Non-selected*.
ErrorStatus is identical to *ProgramControl.ErrorStatus*, but only read access is possible.

SWNo. 82: ProgramID
ProgramID is used to identify the selected program. The record includes a name for the program, version number, the required version number for the interpreter program and a name for the Softwarehouse, which created the program. Compile time, compiler version and actual size for the program is also a part of *ProgramID* as well as SumCheck (2's complement word addition without carry) and a code type identifier. The *SumCheck* value may be used for check sum calculations when the program is selected, i.e. following a *SelectProgram* command (38). *CodeType* must match *CodeType* found in *ChType*.

```
Record
        ProgramName     : String[20];
        Version         : Word;
        InterpreterVers : Word;
        SoftwareHouse   : String[20];
        CompileTime     : DateTimeRec;
        CompilerVersion : Word;
        ActualSize      : LongInteger;
        SumCheck        : Word;
        CodeType        : Word;
        NoOfTask        : Word;
        RAMNeed         : LongInteger;
        Reserved1       : LongInteger;
        Reserved2       : LongInteger;
End
```

The values and interpretation for *CodeType* are specified by the International P-NET User Organization. This description is available in a separate document, no. 590 003 and may be issued on request.

SWNo. 87: MemoryInfo
For each program segment, selected by *LibraryControl.LibraryIndex*, a corresponding memory information can be read. The memory information holds the actual size for the program, the max. size for the program segment and a code for the memory type in which the program is stored.

```
Record
        ActualSize      : LongInteger;
        MaxSize         : LongInteger;
        MemoryType      : Word;
End
```

ActualSize includes the size of the program code and the header with the *ProgramID*.

MaxSize indicates the max size for the program segment and is the max size for a program to download within the memory area specified by ***MemoryType***.

The values and interpretation for *MemoryType* are specified by the International P-NET User Organization. This description is available in a separate document, no. 590 003 and may be issued on request.

SWNo. 88: IDAndCode
This SoftWire number is used for up- or download of programmes from/to the Program Channel. When a program is downloaded to the Program Channel, the entire program and a header with the *ProgramID* and size indicator, is stored in *IDAndCode* by means of a LongStore instruction.

The program and the header with the *ProgramID*, a size indicator and a code-type indicator is a variable of the following type:

```
IDAndCode = Array[1..ActualSize] Of Byte;
```

The format for the data stored in the first part of *IDAndCode* matches exactly the format for *ProgramID*, and *IDAndCode* is interpreted as a record of the following type:

```
Record
        ProgramName     : String[20];
        Version         : Word;
        InterpreterVers : Word;
        SoftwareHouse   : String[20];
        CompileTime     : DateTimeRec;
        CompilerVersion : Word;
        ActualSize      : LongInteger;
        SumCheck        : Word;
        CodeType        : Word;
        NoOfTask        : Word;
        RAMNeed         : LongInteger;
        Reserved1       : LongInteger;
        Reserved2       : LongInteger;
        ProgramCode     : Array[1..(ActualSize-HeaderSize)] Of Byte;
End
```

Before the program can be downloaded to the channel, the download program must ensure that the necessary memory area is available, the code-type for the program code is in accordance with the code-type specified for the channel and that the interpreter program in the operating system is of the right version.

To download or upload a program, the corresponding command must be sent to *LibraryControl.Command* register to initiate the sequence. The download program must wait for *LibraryStatus.State = Loading* before the download is executed.

ActualSize for a program is given in *LibraryProgramID.ActualSize* after a complete download.

SWNo. 89: ChConfig

The specification for how the selected program must operate after a reset or power failure is held in *ChConfig*. This configuration includes a number specifying which program must be invoked after reset.

```
Record
        Enablebit       : Array[0..7] Of Boolean;
        Functions       : Byte;
        Ref_A           : Byte;
        Ref_B           : Byte;
End
```

EnableBit is not used.

Functions is not used.

Ref_A holds the selected program number which must be invoked after module reset or power up. If *Ref_A = 0* it is automatically changed to 1, selecting EPROM.

Ref_B is not used.

SWNo. 8A: LibraryControl

LibraryControl is used to set and change state for a program in the library. The program in the library is chosen with *LibraryIndex*. Commands can be sent to *LibraryControl* to control a download or upload sequence.

LibraryControl is a record of the following type:

```
Record
        Command         : Byte;
        LibraryIndex    : Word;
End
```

Command is used to send a command to the Program Channel for changing the state of the program chosen by *LibraryIndex*. A list of possible commands is given below. The commands and the corresponding numbers conform to the Request Instructions for download used by (MMS).

	Command	**Purpose**
26	InitiateDownloadSequence	Prepare for download. (MMS = InitiateDownloadSequence)
28	TerminateDownloadSequence	Cancel download and end sequence. (MMS = TerminateDownloadSequence)
29	InitiateUploadSequence	Prepare for upload (not implemented) (MMS = InitiateUploadSequence)
31	TerminateUploadSequence	Cancel upload and end sequence. (MMS = TerminateUploadSequence)
36	DeleteProgram	Delete program from the library. (MMS = DeleteProgram)

The Program Channel automatically sets *Command* to 0 after writing to the variable. *State* is immediately updated to one of the corresponding temporary states, *Complete* or *Incomplete* each time a command is sent to the *Command* variable. By the change in state it is possible to read the variable *LibraryStatus.State* to check if the *Command* was executed successfully or the operation failed.

LibraryIndex chooses one of the programmes in the program library. When a program is chosen, all data concerning this program may be accessed. The data for the chosen program may be read at the variables *LibraryProgramID*, *LibraryStatus* and *MemoryInfo*. Upload and download of the complete program, including the program code, is done via the *IDAndCode* variable.

If *LibraryIndex* is equal to *ProgramStatus.SelectedProgram*, download is not possible.

The value of *LibraryIndex* must be 1 or 2.

SWNo. 8B: LibraryStatus

LibraryStatus is used to read the current state for a program in the library. *LibraryIndex* indicates the chosen program in the library. *MaxLibraryIndex* states the max number of programmes in the library.
LibraryStatus is a record of the following type:

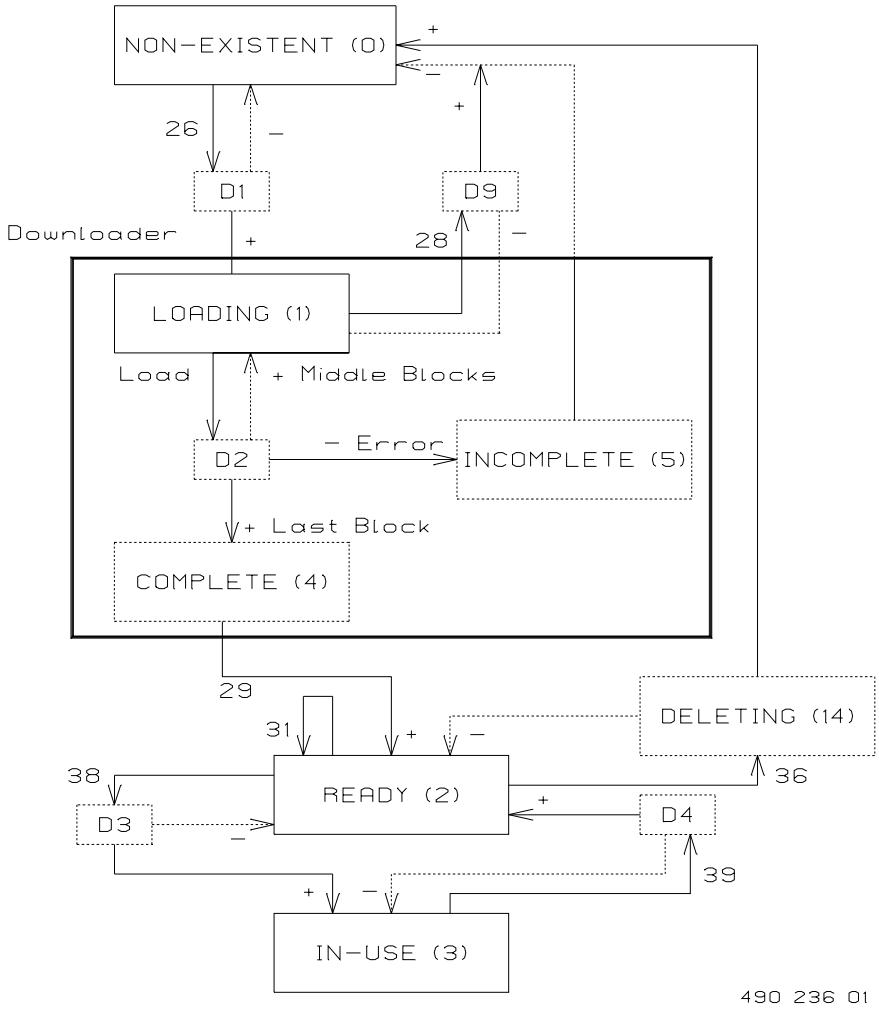
```
Record
        State           : Byte;
        LibraryIndex    : Word;
        MaxLibraryIndex : Word;
End
```

State indicates the current state for the program, e.g. *loading*, *ready*, *nonexistent* etc. A list of possible states is given below. The states and the corresponding numbers conform exactly to the states for download domain used by The Manufacturing Message Specification.

State		Explanation
0	Non-existent	No program in this memory type or program segment. (MMS = Non-existent)
1	Loading	Download in progress.(MMS = Loading)
2	Ready	The program is ready to be selected to run. (MMS = Ready)
3	In-use	This program is selected in ProgramControl. The state change to/from In-use is entirely controlled in ProgramControl. Download is not allowed. (MMS = In-use)
4	Complete	The program is completely downloaded and will change state to ready. (MMS = Complete)
5	Incomplete	An error has occurred during download and the program changes state to Non-existent. (MMS = Incomplete)
14	Deleting	Deletion in progress, e.g. clearing Flash. (MMS = Deleting)

LibraryIndex is identical to *LibraryControl.LibraryIndex*.

PROGRAM STATE DIAGRAM



\+ State transition succeeded - State transition failed.

MaxLibraryIndex indicates the max. number of programs which can be, or are stored in the program channel. This value includes the number of programs and available program segments. The manufacturer determines how the program memory is managed when downloading and deleting programmes in the program library and assessing what value *MaxLibraryIndex* is set to. The hardware in the programmable device and the Program Invocation Management defines the total number of programs to access and select.

For the PD 5000 operating system, there are 2 types of memory and only 1 program for each memory type, giving *MaxLibraryIndex = 2*.

SWNo. 8C: LibraryProgramID
LibraryProgramID corresponds completely with *ProgramID*, but is used to identify the program in the program library.

A number, *LibraryIndex*, which is found in *LibraryControl*, chooses a program from the library.

LibraryProgramID is a record of the same type as *ProgramID*. The description of *ProgramID* is found at SWNo. 82.

SWNo 8D: Maintenance
The *Maintenance* variable is used for service management and maintenance purposes, and holds the last date of service and an indication of the type of service.

```
Record
        Date            : Byte;         (* Offset = 0 *)
        Month           : Byte;         (* Offset = 1 *)
        Year            : Byte;         (* Offset = 2 *)
        Category        : Byte;         (* Offset = 3 *)
end
```

SWNo 8E: ChType

```
Record
        ChannelType     : Word;
        Exist           : Array[0..15] Of Boolean;
        InterpreterVers : Word;
        CodeType        : Word;
End
```

For the Program Channel, *ChType* has the following value:

ChannelType = 11
Exist =

15								7							0
1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1

CodeType indicates which kind of program code can be executed in the Program Channel.

InterpreterVers = 400 (version 4.00)
CodeType = 3 (Motorola 68020 machine code)

SWNo 8F: CHError
No errors can be reported in this channel.

9 Process-Pascal program channel (channel 9)

The Process-Pascal program channel provides the posibility to download and run Process-Pascal programmes in the PD 5000 controller, without changing the EPROM. The program can be downloaded to FLASH or RAM, but can be run in EPROM, FLASH or RAM. Selection of EPROM, FLASH or RAM program is performed from a library.

The programs in the library are indexed by a number, *LibraryIndex*, which is found in *LibraryControl*. The total number of programs that can be stored in the library-memory, is stated in *MaxLibraryIndex*, which is found in *LibraryStatus*.

Registers in Program Channel — Channel identifier: **PPProgCh**

SWNo.	Identifier	Memory Type	Read Out	Type
90	ProgramControl	BatteryRAM	- - - - - - -	Record
91	ProgramStatus	RAM Read Only	- - - - - - -	Record
92	ProgramID	Read Only	- - - - - - -	Record
93	TaskControl	BatteryRAM	- - - - - - -	Record
94	TaskStatus	RAM Read Only	- - - - - - -	Record
95	SystemPointer	Read Only	Hex	LongInteger
96				
97	MemoryInfo	Read Only	- - - - - - -	Record
98	IDAndCode	Special function	- - - - - - -	Record
99	ChConfig	BatteryRAM	- - - - - - -	Record
9A	LibraryControl	BatteryRAM	- - - - - - -	Record
9B	LibraryStatus	RAM Read Only	- - - - - - -	Record
9C	LibraryProgramID	Read Only	- - - - - - -	Record
9D	Maintenance	BatteryRAM	- - - - - - -	Record
9E	ChType	PROM Read Only	- - - - - - -	Record
9F	ChError	RAM Read Only	- - - - - - -	Record

SWNo. 90: ProgramControl
ProgramControl is used to set and change the state of the current program, which has been selected and invoked, via the Program Channel. The selected program number is inserted and indicated as a part of *ProgramControl*. Commands can be sent to *ProgramControl* to stop, start or reset the program.

```
Record
        Command         : Byte;
        ProgramToSelect : Word;
        ErrorStatus     : Bit32;
End
```

Command – Please refer to the descriptions in the Operating system chapter.

ProgramToSelect holds the library index for the program to select or the already selected program. *ProgramToSelect* is copied from *ChConfig* following a module reset or power up. It can hold the values 1 for EPROM, 2 for FLASH or 3 for RAM.

No functions are implemented for *ErrorStatus*.

SWNo. 91: ProgramStatus
ProgramStatus summarises the state and error conditions for the selected program. The library index for the selected program is also indicated.

```
Record
        State           : Byte;
        SelectedProgram : Word;
        ErrorStatus     : Bit32;
End
```

State - Please refer to the descriptions in the Operating system chapter.

SelectedProgram holds the library index for the selected program. *SelectedProgram* is 0 if *State* is *Non-selected*.

ErrorStatus is identical with *ProgramControl.ErrorStatus*, but only read access is possible.

SWNo. 92: ProgramID
Please refer to the descriptions in the Operating system chapter.

SWNo. 93: TaskControl
No functions are implemented for *TaskControl*.

SWNo. 94: TaskStatus
No functions are implemented for *TaskStatus*.

SWNo. 95: SystemPointer
This variable holds a pointer to system specific data. These system data can be used for debugging, reading out of kernel data etc. Only the manufacturer of the device and the creator of e.g. the debugger, will understand the information associated with the *SystemPointer*.

SWNo. 97: MemoryInfo
For each program segment, selected by *LibraryControl.LibraryIndex*, corresponding memory information can be read. The memory information holds the actual size for the program, the max. size for the program segment and a code for the memory type in which the program is stored.

MemoryInfo is a record of the following type:

```
Record
        ActualSize      : LongInteger;
        MaxSize         : LongInteger;
        MemoryType      : Word;
End
```

ActualSize includes the size of the program code and the header with the *ProgramID*.

MaxSize indicates the max. size for the program segment, and is the max. size for a program to download within the memory area specified by ***MemoryType***.

The values for and interpretation of *MemoryType,* are specified by the International P-NET User Organization. This description is available in a separate document, No. 590 003, and will be issued on request.

SWNo. 98: IDAndCode
Please refer to the descriptions in the Operating system chapter.

Size of the SWTable is given in the first 4 bytes in *ProgramCode*.

SWNo. 99: ChConfig
The specification of how the selected program should operate after a reset or power failure, is held in *ChConfig*. This configuration includes a number specifying which program must be invoked after reset.

```
Record
        Enablebit       : Array[0..7] Of Boolean;
        Functions       : Byte;
        Ref_A           : Byte;
        Ref_B           : Byte;
End
```

Enablebit[0] indicates how the selected program should operate after a power failure or module reset.

Enablebit[0] = *TRUE* indicates that the selected program must perform an autostart, resulting in *State* = *Running*.

Enablebit[0] = *FALSE* indicates that the selected program must not autostart, but must go to *State* = *Idle*.

The Program Invocation Management in the Program Channel must send Commands to the selected program after a module reset, to change *State* to that specified by *Enablebit[0]* (*Idle* or *Running*).

Functions is not used.

Ref_A holds the selected program number to be invoked following a module reset or power up. If *Ref_A* = *0*, 1 is automatically inserted, selecting EPROM.

Ref_B is not used.

SWNo. 9A: LibraryControl
Please refer to the descriptions in the Operating system chapter.

The value of *LibraryIndex* must be 1, 2 or 3.

SWNo. 9B: LibraryStatus
LibraryStatus is used to read the current state of a program in the library. *LibraryIndex* indicates the chosen program in the library. *MaxLibraryIndex* indicates the max number of programmes in the library.

LibraryStatus is a record of the following type:

```
Record
        State           : Byte;
        LibraryIndex    : Word;
        MaxLibraryIndex : Word;
End
```

State - Please refer to the descriptions in the Operating system chapter.

MaxLibraryIndex indicates the max. number of programmes which are, or can be stored in the program channel. This value includes the number of programs and available program segments. The manufacturer determines how the program memory is managed when downloading and deleting programmes in the program library and monitoring what value *MaxLibraryIndex* is set to. The hardware in the programmable device and the Program Invocation Management defines the total number of programs for access and selection.

In the PD 5000 controller, there are 3 types of memory and only 1 program for each memory type, giving *MaxLibraryIndex = 3*.

SWNo. 9C: LibraryProgramID
LibraryProgramID corresponds completely with *ProgramID*, but is used to identify the program in the program library.

A number, *LibraryIndex*, which is found in *LibraryControl*, selects a program from the library.

LibraryProgramID is a record of the same type as *ProgramID*. Please refer to the Operating system chapter for a description of *ProgramID*.

SWNo 9D: Maintenance

The *Maintenance* variable is used for service management and maintenance purposes, and holds the last date of service and an indication of the type of service.

```
Record
        Date        : Byte;     (* Offset = 0 *)
        Month       : Byte;     (* Offset = 1 *)
        Year        : Byte;     (* Offset = 2 *)
        Category    : Byte;     (* Offset = 3 *)
end
```

SWNo 9E: ChType

```
Record
        ChannelType     : Word;
        Exist           : Array[0..15] Of Boolean;
        InterpreterVers : Word;
        CodeType        : Word;
End
```

For the Program Channel, *ChType* has the following value:

ChannelType = 11

Exist =

15								7							0
1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1

CodeType indicates the kind of program code that can be executed in the Program Channel.

InterpreterVers = 400 (version 4.00)
CodeType = 4 (Process-Pascal)

SWNo 9F: CHError
No errors can be reported in this channel.

10 KeyMouse channel (channel $A)

Variables in KeyMouse channel — Channel identifier: **KeyMouse**

SWNo	Identifier	Memory type	Read out	Type
A0	KeyboardBuffer	BatteryRAM	- - - - - -	Buffer
A1	StatusIndicator	BatteryRAM	- - - - - -	Bit8
A2	KeyConvertTable	BatteryRAM	- - - - - -	Array
A3	Typematic	BatteryRAM	- - - - - -	Record
A4				
A5	MouseBuffer	BatteryRAM	- - - - - -	Buffer
A6	MouseSetup	BatteryRAM	- - - - - -	Record
A7				
A8				
A9	ChConfig	BatteryRAM	- - - - - -	Record
AA				
AB	InputStringPtr	BatteryRAM	- - - - - -	Pointer
AC	InputField	BatteryRAM	- - - - - -	Record
AD				
AE	ChType	PROM Read Only	- - - - - -	Record
AF	ChError	RAM Read Only	Binary	Record

This channel is used for keyboard and mouse control.

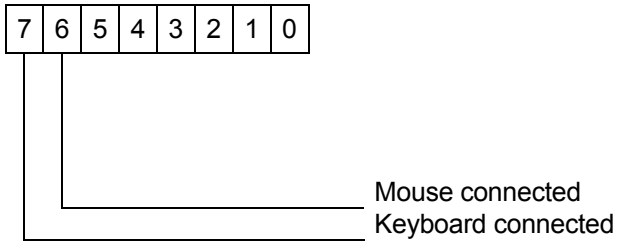
SWNo A0: KeyboardBuffer

Buffer[10] of Byte

Data received from the keyboard is transferred to the buffer directly, or through *KeyConvertTable*. If interrupt on "External Store" is connected to this variable, an interrupt is generated when data is transferred to the buffer.

Please refer to chapter 17 to get a survey of the key codes for PD 5010 and PD 5015.

SWNo A1: StatusIndicator



The *StatusIndicator* can be used to check whether a keyboard or a mouse is connected to a PD 5020 VGA Controller.

SWNo A2: KeyConvertTable

```
Array[1..128] of
        Record
                Normal      : Array[1..2] of Byte;
                Shift       : Array[1..2] of Byte;
                CapsLock    : Array[1..2] of Byte;
                NumLock     : Array[1..2] of Byte;
        End
```

Example of *KeyConvertTable*:

ScanCode	Normal	Meaning	Shift	Meaning	CapsLock	Meaning	NumLock	Meaning
1	00 43	F9	00 5C	Shift-F9	00 43	F9	00 43	F9
. . .								
22	31 00	1	21 00	!	31 00	1	31 00	1
. . .								
28	61 00	a	41 00	A	61 00	a	61 00	a
. . .								
41	20 00	Space	20 00	Space	20 00	Space	20 00	Space
. . .								
105	00 4F	End	31 00	1	00 4F	End	31 00	1
. . .								

If the first code in the *KeyConvertTable* entry is = 00, both codes are transferred to the *KeyboardBuffer* (only 1 interrupt is generated). If the first code is <> 00, only the first code is transferred to the buffer.

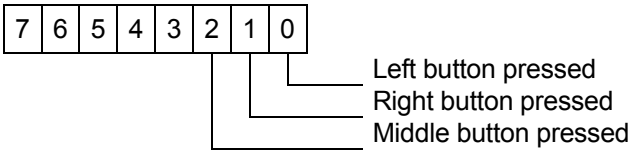
SWNo A3: Typematic
No functions are implemented for *TypeMatic*.

SWNo A5: MouseBuffer

```
Buffer[10] of
        Record
        x               : Integer;
        y               : Integer;
        State           : Bit8;
End
```

Data received from the mouse is transferred to the buffer. If interrupt on "External Store" is connected to this variable, an interrupt is generated when data is transferred to the buffer.

State:



SWNo A6: MouseSetup
No functions are implemented for *MouseSetup*.

SWNo A9: ChConfig

```
Record
        Enablebit       : Bit8;         (* Offset = 0 *)
        Functions       : Byte;         (* Offset = 1 *)
        Ref_A           : Byte;         (* Offset = 2 *)
        Ref_B           : Byte;         (* Offset = 3 *)
end
```

where each field has the following usage:

Enablebit: Not used.

Ref_A **:** UpdateChar, used when the *Inputstring* is placed in updatefield.

Ref_B **:** Not used.

SWNo AB: InputStringPtr
A pointer to the input string, from where data are fetched during *PerformUpdate*.

SWNo AC: InputField

```
Record
        InputX          : Integer;
        InputY          : Integer;
        InputLength     : Byte;
end
```

InputX and *InputY* defines the position on the screen for the *InputString*.

InputLength determines the length of the *InputString* and is used by the keyboard program.

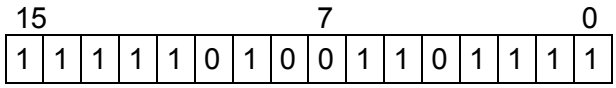
SWNo AE: ChType

```
Record
        ChannelType: Word;              (* Offset = 0 *)
        Exist: Bit16;                   (* Offset = 2 *)
        Functions: Bit16;               (* Offset = 4 *)
end
```

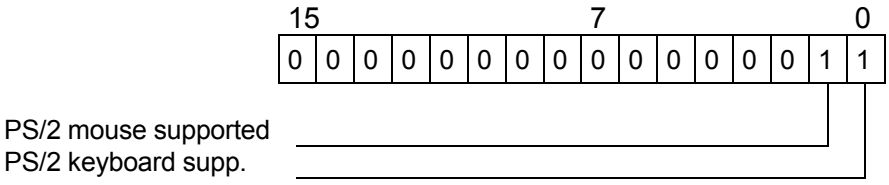
ChType has the following value:

ChannelType = $8005

Exist =



Functions =



SWNo AF: CHError
No errors can be reported in this channel.

11 Display channel (channel $B)

Variables in Display channel — Channel identifier: **Display**

SWNo	Identifier	Memory type	Read out	Type
B0	DumpData	BatteryRAM	- - - - - -	Record
B1	DumpStatus	BatteryRAM	- - - - - -	Bit8
B2	ScreenInfo	BatteryRAM	- - - - - -	Record
B3	CursorHide	BatteryRAM	- - - - - -	Array
B4	VideoControl	BatteryRAM	- - - - - -	Record
B5	ColorTable	BatteryRAM	- - - - - -	Array
B6				
B7				
B8				
B9	ChConfig	BatteryRAM RPW	- - - - - -	Record
BA				
BB				
BC				
BD				
BE	ChType	PROM Read Only	- - - - - -	Record
BF	ChError	RAM Read Only	Binary	Record

This channel is used for display control, and for display dump (readout of display contents from P-NET).

SWNo B0: DumpData
This variable is a dummy variable that is used for bitmap data from the display, during display dump.

SWNo B1: DumpStatus
This variable is a dummy variable that is used for bitmap data from the display, during display dump.

SWNo B2: ScreenInfo
System information to set up the screen and cursor, is held in this variable.

```
Record
        Video               : BitMapPtr;
        Width               : Integer;
        Height              : Integer;
        CursorX             : Integer;
        CursorY             : Integer;
        CursorForeground    : Byte;
        CursorBackground    : Byte;
        Cursor              : BitmapPtr;
        ScreenX             : Integer;
        ScreenY             : Integer;
        ScreenWidth         : Integer;
        ScreenHeight        : Integer;
End
```

Video holds a pointer to the screen (the video RAM). It is not possible to access the display directly via this pointer. The pointer is set up by the standard procedure *SetScreen*.

Width and *Height* defines the width and height of the screen, in pixels. The values are set up by the standard procedure *SetVideo(x,y)*.

CursorX, *CursorY* defines the actual position of the cursor. It is used for reading only (the cursor cannot be moved by writing to these values). The cursor position is changed by means of the standard procedures *CursorToAbs(x,y)*, *MoveCursor(x,y)* or *CursorTo(x,y)*. Refer to Process-Pascal manual for further details.

CursorForeground, *CursorBackground* defines the foreground and background colours of the cursor. The colours can be accessed directly, or set by the standard procedure *SetCursorColors*. If the colours are accessed directly, the cursor will not change colour until it is moved.

Cursor holds a pointer to the cursor bitmap. The value of this variable is for internal use only. The pointer is set up by means of the standard procedure *SetCursor*.

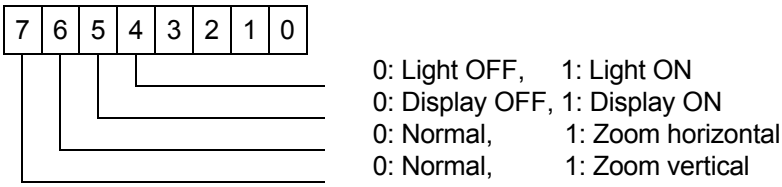
ScreenX and *ScreenY* are used in controllers with multiple windows. The standard procedure *SetWindow(x,y)* will insert x in *ScreenX* and y in *ScreenY*. Refer to Process-Pascal manual for further details.

ScreenWidth, *ScreenHeight* holds the physical width and height of the screen in pixels.

SWNo B4: VideoControl

```
Record
        Code        : Bit8;
        Light       : Byte;
        Contrast    : Byte;
End
```

Code has the following meaning:



Light This variable holds the light strength, if adjustable under software control, which is the case for PD 5010 / PD 5015.

Contrast This variable holds the contrast strength, if adjustable under software control, which is the case for PD 5010 / PD 5015.

The *VideoControl* variable can be accessed via the standard procedures *DisplayOnOff*, *LightOnOff*, *ContrastControl* and *LightControl*.

SWNo B5: ColorTable
No functions are implemented for this variable.

SWNo B9: ChConfig

The *ChConfig* variable is a record of the following type:

```
Record
        Enablebit       : Bit8;         (* Offset = 0 *)
        Functions       : Byte;         (* Offset = 1 *)
        Ref_A           : Byte;         (* Offset = 2 *)
        Ref_B           : Byte;         (* Offset = 3 *)
end
```

No functions are related to *ChConfig*.

SWNo BE: ChType

```
Record
        ChannelType     : Word;         (* Offset = 0 *)
        Exist           : Bit16;        (* Offset = 2 *)
        Functions       : Bit16;        (* Offset = 4 *)
end
```

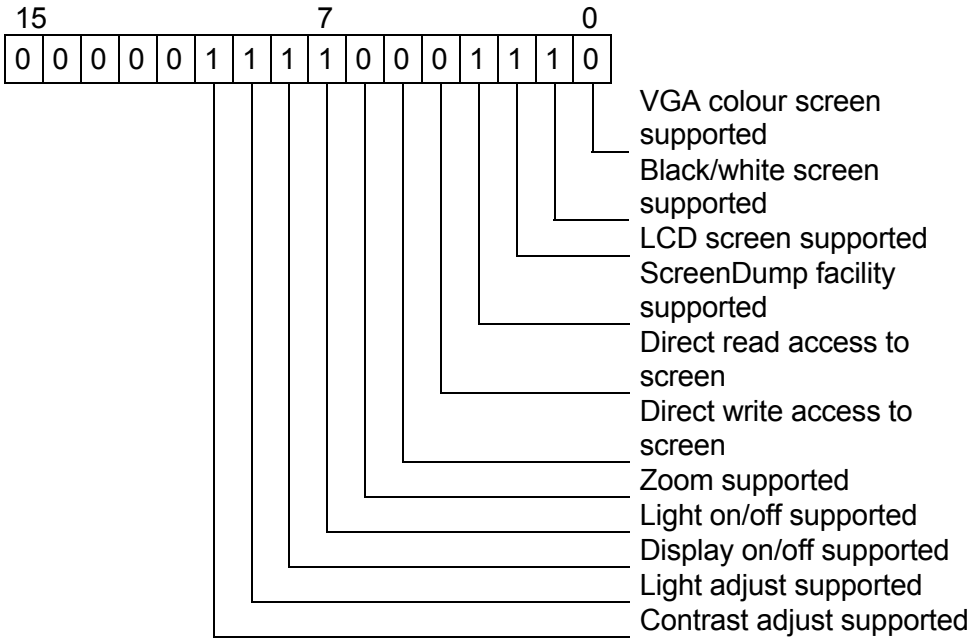
ChType has the following value:

ChannelType = $8006

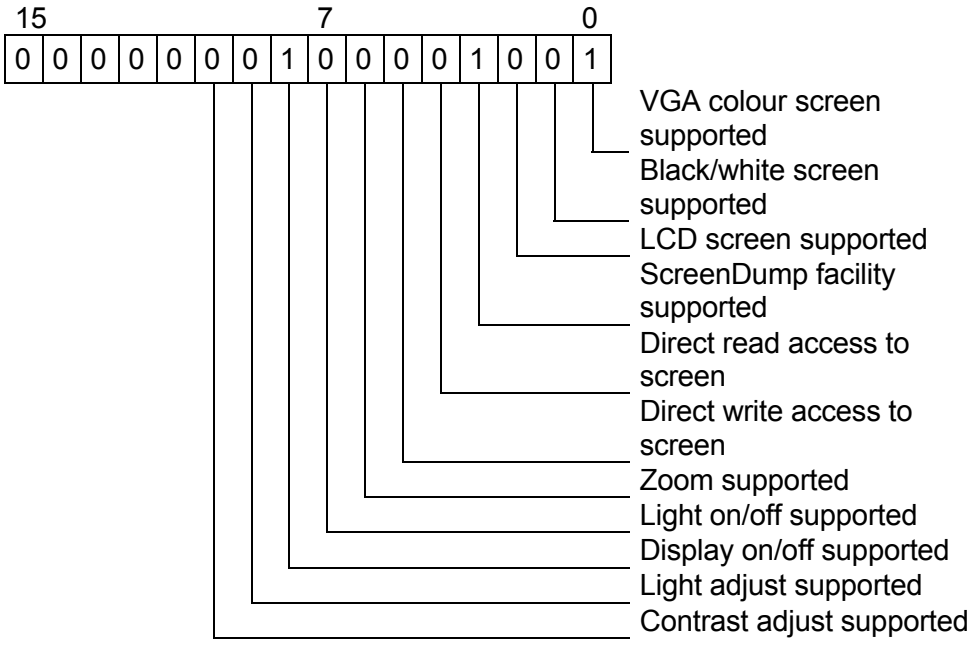
Exist =

15								7							0
1	1	1	0	0	0	1	0	0	1	1	1	1	1	1	1

Functions (for PD 5010/15) =



Functions (for PD 5020) =



SWNo BF: CHError
No errors can be reported in this channel.

12 Fixed softwire numbers ($100 - $113)

Softwire $100 - InterfaceErrorBuffer

The *InterfaceErrorBuffer* is a buffer where the element type is a record having the following structure:

```
Record
    SWNo        : Word;
    VARAddr     : LongInteger;
    VAROffset   : LongInteger;
    ErrorCode   : Word;
End

InterfaceErrorBuffer: Buffer[10] of InterfaceErrorRecord
```

The buffer can hold of a number of records with information on recently detected P-NET errors. An element is transferred to the *InterfaceErrorBuffer* by the operating system, when a P-NET error occurs. By means of the statement *Enable(Error)* in Process-Pascal, the user defines what type of P-NET errors will result in a transfer of an element to the *InterfaceErrorBuffer*. Refer to the Process-Pascal manual for further information on the *Enable(Error)* statement.

Since the variable *InterfaceErrorBuffer* is of type buffer, a complete element must be read, and it is not possible to just read a single field in a buffer element. A new variable of the same type as an element in the buffer should be declared. When an error occurs, the entire element can be transferred from the *InterfaceErrorBuffer* to the variable of type *InterfaceErrorRecord*. Now the fields of the variable can be accessed separately.

NOTE: When activating the automatic error detecting system and a report element is stored in the buffer, relevant program must be written to read this report element from the *InterfaceErrorBuffer*, to prevent the buffer from running full.
It is possible to connect a *SoftwireInterruptTask* to the *InterfaceErrorBuffer*. The corresponding *SoftwireInterruptTask* task will then automatically be activated each time an element is transferred to the buffer by the operating system.

The *InterfaceErrorRecord* is defined to include the following fields:

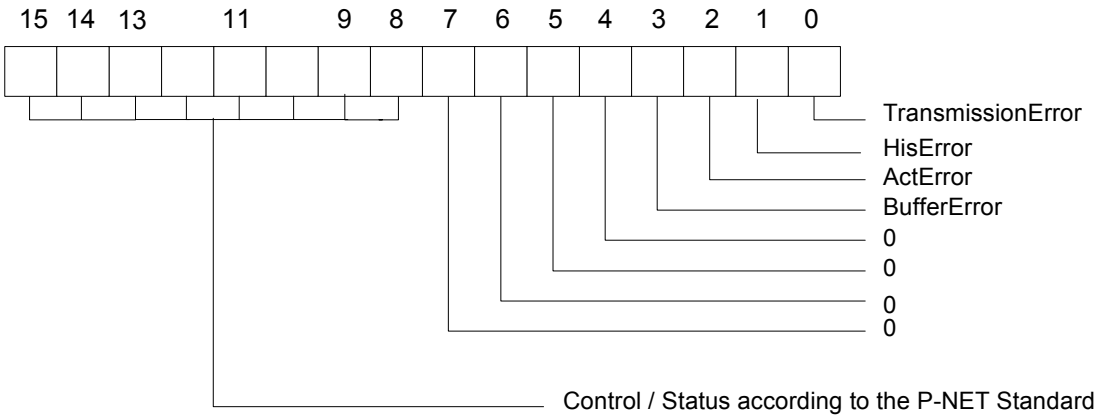
SWNo holds the Softwire number for the variable, within the declared interface module that caused the error. That is, the Softwire number of the external variable in the Softwire table in the PD 5000 controller.

The standard function ***VARNAME(SOFTWIRENo)*** returns the string constant after NAME for the module variable, if it is declared. Refer to chapter VARIABLE DECLARATION in the Process-Pascal manual.

VARAddr holds the logical address of the variable within the interface module. For simple interface modules (I/O modules), the contents of *VARAddr* is a number, which combines the channel number and the register number of the variable. If the module is a controller, *VARAddr* holds the Softwire number of the variable in the other controller that caused the interface error.

VAROffset holds an offset for the variable (in the interface module) that caused the interface error. The field variable *VAROffset* can be used to locate a variable field in a complex variable.

ErrorCode holds the error code relating to the interface error. The contents of the *ErrorCode* reflects the control/status field from the P-NET frame. The *ErrorCode* field is declared as a word, where each bit has the following meaning:



The format for the error code in PD 5000 is different from the error code that is used in PD 4000. A procedure, called *ConvertErrorCode* that is able to convert from the PD 5000 error code format to the PD 4000 error code format is available in the PD5000SP.INC file.

Softwire $101, ControllerCode
ControllerCode is a record of the following type:

```
Record
   MaxPowerdown   : LongInteger;
   CountryCode    : Boolean;
End
```

MaxPowerdown holds a time in seconds. If the duration of a power failure is shorter than *MaxPowerdown*, the controller continues program execution from where it was before the power failure. Otherwise the program restarts, like after a RESET.

CountryCode is used to select the decimal separator:

```
FALSE      : Separator = comma ( , )
TRUE       : Separator = point ( . )
```

Softwire $102, ActualPowerdownTime

ActualPowerdownTime: LongInteger;

This variable indicates for how long the controller was without power the last time it was powered down. If a Softwire interrupt task is connected to this variable, with interrupt condition ”InternStore”, the interrupt task will be activated after each power down. Please note that if the power failure lasted less than one second, *ActualPowerDownTime* will be zero, but if an interrupt is connected, this interrupt will still be activated.

Softwire $103 - NodeList

Nodelist is defined as an array of records, where the record is defined as follows:

```
Record
      Code        : Byte;
      StdChannel  : Boolean;
      DeviceType  : Integer;
      NodeAddr    : String[10];
End;
```

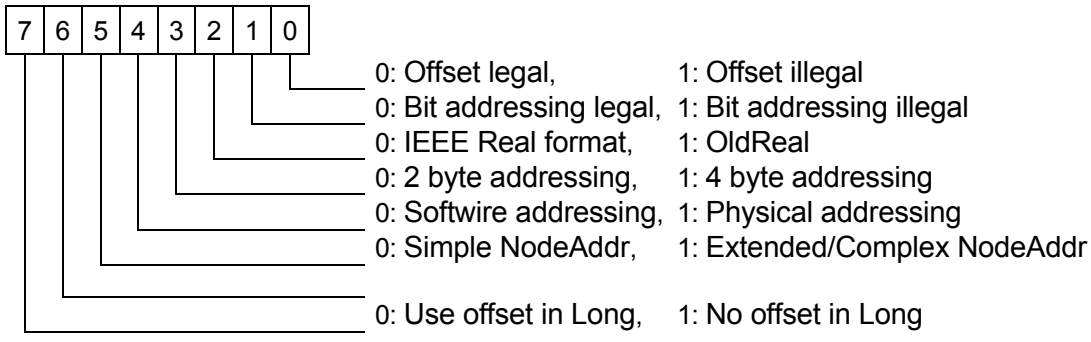
Nodelist: Array[1..10] of NodelistElement;

Nodelist is used for accessing variables, not directly declared in Process-Pascal. This is performed by means of the *PointerToNode* statement. How this is accomplished is described in the Process-Pascal manual.

Accessing variables through *Nodelist* is only possible from within the Process-Pascal program in this controller (there are no gateway functions related to the *Nodelist*).

The number of elements in *Nodelist*, and the length of the *NodeAddr* (max 25), is defined individually in the Process-Pascal program.

The ***Code*** field indicates the capabilities for the node to access, and is defined in the following way:



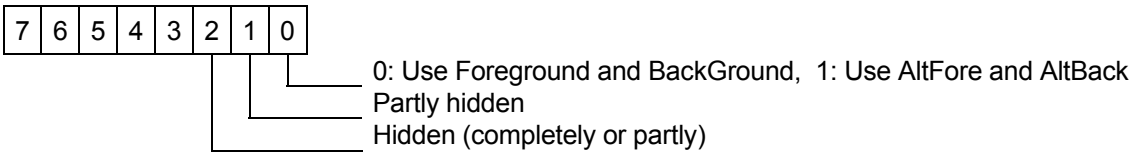
Softwire $104, DefaultPen

Writing on the display always requires a pen. If no pen is mentioned in the statement for writing - e.g. *Display(), DefaultPen* is used as default. If a local *DefaultPen* is declared, it will be used - otherwise the globally declared *DefaultPen* will be used. The pen holds information on charactergenerator, colours, window number, pen position etc.

A Pen is declared as a record of the following type:

```
Record
        CharGen     : CharacterGeneratorPtr;
        ForeGround  : Byte;
        BackGround  : Byte;
        RefX        : Integer;
        RefY        : Integer;
        AbsX        : Integer;
        AbsY        : Integer;
        Status      : Array[0..7] of Boolean;
        WindowNo    : Byte;
        AltFore     : Byte;
        AltBack     : Byte;
End
```

Status is interpreted in the following way:



The *Status* bits are mainly used in connection with the PD 5020 VGA Controller.

Status[1] and *Status[2]* may be used to reduce the time taken to update the display. *Status[1]* is set TRUE by the operating system, if the item to write on the display is partly hidden under another window. *Status[2]* is set TRUE if the item is either completely or partly hidden:

Status[2,1]:	0,0	: Not hidden
	0,1	: Illegal
	1,0	: Completely hidden
	1,1	: Partly hidden

The flags must be cleared by Process-Pascal. They are never cleared by the operating system.

Softwire $105 - PDBoxDefinition

PDBoxDefinition = Array[0..0] of Word

This is an array of constants, generated automatically by the compiler. The constants in the array are the Softwire numbers of all global variables that are declared to be external in the Process-Pascal program within the controller.

PDBoxDefinition can for example be used to initialise the modules, in which the external variables are located. Refer to the Process-Pascal manual, or the description (in this manual) about initialising modules (the INITBOX.INC file).

Softwire $106 - ExtTimeDate
This is the real-time clock. The real-time clock is of the same type as the *TimeDate* variable that is found on the Service channel. Please refer to the Service channel for further information.

Softwire $113 - ResetCode

ResetCode is a byte, holding a code that indicates the reason for the last reset.

The *ResetCode* is defined as follows:

$00 : STORE $FF to SW $03
$01 : Master Reset
$02 : Reset button pressed
$03 : Watchdog
$04 : Power fail
$05 : Instruction Error
$06 : Address error
$07 : Privilege violation
$08 : Illegal address (Bus error)

13 SERVICE program.

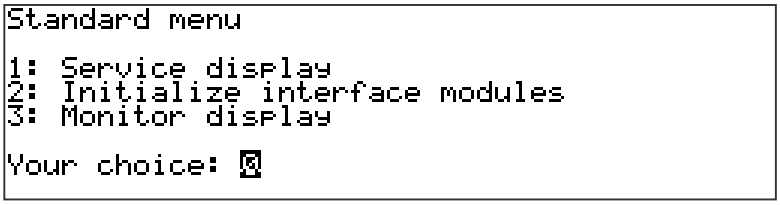
The SERVICE program is an engineering package for PD 5010, which enables you to monitor and change all variables in the interface modules specified in YOUR Process-Pascal program. All modules produced by PROCES-DATA and modules utilising the standardised general purpose channel types can be accessed.

Furthermore, it is used for automatic checking and configuration of all declared interface modules, and a MONITOR display is also implemented. The SERVICE program is a very helpful tool during the installation/test phase, and in small application programs it can significantly reduce your own programming effort in connection with configuration and other adjustments.

The "skeleton" of how to implement these facilities in your own Process-Pascal program, is shown in the file SERCON.PP, which can be found in the EXAMPLES folder within the Process-Pascal suite of programs.

The entire source code for the SERVICE program is found in the INC folder.

Calling the procedure *StandardMenu* (screen layout is found in the file MENU.INC) activates the SERVICE program. From this menu, three different tools can be selected:



Each of the tools is described in the following pages.

The SERVICE program operates with 5 function keys, all specified in the file SERVKEYS.INC:

Key no. $19 is "previous page", and selects the previous page for a channel.
Key no. $1A is "next page", and selects the next page for a channel.
Key no. $1B is "previous channel", and selects the previous channel in a module.
Key no. $1C is "next channel", and selects the next channel in a module.
Key no. $30 is "new display", and exits the SERVICE program.

The key No. for these function keys can be changed to suit your own application.

SERVICE DISPLAY

The Service display can operate on and display information for:

1 INTERNAL simple variables,
2 CHANNELS in Interface modules

1 INTERNAL simple variables, which are found via a SoftWire number, are selected by entering the **SWNo** in line 2. By entering a legal SWNo, information is displayed for the variable found.

```
PD5010 Service                    1
Softwire no  : $0001   Type:  Integer
Variable name:
P-NET address: Internal variable
Readout code :  5: 0

Value :  5010
```

Only global variables, which are specified in your Process-Pascal program, can be accessed. Refer to the MAP file, for a complete list of global variables.

The read-out Code for the value can be changed. The format corresponds to the format specified for the standard procedures Display/Update in Process-Pascal.

The variables can be changed in the field for **Value**.

2 CHANNELS in Interface modules, which can be found either via a SoftWire number, a NAME or a channel number.

```
PD5010 Service        Page 1
Softwire no  : $01A0   Module type: 3230
Variable name: 2 Cheese vat 3230
Channel no   : 00   Service channel

Device type  : 3230   Prg. version :  104
Manufacturer:    1   Proces-Data DK
```

The **SWNo** is entered in line 2. By entering a SWNo for a module, information is displayed for the service channel. By entering a SWNo for a channel declared as an indirect variable, information is displayed for the selected channel.

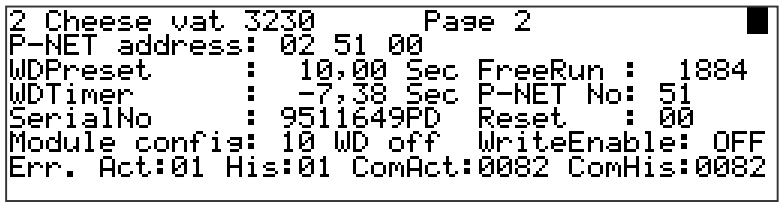
The **NAME** corresponds to the NAME assignment in the variable declaration. Only variables declared with NAME can be found this way. When entering a NAME, or a part of a NAME, all NAMEs in the controller are searched and compared with the *InputString*.

The search is performed backwards, which means that the search string (*InputString*), must be equal to the **last** part of the NAME. When the search string is found, information is displayed for the found channel.
The search string for NAME is entered in line 3 after "Variable name:".
(When you create a program and specify the NAMEs, it's recommended to append a unique numerical identifier to the NAME, to make it easy to enter the search string via the numeric keys, e.g. 200-45.2, for location, module No. and channel No., or module type e.g. 3230).

When an interface module has already been found already, by SWNo or by NAME, a **Channel** number can be entered in line 4, to select a new channel within the same module.
The function keys "previous channel" and "next channel" can also be used to select a new channel within the same module.

The complete information for a channel is displayed on several pages. The function keys "previous page" and "next page" can be used to select a new page for the same channel.

Page 2 for a Service channel is shown below:



Refer to the manuals for the different modules, to get more detailed information about the individual variables and their function.

The bottom line of all channel displays is reserved to display transmission errors for the module.

CHANGING THE P-NET ADDRESS.

Line 2 displays the P-NET address, which is defined in the Process-Pascal program for the module. If no access is obtained to the module, the P-NET address can be changed, to set the correct address. If the P-NET address is changed, there is no check of the module type (see below). This feature enables you to access additional modules, which were not declared in the program.

If a different module type is found than that specified in the variable declaration in the Process-Pascal program, a new display is shown, and you must confirm or cancel the selection.

```
Wrong module type found: 3120

Enter actual module type to continue,
or select new display to cancel.3230

Actual module type:    0
```

If you enter the actual module type (3120 in the example above), normal access is obtained to this module and all the channels in the actual module can be scanned, but the "Variable name" relates to the originally entered NAME or SWNo.

The program can be exited at any time, by pressing the key "new display", and it then returns to "Standard menu".

Only a few of the channel types are shown in this description, but the structure and the access possibilities are identical for all the channel types.

The 4 pages below survey a Digital I/O channel:

```
PD5010 Service          Page 1     3230_
Softwire no  : $01A0   Module type: 3230
Variable name: 2 Cheese vat 3230
Channel no   : 02   Digital in/out

Output       : 0    Counter   :    45
```

```
2 Cheese vat 3230       Page 2    3230_
P-NET address: 02 51 00
FlagReg : 01000010       ChError Act: 10
OutTimer:  -6,16 Sec             His: 10
OutPres.:  10,00 Sec
Current :  0,000 Amp    MinCurrent: 0,000
MaxCurr.:  1,000 Amp
```

```
2 Cheese vat 3230       Page 3    3230_
P-NET address: 02 51 00
Functions: 10  Output
Feedback  in=out: 00  in<>out: 00

Enable bit: 00000000
```

```
2 Cheese vat 3230       Page 4    3230_
P-NET address: 02 51 00
FBTimer :          6,58 Sec
FBPreset:         10,00 Sec
Operating time :    744 sec
Maintenance    : 00/00/00 - 00 (D/M/Y-Cat)
```

The 3 pages below survey an Analogue input channel:

```
PD5010 Service          Page 1     3221_
Softwire no  : $01A0   Module type: 3221
Variable name: 3 Contents vat 500
Channel no   : 08   Analog input

Input signal :  30,10
```

```
3 Contents vat 500      Page 2    3221_
P-NET address: 02 51 00

Input signal :  30,16

FullScale    : 32,000    ChError Act: 10
ZeroPoint    : 0,0000    ChError His: 10
```

```
3 Contents vat 500      Page 3    3221_
P-NET address: 02 51 00
LowLevel   :  15,00    HighLevel  :  25,00
Functions: 1B        Filter constant
Temperature Pt100      2,000 sec
Enablebit: 00010000
Maintenance    : 00/00/00 - 00 (D/M/Y-Cat)
```

Initialise interface modules

The second choice on the standard menu, selects a tool to check and initialise all the interface modules that are declared in your Process-Pascal program.

This tool is also used when an interface module is replaced.

CHECK MODULES.

In the first step, the P-NET network is scanned for to search for all the modules, which are declared in the Process-Pascal program.

When a module is found, a check is made that the module type corresponds with that declared in the program.

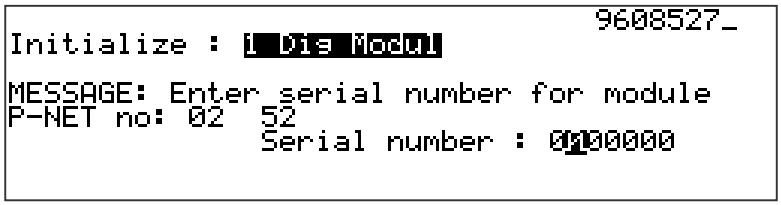
If a wrong module type is found, a new display is shown, and you are asked to skip initialising or to change the module and try again.

If the module not is found, a new display is shown, and you are asked to:

1 Skip initialising the module, or change the P-NET address on the motherboard to the expected address and try again (PD 1000 series modules),

or

2 Enter the SerialNo for the module (the serial number is printed on the module for PD 3000 series modules).



When all modules have been found or skipped, the program continues with the next step, to initialise modules.

The program can be exited at any time by pressing the key "new display", and it then returns to "Standard menu".

INITIALISE MODULES.

In the second step, all the configuration procedures that are defined in the variable declaration in your program after the CONFIG statement, are called and executed.

Each of the configuration procedures compares the value of the variable with the configuration value. If the value of the variable is not the same as the configuration value, the variable is set to the new value and it is checked again to confirm that the variable now has the correct value.

If the variable cannot be set to the configuration value, a new display is shown, depending on the module type, and you are asked to:

1 Skip initialising the module or "Set program enable switch ON" and try again (PD 1000 series modules),

or

2 A message is shown: "MODULE CONFIGURATION ERROR !!!" and you can skip initialising the module or try again (PD 3000 series modules).

When all the defined variables are configured or skipped, the program displays a status for the configuration (number of errors on modules and channels).

The program can be exited at any time by pressing the key "new display", and it then returns to "Standard menu".

The following procedures are found in the file **CONFIG.INC** and are available for defining a CONFIG statement:

SetLongInteger*(VAR CodeReg:LONGINTEGER; ConfigValue:LONGINTEGER);*
This procedure is used to configure the Code9 register in non-standardised channels.

SetReal*(VAR RealReg:REAL; ConfigValue:REAL);*
This procedure is used to configure a real value, e.g. SetPoint or HighLevel.

SetInteger(VAR IntegerReg:Integer; ConfigValue:Integer);

SetByte(VAR ByteReg: BYTE; ConfigValue:BYTE);
This procedure is used to configure any byte value, e.g. Functions in a *ChConfig* register.

SetBoolean(VAR BooleanReg: BOOLEAN; ConfigValue:BOOLEAN);
This procedure is used to configure any boolean value, e.g. a single *EnableBit* in a *ChConfig* register.

SetBit8(VAR Bit8Reg:Bit8; Bit8Byte:BYTE);
This procedure is used to configure any Bit8 register, e.g. every *EnableBit* in a *ChConfig* register. The boolean array occupies a byte and the configuration value for the boolean array must be represented in a byte, e.g. $80 sets bit 7 and clears all the other booleans.

PT100(VAR TempChannel:ChAnalogIn);
This procedure is only valid for **PD 1611 and PD 1652**.

Standard_PT100(VAR TempChannel:AnalogInCh);
This procedure is valid for all standardised analogue input channels.
The procedure selects the input type as "Pt100" with NO FILTER, clears all *EnableBits* and *Zeropoint* is set to 0.

DigitalInput(VAR InChannel:ChDigitalIO);
This procedure is only valid for PD3100 and PD3150. The Procedure sets the channel to "Input Only".

DigitalOutput(VAR OutChannel:ChDigitalIO);
This procedure is only valid for PD3100 and PD3150. The Procedure sets the channel to "Output".

Std_DigitalOutput(VAR OutChannel:DigitalCh);
This procedure is valid for all standard digital I/O channels. The Procedure sets the channel to "Output".

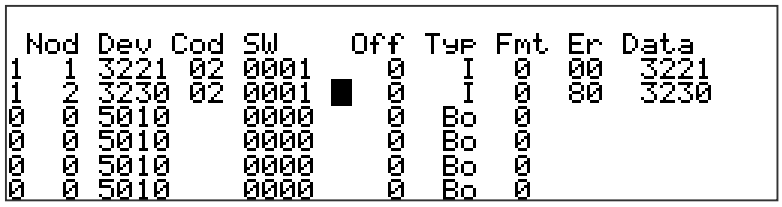
OutputWithFeedBack(VAR OutputChannel:ChDigitalIO; ChannelNumberA, ChannelNumberB:BYTE; PresetTime :REAL);
This procedure is only valid for PD3100.

Monitor display

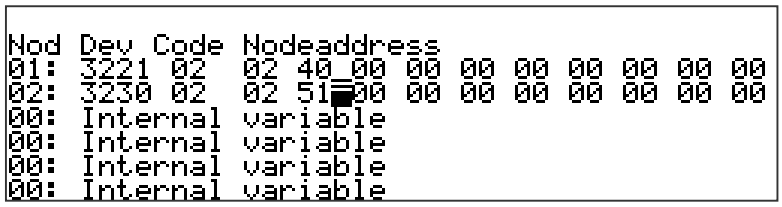
The third choice on the standard menu selects a 6 line MONITOR display, including help pages. The facilities in this MONITOR are similar to the facilities in the VBMON program for a PC. You can access all global variables on the entire P-NET network.

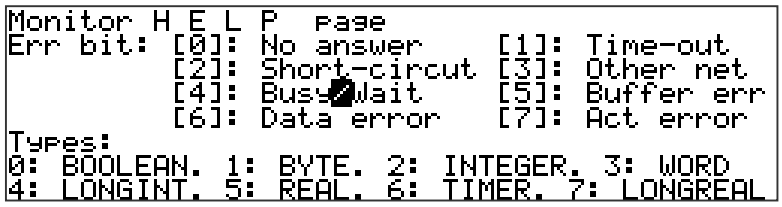
Please observe that the MONITOR program uses the first 6 elements from the *NodeList*.

Nod denotes an element from the *NodeList*. **Dev** is the device type. **Cod** is automatically set when a known device type is selected (Cod specifies the access mode: 2 or byte addr, offset etc.). **SW** selects the SoftWire no.. **Off** is an offset. **Typ** specifies the variable type to access. **Fmt** specifies the readout mode for the variable. **Er** is the error code. **Data** is the actual loaded data.



Nodeaddress specifies the complete node address to access the module, including port number. Node number 0 selects an internal variable.



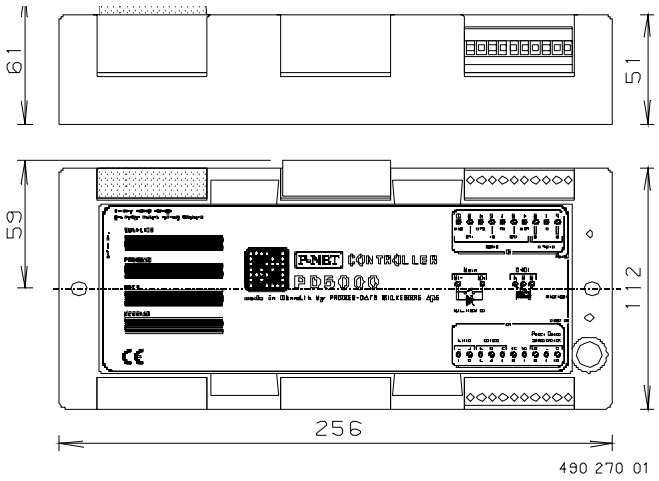


14 Construction, Mechanical.

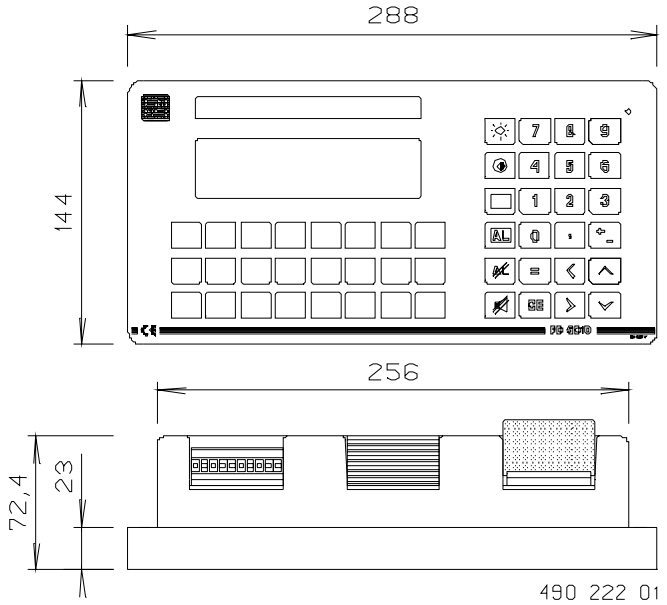
The PD 5000 controller is housed in a black plastic case. The module is designed for plugging directly on to a mounting rail (EN 50 022 / DIN 46277), or for front panel mounting. The module incorporates two snap connectors, which provide the terminals for field connection, power and communications. The PD 5020 also provides an interface for a PS/2 mouse and a standard PC keyboard.

Below are the mechanical diagrams for the various Controller configurations.

PD 5000/PD 5020 **Scale Drawing (in mm)**



PD 5010/PD 5015 **Scale Drawing (in mm)**



Materials

Case (injection moulded):	Black NORYL GFN
Front foil:	Polycarbonate
Back plate:	Black anodised aluminium

Weight.

PD 5000	0.9 Kg
PD 5010/PD 5015	1.5 Kg
PD 5020	1.0 Kg

15 Specifications.

All electrical characteristics are valid at an ambient temperature -25 ºC to +70 ºC, unless otherwise stated.

All specifications apply within the approved EMI conditions.

Power supply.

Power supply DC:	nom 24.0 V
	min 20.0 V
	max 28.0 V
Ripple:	max 5 %
Power consumption:	max 6.0 W

Program storage.

PD 5000/PD 5010/PD 5015:

RAM Memory size:	512 Kbytes
FLASH Memory size:	512 Kbytes
With extension:	1 Mbyte FLASH/1 Mbyte RAM

PD 5020:

RAM Memory size:	2.5 Mbyte
FLASH Memory size:	1.5 Mbyte

Display.

PD 5010:	256 by 64 pixels
PD 5015:	240 by 128 pixels
PD 5020:	Standard VGA colour monitor

Keyboard.

Provides a programmable key layout using sealed, membrane switch technology.

PD 5010:	48 keys
PD 5015:	44 keys
PD 5020:	PC keyboard and PS/2 mouse

Ambient Temperature.

PD 5000/PD 5020:

Operating temperature:	-25 ºC to +70 ºC
Storage temperature:	-40 ºC to +85 ºC

PD 5010/5015:

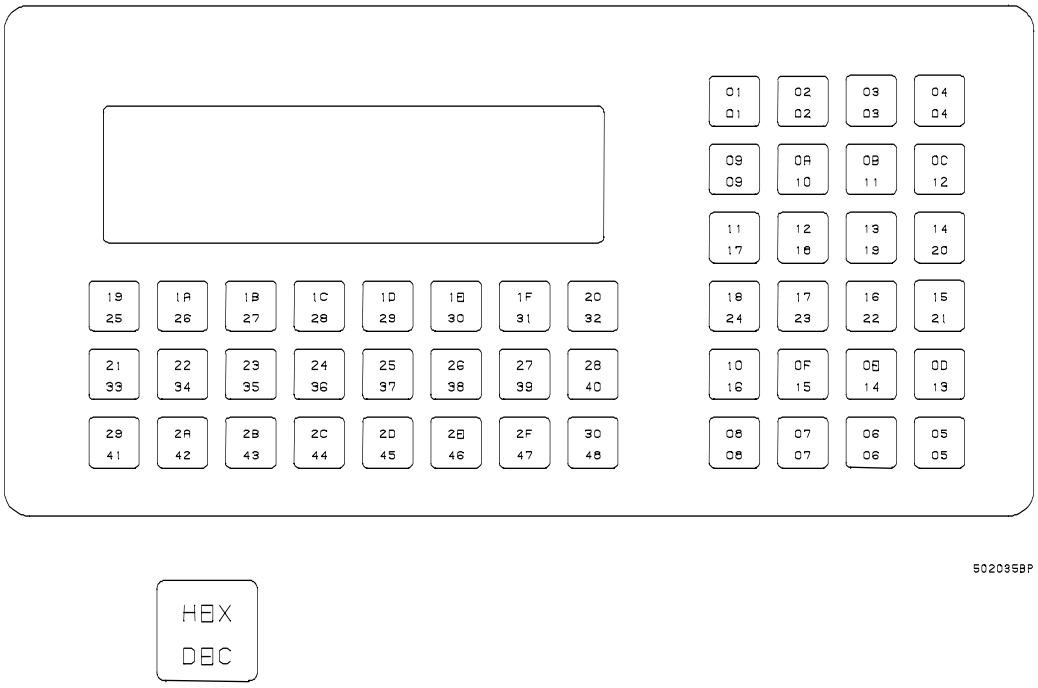
Operating temperature:	-10 ºC to +45 ºC
Storage temperature:	-20 ºC to +60 ºC

16 Approvals.

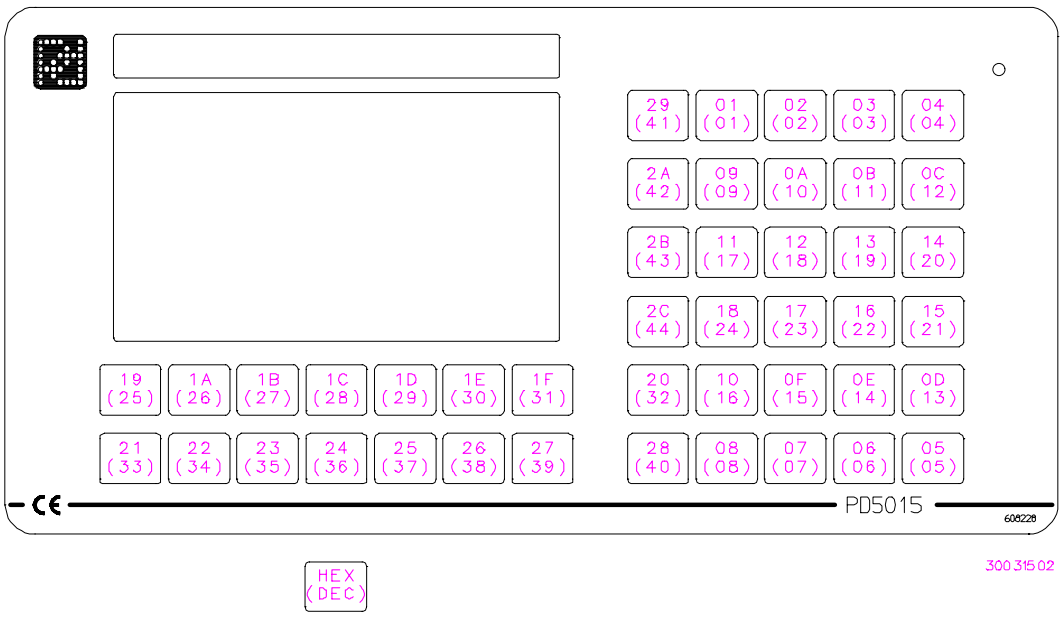
Compliance with EMC-directive no.:	89/336/EEC
Generic standards for immunity:	
Industry	EN 50082-2
Vibration (sinusoidal):	IEC 68-2-6 Test Fc

17 Key Codes for PD 5010 and PD 5015

Key codes for PD 5010:



Key codes for PD 5015:



18 Survey of channels in the PD 5000 controller.

SwNo.	Service 0	Communication 1,2,3	Gateway 5	Alarm Digital I/O 6	Program 8, 9	KeyMouse $0A	Display $0B
x0	NumberOfSWNo	ActualMode	GatewayRecord	Flagreg	ProgramControl	KeyboardBuffer	DumpData
x1	DeviceID	OutputBuffer	GatewayInterrupt	OutTimer	ProgramStatus	StatusIndicator	DumpStatus
x2		InputBuffer		Counter	ProgramID	KeyConvertTable	ScreenInfo
x3	Reset				TaskControl *	TypeMatic	CursorHide
x4	PnetSerialNo				TaskStatus *		VideoControl
x5					SystemPointer	MouseBuffer	ColorTable
x6	TimeDate					MouseSetup	
x7	FreeRunTimer				MemoryInfo		
x8					IDAndCode		
x9	ModuleConfig	ChConfig	ChConfig	ChConfig	ChConfig	ChConfig	ChConfig
xA					LibraryControl		
xB	Mailfilter				LibraryStatus	InputStringPtr	
xC	Mailbox	DefaultMode		UserLIArray	LibraryProgramID	InputField	
xD	WriteEnable			Maintenance	Maintenance		
xE	ChType	ChType	ChType	CHType	ChType	ChType	ChType
xF	CommonError	ChError	ChError	ChError	CommonError	ChError	ChError

* Channel 9 only