502 090 01

P-NET CONTROLLER

PD 4000

Manual

GB *May 1999*

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

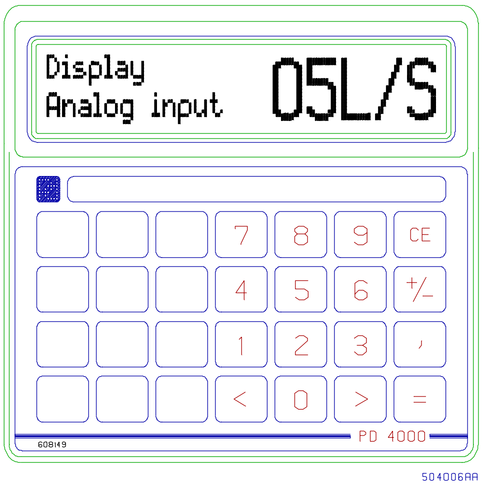
502 090 01

Contents

1 General information.

1.1 Features

* 16 Bit Microprocessor
* Programmed in High Level Multi Tasking Language
* Completely Sealed Construction
* Large Data Storage Capacity
* Membrane click-switch Keyboard
* Backlit Graphics LCD Display
* User Definable Overlay
* P-NET Fieldbus Communication
* Battery Back-up
* Real Time Clock



1.2 Application

The PD 4000 is designed as a control computing and display element in both highly complex or simple process control systems. It is used in conjunction with the collection of distributed input/output and control modules, which provide digital, analogue, flow and weighing facilities, via the P-NET fieldbus (EN 50170 Vol. 1). The Controller is completely sealed, and is therefore suitable for use in any industrial environment. The compact design and the outstanding environmental specifications for the Controller, makes it exceptional for machines and mobile applications.

1.3 Display

The display is a fast graphics LCD, using Supertwist technology, providing wide viewing angle. The display has a resolution of 150 by 20 pixels, enabling a variety of character fonts and graphics to be used, e.g. 3 lines with 25 characters each. The viewing area is 120 mm * 19.2 mm. An LED back light is incorporated. The display is protected with non-reflecting glass.

1.4 Keyboard

The keyboard is a membrane click-switch foil, with metal domes. The keyboard has 28 keys. The key functions depend upon the type of application, and are defined by the user program. The unique design includes a self adhesive keyboard foil, which provides the ability to customise the unit, and ensures an ideal operator/instrumentation interface.

1.5 Microprocessor

The Controller utilises a 16-bit HC 68001 microprocessor with a clock frequency of 9.8 MHz, giving it exceptional power and memory addressing capability.

1.6 Memory

The program memory consists of a flash EPROM of 256 K bytes, and a 64 K bytes boot PROM. The data memory has 512 K bytes of CMOS RAM with battery back up. In the event of a power failure, the Controller will save the current program state. When power is restored, control will either continue from the failure point, or reset, depending on instruction.

1.7 Real Time Clock

The Controller is equipped with a real time clock with battery back-up. It is configured for 24 hour format and enables the display or recording of real time, in seconds, minutes, hours, days, months and years.

1.8 Programming

The Controller is programmed in Process-Pascal, which is a multi-tasking high level language developed especially for the programming of process control activities, which utilise P-NET distributed interface modules. Process-Pascal is an extension of standard PASCAL. The compiled program is downloaded to the flash EPROM in the Controller via the P-NET interface. The powerful nature of the compiler enables a system designer to write independent processes as separate, testable tasks, and to define process elements, such as valves, sensors, keyboards and displays as named program variables. This makes it particularly easy to design control programs, which can also incorporate system instrumentation requirements.

1.9 P-NET Communication Interface

The Controller has a P-NET multi-master interface. P-NET uses the RS-485 Serial interface for communicating with P-NET interface modules, at a transmission speed of 76,800 baud.

1.10 Mechanical construction

The Controller is housed in a black injection moulded plastic enclosure. The entire Controller is completely filled with silicone. This construction makes it extremely resistant to water, dust and vibration.

The controller is made up of 2 parts, consisting of the main part, with built-in electronics, display and keyboard, and the power supply part, with power supply, battery for the CMOS RAM, and P-NET interface circuitry. The power supply part is attached to the main part by means of 4 screws.

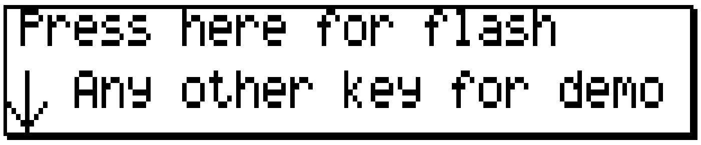
2 Getting started

2.1 Boot PROM program, Master reset

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

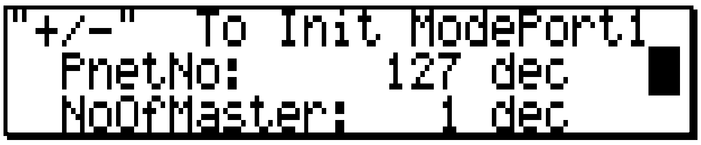
A Master reset is performed by detaching the power supply part of the controller for at least 10 seconds.
NOTE: NEVER disconnect the power supply part when power is turned on. This might DESTROY the controller.

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



```
Press here for flash
↓ Any other key for demo
```

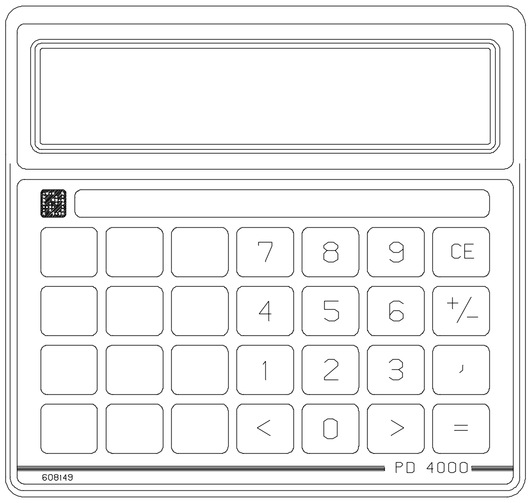
Pressing any key other than the one in the upper left corner, when the message ”Press here for flash” is shown on the display, will change the display to the following:



```
"+/-"  To Init ModePort1
 PnetNo:      127 dec   █
 NoOfMaster:    1 dec
```

Now the P-NET node address for the controller, and the number of masters, can be keyed in. It should be understood that the program for this routine is permanently stored in the boot PROM, and is thus the same in all controllers. However, the keyboard layout defined in the users application program may be different from that expected by the boot PROM program, perhaps with the “+/-“ key and others placed elsewhere. Therefore, the application keyboard overlay might NOT be in accordance with the boot PROM program.

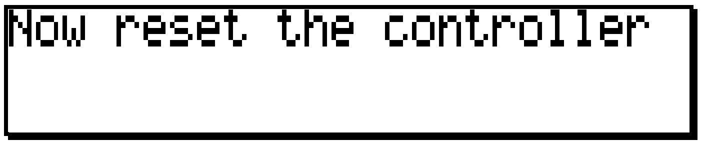
The boot PROM program expects the keys to be placed according to the default keyboard foil, as shown in this figure:



After keying in the P-NET node address and the number of masters, the ”+/-” key should be pressed. This will make the controller initialise the P-NET system according to the new values, and will start a small demo program. The demo program can be used to check that all the pixels in the display can be turned on and off, and gives an indication of the serviceability of the controller.

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

If a program has previously been downloaded to the flash memory, this program can be selected to run instead of the boot program. Following a Master reset, the display will show the first screen illustrated on the previous page. The arrow points to the key in the upper, left corner of the keyboard. Pressing this key will change the display to show the message:



```
Now reset the controller
```

When the controller has been reset, e.g. by turning the power off and on, or by using the reset button in the download utility, the Process-Pascal program in the flash memory will start.

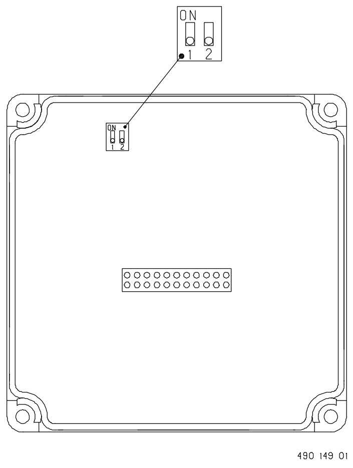
If no valid program has been downloaded to the flash memory, the attempt to start one will give an undefined result. It may then be necessary to perform another Master reset, and download a valid program to the flash memory.

2.2 Downloading programs

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

When a Process-Pascal program is to be downloaded to flash memory, it is ALWAYS downloaded together with 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. 50 K bytes of program memory.
Please refer to the VIGO Users Manual (ref.no. 502086), for further information on how to use the Download tool under Windows 95/98 or NT.
In order to enable download of a program to either flash or RAM, switch 1, which is mounted inside the power supply part of the controller, must be in position ON. The switch is placed inside the controller, to make it possible to seal the controller. This ensures that the program in flash cannot be changed without breaking the seal. This facility can be very useful when the controller is used in applications requiring some sort of official approval.

However, it also means, that if the switch is turned OFF, it is not possible to update the controller without having to give it a Master reset. Therefore, the switch should normally be left in the ON position, for applications not requiring the controller to be sealed. See the drawing below.
NOTE: When the switch is in the OFF position, physical addressing is not possible and it is therefore not possible to debug the program using the Process-Pascal debugger.



When a Process-Pascal program is to be downloaded to RAM, the version of the operating system in the controller must be compatible with the version of the Process-Pascal program to be downloaded. Versions are compatible when the first 2 digits in the version number are the same. For instance, version 2.31 is compatible with version 2.30, but NOT with version 4.00. Hence, if a Process-Pascal program is compiled with a compiler of version 4.0x, and the controller holds a boot PROM with an operating system of version 2.3x, it will NOT be possible to download to RAM, when the program in the controller is running in boot PROM. In this case, the Process-Pascal program should first be downloaded to flash together with version 4.0x of the operating system. When the controller program is running in flash, new Process-Pascal programs of version 4.0x can be downloaded to RAM.

Downloading of programmes to RAM can be performed using the VIGO program Pddownload.exe.

2.3 CMOS RAM

The PD 4000 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, leaving 504 K bytes free for use by a Process-Pascal program. The RAM memory is typically used for user data, variables, stack memory for tasks etc. associated with the Process-Pascal program, but may also be used for Process-Pascal code. However, normally the Process-Pascal code would be loaded into flash memory.

After a Master reset, the contents of the CMOS RAM are lost. The entire RAM is cleared to zeros, then the Process-Pascal program in the boot PROM is started, which uses about 1.5 K bytes of RAM. Thus, after a Master reset, the contents of this part of the RAM is undefined.

3 Programming

The PD 4000 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, being the >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 specifically for the PD 4000 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 these files.

4 System file

The PD4000.SYS file contains a declaration of the variables and constants that are needed by the controller's operating system. Most of the declarations should NOT be changed, as the operating system expects these system variables to be at specific Softwire numbers, and have specific types.

The PD4000.SYS file must always be included immediately before the Process-Pascal program. See the examples in the Process-Pascal library, e.g. PD4000.pp.

The {$L-} statement is a compiler directive, which indicates to the compiler that it should not output anything to the >xxx.LST< file, until the statement {$L+} is found.

The following TYPE declaration declares various RECORD structures for the P-NET system, the real time clock, the display etc.

The CONST and VAR part of the file declares the constants and variables placed in the first part of the SoftWire List. The SoftWire number of the constants / variables is shown as comments in the beginning of each line. Every globally declared constant / variable occupies one SoftWire number. It is the SoftWire number that provides the means to access these data via P-NET.

```
(*$L- *)
{===============================================================================}
{                                                                               }
{  ProjectName  : PD4000                                                        }
{  Unity        : INCLUDE                                                       }
{  Name and date: PD4000.SYS 22-03-99                                           }
{  Writer       : PROCES-DATA A/S                                               }
{  Modification :                                                               }
{                                                                               }
{===============================================================================}
{  Type identifiers and field identifiers must NOT be changed in the following }
{  part of this declaration                                                     }
{===============================================================================}

DEFINE
  MIBAccessSystem = [ MIB_NoBackup, MIB_InVisible ];

{$MIB MIBAccessSystem}

TYPE
  Memory = Real;
  Bit8  = ARRAY[0..7] OF BOOLEAN;
  Bit16 = ARRAY[0..15] OF BOOLEAN;
  Bit24 = ARRAY[0..23] OF BOOLEAN;
  Bit32 = ARRAY[0..31] OF BOOLEAN;

  UARTrec = RECORD
              PnetNo           : INTEGER;
              NoOfMaster       : INTEGER;
            END;

  DateTimeArr = ARRAY[0..7] OF BYTE;

  CharacterGeneratorType = ARRAY[0..0] OF BITMAP;
  CharacterGeneratorPtr = POINTER TO CharacterGeneratorType;
  BitMapPtr = POINTER TO BITMAP;
  StringPtr = POINTER TO STRING[255];
  ControllerCodeRec = RECORD
                        MaxPowerDown: LONGINTEGER;
                        UpdateSign  : CHAR;
                        InputStr    : StringPtr;
                        CountryCode : INTEGER;
                      END;

  PenInformationType = RECORD
                          CharGen    : CharacterGeneratorPtr;
                          ForeGround : BYTE;
                          BackGround : BYTE;
                          RefX       : INTEGER;
                          RefY       : INTEGER;
                          AbsX       : INTEGER;
                          AbsY       : INTEGER;
                          Status     : ARRAY[0..7] OF BOOLEAN;
                          WindowNo   : BYTE;
                        END;

  ScreenInformationType = 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;

  InterFaceErrorRecord = RECORD
                           SWNo       : WORD;
                           VARAddr    : LONGINTEGER;
                           VAROffset  : WORD;
                           ErrorCode  : WORD;
                         END;

  NodeListElement = RECORD
                      Code       : BYTE;
                      StdChannel : BOOLEAN;
                      DeviceType : INTEGER;
                      NodeAddr   : STRING[10];
                    END;

  IntRecordType = RECORD
                    SWNo: INTEGER;
                    Offset: INTEGER;
                  END;

  ErrorStringType = STRING[80];


CONST
(* 0 *) UsedSoftWireNumbers = INTEGER(0) [ MIBProperties = MIB_Visible]
                   (* To be patched by the compiler during link phase *);
(* 1 *) DeviceType          = 4000 [ MIBProperties = MIB_Visible];
(* 2 *) PdPrgVersion        = INTEGER(0400) [ MIBProperties = MIB_Visible];

VAR
(* 3 *) Error3          : BYTE [ MIBProperties = MIB_Visible ]
                            AT ADDRESS: $00FFFF03;
(* 4 *) SerialNumber    : LONGINTEGER [ MIBProperties = MIB_Visible ];

(* 5 *) SWNo_5          : BYTE AT ADDRESS: $FFFF02;
(* 6 *) DateTime        : DateTimeArr [ MIBProperties = MIB_Visible]
                            AT ADDRESS: $00FFFF24;
```

```
(* 7 *) FreeRunTimer    : LONGINTEGER [ MIBProperties = MIB_Visible]
                            AT ADDRESS: $00FFF806;
(* 8 *) SoftWire8       : LONGINTEGER;
(* 9 *) SWNo_9          : BYTE AT ADDRESS: $FFFF02;
(* A *) ModePort1       : UartRec [ MIBProperties = MIB_Backup, MIB_Visible ];
(* B *) SoftWireB       : INTEGER;
(* C *) SoftWireC       : INTEGER;
(* D *) SoftWireD       : INTEGER;
(* E *) SoftWireE       : INTEGER;
(* F *) ErrorF          : BYTE AT ADDRESS: $00FFFF05;
(*10 *) SoftWire10      : INTEGER;
(*11 *) SWNo_11         : BYTE AT ADDRESS: $FFFF02;
(*12 *) InterFaceErrorBuffer : BUFFER[10] OF InterFaceErrorRecord;
(*13 *) SWNo_13         : BYTE AT ADDRESS: $FFFF02;
(*14 *) IntVecTable     : ARRAY[0..31] OF Memory;
(*15 *) SoftWire15      : INTEGER;
(*16 *) BreakPoint      : INTEGER;
(*17 *) SWNo_17         : BYTE AT ADDRESS: $FFFF02;
(*18 *) ControllerCode  : ControllerCodeRec AT ADDRESS: $00FFFF30;
(*19 *) SWNo_19         : BYTE AT ADDRESS: $FFFF02;
(*1A *) DefaultPen      : PenInformationType;
(*1B *) SWNo_1B         : BYTE AT ADDRESS: $FFFF02;
(*1C *) ScreenInfo      : ScreenInformationType [ MIBProperties = MIB_Visible ]
                          AT ADDRESS: $FFFFC4;
(*1D *) SoftWire1D      : INTEGER;
(*1E *) SoftWire1E      : INTEGER;
(*1F *) SoftWire1F      : INTEGER;
(*20 *) SoftWire20      : INTEGER;
(*21 *) SoftWire21      : INTEGER;
(*22 *) ActualPowerDownTime : LONGINTEGER AT ADDRESS: $00FFFF2C;
(*23 *) SWNo_23         : BYTE AT ADDRESS: $FFFF02;
(*24 *) KeyboardBuffer  : BUFFER[10] OF BYTE [ MIBProperties = MIB_Visible ]
                            SOFTWIREINTERRUPT:0 [InternStore, ExternStore];
(*25 *) SWNo_25         : BYTE AT ADDRESS: $FFFF02;

CONST
(*26 *) DefaultModePort1 = UartRec(PnetNo: $2, NoOfMaster: $6);
(*27 *) SoftWire27       = INTEGER(0);
(*28 *) SoftWire28       = INTEGER(0);

VAR
(*29 *) Softwire29      : BYTE;
(*2A *) Softwire2A      : BYTE;
(*2B *) SWNo_2B         : BYTE AT ADDRESS: $FFFF02;
(*2C *) NodeList        : ARRAY[1..10] of NodeListElement
                            [ MIBProperties = MIB_Backup, MIB_Visible ];
(*2D *) SWNo_2D         : BYTE AT ADDRESS: $FFFF02;
(*2E *) CursorHide      : ARRAY[1..50] of INTEGER;   (* max cursorsize 32*25 *)
(*2F *) SoftWire2F      : INTEGER;
(*30 *) SoftWire30      : INTEGER;
(*31 *) SWNo_31         : BYTE AT ADDRESS: $FFFF02;
CONST
(*32 *) PDBoxDefinition = ARRAY[0..0] OF WORD ([0]:0);
                           (* To be patched by the compiler *)


{-------------------------------------------------------------------------------}
{  The rest of this declaration may be moved, but it contains various          }
{  declarations for the PD4000 LCD display                                     }
{-------------------------------------------------------------------------------}
(*$L+ *)

VAR
  Screen          : VIDEOBITMAP[900] [ MIBProperties = MIB_Visible ]
                      PLACE: $34 AT ADDRESS: $00FFE49A;
```

```
{$MIB MIB_NoWriteAccess, MIB_InVisible}

  contrastLCA1    : BYTE PLACE: $35 AT ADDRESS: $00200001;
  contrastKOPI    : BYTE PLACE: $36 AT ADDRESS: $00FFFFA5;
  LCDIns1         : BYTE PLACE: $37 AT ADDRESS: $00280000;
  LCDIns2         : BYTE PLACE: $38 AT ADDRESS: $00300000;
  LCDIns3         : BYTE PLACE: $39 AT ADDRESS: $00380000;
  ContrastValue   : INTEGER PLACE: $3A AT ADDRESS: $00FFFF08;
  AltFore         : BYTE PLACE: $3B AT ADDRESS: $00FFFF9A;
  AltBack         : BYTE PLACE: $3C AT ADDRESS: $00FFFF9B;
  StartCode       : BYTE [ MIBProperties = MIB_WriteAccess ]
                      PLACE: $3D AT ADDRESS: $00FFFF0B;

{$MIB MIB_WriteAccess, MIB_Visible}

  Year    -> DateTime[6];
  Month   -> DateTime[5];
  Day     -> DateTime[4];
  Hour    -> DateTime[2];
  Minute  -> DateTime[1];
  Second  -> DateTime[0];

 CONST
  InputLength  = 9;
  ScreenHeight = 20;
  ScreenWidth  = 152;  (* ScreenWidth MOD 8 must be 0 *)
  White        = $00;
  Black        = $0F;
  Transparent  = $10;
  Inverse      = $1F;

VAR
  InputString       : STRING[InputLength];
  KeyPressed        : BYTE;
  GlobalErrorString : ErrorStringType;

{$MIB MIB_Backup}
```

Below is a description of some of the Softwire numbers defined in the system file. By default, two of the VIGO MIB properties for the system variables are set to *NoBackup* and *InVisible*. Where alternative settings are to be used, the specific VIGO MIB property is set for particular system variables, e.g. [MIBProperties = MIB_Visible]
The VIGO MIB_Backup property is set at the end of the system file, which means that the values for the following user defined variables will be capable of being backed up.

4.1 Softwire 0, UsedSoftwireNumbers

This parameter is declared as an integer constant, with the value of 0. When the program is compiled, the Process-Pascal compiler will automatically insert the highest Softwire number used, instead of the value 0. The final Softwire List will include the variables and constants declared in the PD4000.SYS file, and all of the globally declared variables, constants and procedures in the user application.

4.2 Softwire 1, DeviceType

This constant indicates that the device is of type 4000.

4.3 Softwire 2, ProgramVersion

This constant indicates that the program version is 4.0, and the program must be compiled with a Process-Pascal compiler version 4.0. The consistency is checked by the compiler.

4.4 Softwire $03, Error3

This variable is a byte, representing information about the error status of the controller. Furthermore, the controller may be RESET by writing 255 (hex $FF) to this variable. The error codes that can be read in the variable, are defined as follows:

Code in Error3 (hex)	Meaning
$90	Master reset
$89	Reset because of watchdog
$88	Reset because of bus error
$87	Reset because of address error
$86	Reset because of illegal instruction
$85	Reset because of non-initialised interrupt
$84	Reset because of spurious interrupt
$83	Reset because of privilege violation
$82	Reset because of auto interrupt
$81	Reset because of power failure
$80	Reset because of $FF stored to Error3
$70	Error in real-time clock
$61	Error in changing task type to Softwire interrupt task
$20	Update not allowed
$19	Odd address access
$18	Pointer not initialised
$17	Procedure variable not initialised
$16	Cursor bitmap too large
$15	Cursor not initialised
$14	No more windows allowed
$13	Buffer error, a buffer is full / empty
$12	Convert error, error in converting ASCII to numeric
$11	Index error, array index out of bounds
$10	Arithmetic error, division by zero / overflow / underflow

When an error occurs, the corresponding error code is stored in *Error3*, if the error code is larger than the one already stored there. The error code is also stored in *ErrorF*. After Error3 is read from P-NET, the value in *ErrorF* is copied to *Error3*, and *ErrorF* is cleared.

When a variable in the controller is accessed from P-NET, the value of *Error3* is checked. If the value is not 0, the flag “Historical Error” in Control/Status in the P-NET response frame is set to TRUE.

4.5 Softwire $04, SerialNumber

This parameter holds the serial number of the controller. The serial number CANNOT be changed. A special function is related to the serial number, as it is possible to change the P-NET node address of the controller by writing information to this variable. This is done by storing a LONGINTEGER value here (in hexadecimal format, a longinteger occupies 8 digits), where the 6 least significant digits are the serial number of the controller, and the 2 most significant digits are the new P-NET node address.

Example: To change the P-NET node address of the controller with serial number 525614 (printed on the upper edge of the controller), to $12, the value $12525614 is written to P-NET node $7E (a global broadcast node address), at softwire $04. Although there may be a number of controllers or other P-NET devices connected to the bus receiving the request, only the controller with serial number 525614 will react and change its P-NET node address to $12.

4.6 Softwire $06, DateTime

This variable holds the value of the real time clock. In fact, the value in *DateTime* is a RAM copy of the real time clock. The RAM copy is updated once a second by the operating system.

Once a minute, the values of *DateTime* and the real time clock are compared (by the operating system). If they are different, it is assumed that the user has changed the value of *DateTime*, and the operating system will copy the value from *DateTime* to the real time clock.

At midnight, the value of the real time clock is copied to *DateTime* by the operating system, so that the task of keeping track of number of days per month and so on, is handled by the real time clock.

The real time clock is battery powered, and will be running even when the controller is not powered up. After a power up, the value from the real time clock is transferred to *DateTime*.

A complete description on the real time clock is found in the Process-Pascal manual (ref. no. 502 052).

Some indirect variables are declared at the end of the system file, which point to the *Day*, *Month, Year*, *Hour*, *Minute* and *Second*. These variables can be accessed directly from the Process-Pascal program or from VIGO.

4.7 Softwire $07, FreeRunTimer

FreeRunTimer is a timer, to which internal events are synchronised. The timer is of type Longinteger in 1/256 Second.

4.8 Softwire $0A, ModePort1

The value of this variable defines the settings for the P-NET port of the controller. To directly modify the P-NET node address or the number of masters, change the value in *ModePort1*, and call the standard procedure *InitPort1*.

The value of *ModePort1.PnetNo* can also be changed via the SerialNumber, as described under SerialNumber, Softwire $04. In this case, the procedure *InitPort1* does not have to be called to initialise the P-NET system. This is done automatically by the operating system.

Note that by changing the value of *ModePort1*, either from within the Process-Pascal program in the controller, or via P-NET, it will NOT take effect until the procedure *InitPort1* has been called, or the controller has been reset. Following a reset, the value of the variable *DefaultModePort1* is copied to *ModePort1*, before the P-NET system is initialised according to *ModePort1*. However, by OR'ing the *StartCode* (see para.4.20) with '02', the copying of *DefaultMode* is disabled, and the P-NET port is initialised according to the settings in *ModePort1*.

4.9 Softwire $0F, ErrorF

This variable holds an error code for the controller. Please refer to *Error3*, Softwire $03.

4.10 Softwire $12, InterfaceErrorBuffer

This variable is a buffer, which can consist 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:

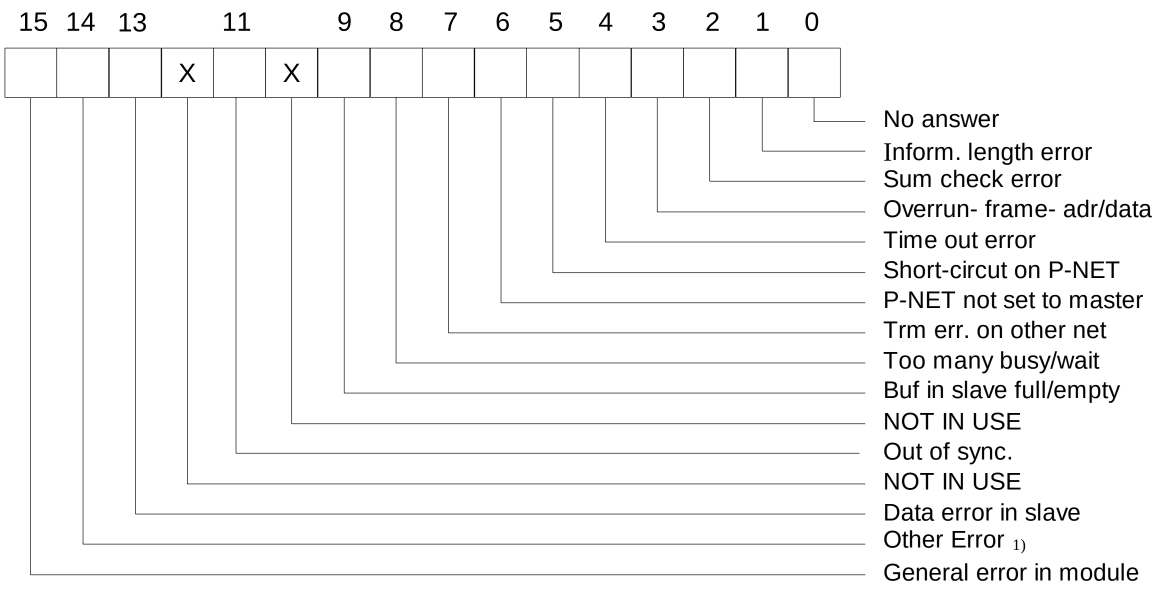
SWNo holds the Softwire number for the variable, within the declared interface module that caused the error. That is, the Softwire number of the external variable in the Softwire table in the PD 4000 controller.

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 errorcode relating to the interface error. The field is declared as a word, where each bit has the following meaning:



1) Data format error, SWNo error, Write protection, Node addr. error, Instruction error

4.11 Softwire $18, ControllerCode

This variable is a record of type *ControllerCodeRec*, consisting of the following fields:

MaxPowerDown indicates a time in seconds. If the duration of a power failure is shorter than *MaxPowerDown*, the controller program execution continues from where it was before the power failure. Otherwise the program restarts, as it would after a reset.

UpdateSign is used in conjunction with keying in new values to variables via the PD 4000 keyboard. When for instance a number key is pressed, the corresponding character is transferred to the *InputString* (refer to the program in KEY4000.INC).

When the ENTER key (“=”) is pressed, the standard Process-Pascal procedure *PerformUpdate* is called from the keyboard program. This procedure converts the contents of *InputString* to, for instance, a real value, and stores the result in the corresponding variable.

If the value of *UpdateSign* is = 0, the operating system will not automatically show the contents of the *InputString*. This must then be performed by the Process-Pascal program, which should be implemented in Key4000.INC.

If the value of *UpdateSign* is not 0, *UpdateSign* is seen as an ASCII character, and the corresponding character will be shown on the display by the operating system. *UpdateSign* is shown in the display field where the variable being updated is normally shown. The variable to UPDATE, is selected by means of the display cursor.

The function associated with *UpdateSign* is illustrated in this example. The Process-Pascal UPDATE statement is *Update(MyVar:6:2)*, and the value of *UpdateSign* is ”*”.

1: The value of *MyVar* is 123.45 and the display shows: ***123.45***
2: Now, the cursor is placed inside the displayed value of *MyVar*, and the key ”2” is pressed. This results in the following being shown on the display: ***2********
3: Now, the keys ”5” and ”6” are pressed, resulting in the following being displayed: ***256******
4: Pressing the ”=” key now will give the following result: ***256.00***

InputStr holds information for the operating system, indicating the selected InputString. *InputString* is used to update variables using the keyboard. This record element is generated by the standard procedure *SETINPUTSTRING* and should <u>not</u> be accessed directly.

CountryCode is used to select the decimal separator.

```
     CountryCode = 0;   (* decimal separator is a comma *)
     CountryCode = 1;   (* decimal separator is a point (GB) *)
```

4.12 Softwire $1A, DefaultPen

This variable is used when writing on the display. Each time a value, a line or a box is to be displayed, a Pen is one of the parameters required within the Process-Pascal statement. If, for example, the contents of the variable *MyVar* is to be displayed, the Process-Pascal statement could be *Display(MyPen, MyVar:5:1)*. However, if the parameter *MyPen* is omitted, the Process-Pascal Compiler will look for a Pen called *DefaultPen*, according to the normal ”scope” rules, as described in the Process-Pascal manual. If no local variable with the name DefaultPen is declared, the global variable *DefaultPen* at Softwire $1A is used.

A Pen must always be of type *PenInformationType*, which consists of the following fields:

CharGen contains a pointer to a character generator. A character generator is an array of bitmaps, where each bitmap represents a character. Typically, the ASCII value for the character is used as an index within the character generator.

The CharGen pointer is set up by use of the standard procedure SET-CHARACTERGENERATOR. The figure below shows an example of the character ”A” from a 6 x 8 character generator:

0	**1**	**1**	**1**	0	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
1	**1**	**1**	**1**	**1**	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
0	0	0	0	0	0

Foreground select the colour used for the TRUE (1) bits in the bitmap.

Background select the colour used for the FALSE (0) bits in the bitmap.

4 different colours may be selected for Foreground and Background: Black, White, Transparent and Inverse.

Normally, *Foreground* is set to *Black* and *Background* to *White*. This means that the pixels on the display corresponding to TRUE bits in the bitmap are turned ON (appear black), and pixels corresponding to FALSE bits are turned OFF.

If *Foreground* is set to *Transparent*, the pixels corresponding to TRUE bits in the bitmap are not changed.

If *Background* is set to *Transparent*, the pixels corresponding to FALSE bits in the bitmap are not changed from their current setting

If *Foreground* is set to *Inverse*, the pixels corresponding to TRUE bits in the bitmap, are inverted. This colour is normally used for the cursor, since this makes it possible to see the cursor, whether it is placed on a black or a white area.

If *Background* is set to *Inverse*, the pixels corresponding to FALSE bits in the bitmap, are inverted.
Foreground and *Background* can be accessed directly, or set by the standard procedure *SetColors*.

RefX, RefY, AbsX, AbsY determines the position of the PEN (NOT the cursor) on the display. The Pen position is defined relative to the upper, left corner of the display. So, the pixel in the upper, left corner has position 0,0. Refer to the Process-Pascal manual for details.

Status is a variable of type *ARRAY[0..7] of BOOLEAN*. Only one bit, bit[0] is defined. The bit is called *UseAltColor*. If this bit is TRUE, writing on the screen will use the colours *AltFore* and *AltBack*, instead of *Foreground* and *Background* from the Pen.

This is a useful facility when, for example, a variable needs to be displayed inversely, in the event of an error situation. Then, instead of asking if the error situation has arisen each time the colours of the pen are changed, bit[0] in Status is set when the error situation arises, and the *AltFore* and *AltBack* colours are set up only once.

WindowNo is set up by the standard initialisation program, and should not be altered by the user.

4.13 Softwire $1C, ScreenInfo

This variable contains information, used by the operating system, about the screen format, cursor position etc. The variable is set up by the standard initialisation program, and should not be accessed directly by the user. Please refer to the Process-Pascal manual for further details.

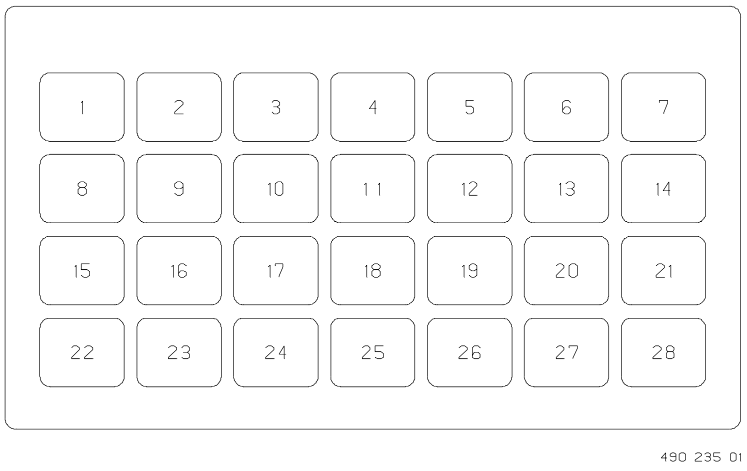
4.14 Softwire $22, ActualPowerDownTime

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 for less than one second, *ActualPowerDownTime* will be zero, but if an interrupt is connected, this interrupt will still be activated.

4.15 Softwire $24, KeyboardBuffer

This variable is a buffer of byte, where the operating system stores a key code when a key is pressed. It is also possible to achieve remote control, by storing ”key codes” in the *Key-boardBuffer* via P-NET (an example of this can be found in VIGO, by selecting the program called ‘Show PD4000 Controller’ from the right mouse menu).

The standard keyboard task, found in the file Key4000.INC, is declared as a softwire interrupt task, connected to *KeyboardBuffer*, with interrupt condition ”*InternStore*" and ”*ExternStore*". The task will run each time a key is pressed, or a ”key code” is stored in the *KeyboardBuffer* via P-NET. The key codes for the PD 4000 keyboard are shown below:



The key functions depend on the application, and are programmed in Process-Pascal. However, the key codes are fixed. The keys are numbered as shown in the figure above.

If one key is pressed, the number of that key is stored in *KeyboardBuffer* by the operating system. If the key is held down for more than 0.5 seconds, the operating system starts to send REPEAT codes with a frequency of 8 Hz. A repeat code consists of the key number + 128 ($80). If, when one key is held down, another key is also pressed, the code for the second key + 64 ($40) is stored in *KeyboardBuffer*.

Example:
Key number 4 is pressed. The code ***4*** is stored in *KeyboardBuffer*. Now the key is held down. After 0.5 seconds, the code ***132 ($84)*** is stored in *KeyboardBuffer* every 1/8 second. Now, with key number 4 still held down, key number 7 is pressed. The code ***71 ($47)*** is sent to the *KeyBoardbuffer*. If both keys are held down for 0.5 seconds, the code ***199 ($C7)*** is stored every 1/8 second. No release code is stored, when the keys are released.

4.16 Softwire $26, DefaultModePort1

This constant holds the default settings for the P-NET port. This constant may be changed in case other values for the default setting are required. Please refer to the description for *ModePort1* for further details.

4.17 Softwire $2C, NodeList

This variable is an array of elements containing node address, node type and code for external nodes. The *NodeList* array can be used to access nodes that have not been declared in the Process-Pascal program. A description of how this is accomplished can be found in the Process-Pascal manual.

The *NodeList* array can be expanded to hold more than 10 nodes, and the length of the *NodeAddr* string in *NodeListElement* can be reduced or increased. Apart from that, the user should make no other changes to the *NodeList* declaration.

4.18 Softwire $2E, CursorHide

This is a variable used to save the appearance of the part of the display now covered by the cursor. When the cursor is moved, the original uncovered display is restored from *CursorHide*.
The size of this variable should be the same as (or larger than) the number of bytes occupied by the cursor.

4.19 Softwire $32, PDBoxDefinition

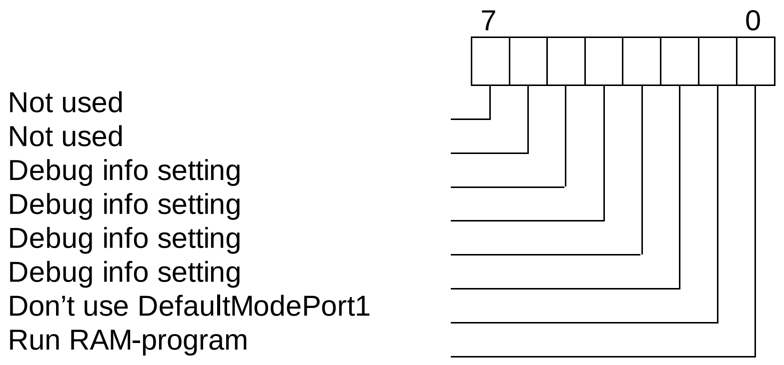
This is an array of constants, generated automatically by the compiler. The constants in the array are the Softwire numbers of all global, external declared variables in the Process-Pascal program within the controller. *PDBoxDefinition* can, for example, be used to initialise the modules within which these external variables are placed. Refer to the Process-Pascal manual or the description (in this manual) of the INITBOX4.INC file.

4.20 Screen

This variable declares the video-ram for the display. There should be no user access to this variable.

4.21 StartCode

This variable is used to define, whether the Process-Pascal program that is stored in RAM or bootPROM /FLASH should run after reset. Furthermore, *StartCode* also defines whether *DefaultModePort1* is copied to *ModePort1* after reset. *StartCode* is defined as a BYTE, where each bit in the byte has the following interpretation:



The remaining bits in *StartCode* are used to set debug information for the operating system, and should not be changed by the user.

4.22 InputString

This variable holds the ASCII characters corresponding to the digits on the keyboard. Each time a digit (or the sign or decimal point) key is pressed, the corresponding ASCII character is appended to *InputString*. This is handled by the Keyboard task.

4.23 Keypressed

This variable holds a copy of the last key code that has been read from the *Keyboardbuffer*. The key code is assigned to *Keypressed* in the Keyboard task and can be used by the user.

4.24 GlobalErrorString

This is a default declaration of a global error string that can be used together with the error handling procedure. A skeleton for an error handling procedure can be found in the example called ‘PD4000 When Error.PP’.

5 Configuration program

5.1 General Description.

The configuration utility contains all the necessary functions to perform everything needed for configuration and initialisation of the modules that have been declared in the user program. The configuration utility includes the procedures for CONFIG statements.

The configuration utility is written for the PD 4000 controller, and has the ability to configure slave modules and controllers on the P-NET.

An example, called CONFIGPP, is available in the Process-Pascal library in the Example folder.

5.2 Procedures and Functions.

The configuration program consists mainly of two procedures: One for verifying the presence and type of the modules, and one for configuring the individual channels in the modules.

To initialise the modules, you must first verify that they are in fact present. This is performed by calling the following procedure:

```
InitializeInterfaceModules;
```

This procedure verifies the type of all the declared modules. If a module doesn't answer or if it is of a wrong module type, options are displayed (see "Using the Initialisation Utility").

Next step is to initialise the individual channels in the modules, according to the CONFIG statements in the global variable declaration. The channels and variables are configured by calling the following procedure:

```
ModuleConfiguration;
```

This procedure sets up an error handler and then executes all the CONFIG statements. If errors are detected, options are displayed (see "Using the Configuration Utility"). When all channels have been skipped or configured, the number of failing channels and modules is displayed. It DOES NOT return until NewDisplay is set TRUE.

The following procedures are defined for CONFIG statements in Config4.INC:

```
SetLongInteger(VAR   CodeReg:LONGINTEGER;
                     ConfigValue:LONGINTEGER);

SetReal(VAR RealReg:REAL; ConfigValue:REAL);

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

SetByte(VAR ByteReg: BYTE; ConfigValue:BYTE);
```

```
SetBoolean(VAR BooleanReg: BOOLEAN; ConfigValue:BOOLEAN);

SetBit8(VAR Bit8Reg:Bit8; Bit8Byte:BYTE);

PT100(VAR TempChannel:ChAnalogIn);          {old modules}

Standard_PT100(VAR TempChannel:AnalogInCh); {new modules}

DigitalInput(VAR InChannel:ChDigitalIO);

DigitalOutput(VAR OutChannel:ChDigitalIO);

OutputWithFeedBack(VAR OutputChannel:ChDigitalIO;
                   ChannelNumberA, ChannelNumberB:BYTE;
                   PresetTime :REAL);
```

Since the format of the analogue channel has changed, there are two procedures for configuring a channel for PT100 input: The PT100-procedure is used for PD1611, PD1651, and PD1652. Standard_PT100 is used for modules following the standardised general-purpose channel types, like the PD3221 UPI, PD3240 and PD3250.

5.3 Global Variables

A number of global variables and constants are declared and used by the configuration utility. Some of these variables are declared in the main program, such as

```
VAR NewDisplay : BOOLEAN;
```

Setting this variable to TRUE will force an immediate exit from the configuration procedure. In the demo program this is achieved by pressing function key number 24 (dec). NewDisplay must be cleared before calling the configuration procedures.

The following constant is declared in InitBox4.INC:

```
ModuleCountBitSize = [integer value];
```

This constant defines the number of entries in a list that is used when skipping the initialisation or configuration of modules. The value of this constant must be greater than, or equal to, the number of modules defined in the program. The configuration utility program itself uses 6 definitions, so the value must be AT LEAST 6 larger than the number of module definitions you make. The default value is 512 and can be used in most cases.

The following variable is declared in InitBox4.INC:

```
ModuleSkipBitList: ARRAY[1..ModuleCountBitSize] OF BOOLEAN;
```

This list is an array that contains a flag for each of the modules declared in your program. As each module (found in PDBOXDEFINITION) is checked for type and presence, the corresponding flag is set, to indicate the state of the module.

This is used during the channel configuration, to avoid configuring channels in modules of wrong type or missing modules.

If a flag is set FALSE, the corresponding module was found, and the error handler will flag errors concerning this module.

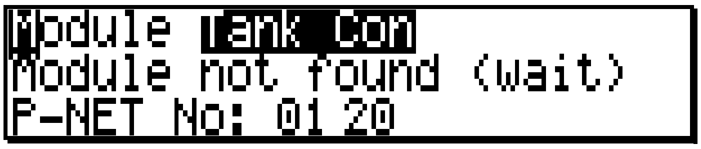
If a flag is set TRUE, the module was abandoned during the first test, and the error handler (named *ErrorCorrection*) will ignore all errors on this module. All configuration subroutines are testing this flag using the function *SkippedModule*.

As each entry in this array is 1 bit in size, the default value takes 64 bytes of RAM, so unless there are more than 506 modules, there is no reason to change the value of *ModuleCountBitSize*. If more modules are defined than this list can hold, an error is generated when configuration is attempted.

Note that the procedure *ModuleConfiguration* requires the information in this array, so the procedure *InitializeInterfaceModules* MUST be called first.

5.4 Using the Initialisation Utility

When the initialisation utility is run, the presence and type of all modules are verified. If a module does not respond, a display like this is shown

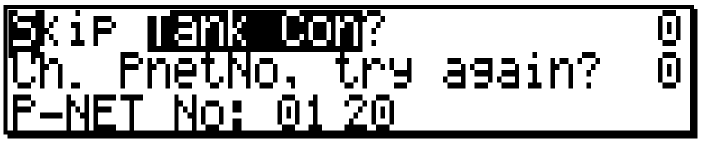


```
Module Tank Con
Module not found (wait)
P-NET No: 01 20
```

indicating that the module NAMED "Tank Con" did not respond. A new module does not respond until its node address has been configured.

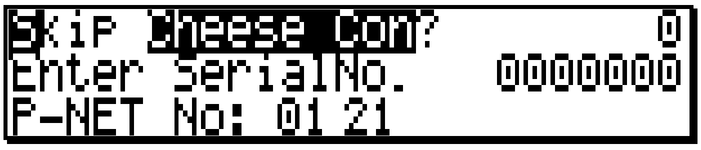
The program waits 5 seconds. Then a new display is shown, depending on the type of module:

Display for PD modules in the 1000 series (old modules):



```
Skip Tank Con?            0
Ch. PnetNo, try again?  0
P-NET No: 01 20
```

Display for modules using standardised general-purpose channels and PD340C modules:

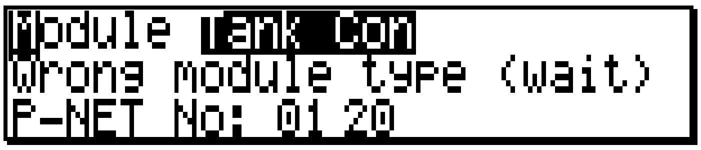


```
Skip Cheese Con?       0
Enter SerialNo.  0000000
P-NET No: 01 21
```

If skip is chosen, (by entering 1 in the field in the right side), the module will be marked 'missing' and will not be configured.

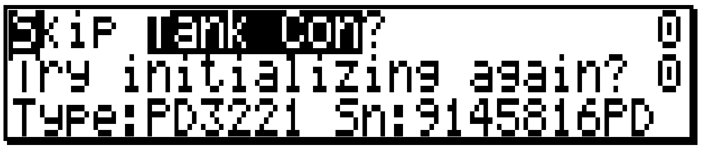
The method of retrying depends on the type of module: If the module is from the PD1000 series, check that the node switches on the motherboard are set correctly, then try again. Otherwise, key in the SerialNumber of the module. The controller will then try to configure the node address of the corresponding module.

If the module is found, but the type is wrong, another display is shown:



```
Module Tank Con
Wrong module type (wait)
P-NET No: 01 20
```

The program waits 5 seconds, and will then show the following:



```
Skip Tank Con?         0
Try initializing again? 0
Type:PD3221 Sn:9145816PD
```

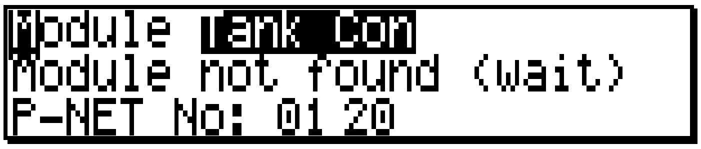
PD3221 was the type of the module actually found where it expected the module NAMED "Tank Con". Before the correct module can be configured, the wrong module must be disconnected. Find it by looking at the module type and serial number. Then disconnect the module.

On modules from the PD1000 series it is not possible to read the SerialNumber via P-NET, and no SerialNumber is displayed. As the node address on these modules is configured with switches, look for modules with the wrong P-NET node address.

It should then be possible to configure the correct module.

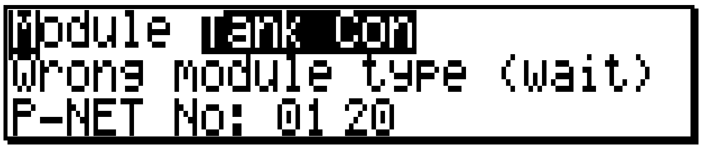
5.5 Error messages

Possible error messages are:



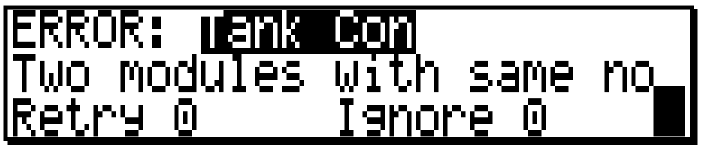
```
Module Tank Con
Module not found (wait)
P-NET No: 01 20
```

"Module not found". There was no answer from the module.



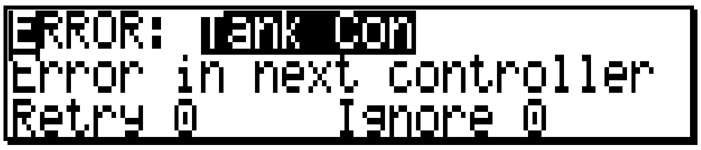
```
Module Tank Con
Wrong module type (wait)
P-NET No: 01 20
```

"Wrong module type". The type of the module is different from that specified in the program.



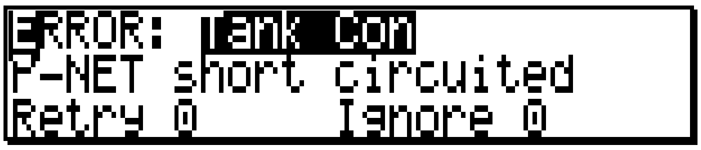
```
ERROR: Tank Con
Two modules with same no
Retry 0     Ignore 0
```

"Two modules with same node address". Two or more modules are sharing the same node address, therefore causing data collisions on the bus. It is not possible to find out which modules are involved. Start disconnecting modules until this error disappears. When the initialisation program asks for the modules, reconnect them one by one and retry.



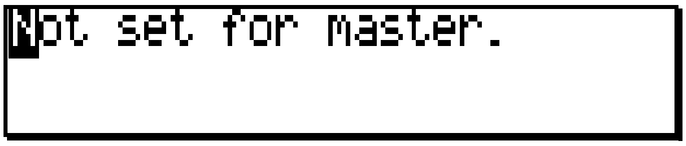
```
ERROR: Tank Con
Error in next controller
Retry 0     Ignore 0
```

"Error in next controller". The next controller (with the succeeding node address) failed the token passing.



```
ERROR: Tank Con
P-NET short circuited
Retry 0     Ignore 0
```

"P-NET short circuited". The P-NET system is exhibiting a short circuit.



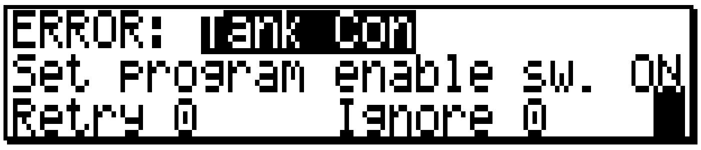
```
Not set for master.
```

"Not set for master". The node address of the controller is higher than the maximum number of masters expected on the net. The controller will never gain access to the net. The node address and the number of masters must be correctly configured. This error does not allow a retry. You can abort the initialization by setting NewDisplay TRUE (function key 24).

5.6 Using the Configuration Utility

After all modules have been initialised, the configuration procedure can be run. Each channel with a CONFIG statement is checked. If the channel configuration differs from the configuration value, the channel configuration is modified and checked again. If the configuration program is unable to modify the settings, an error is received:

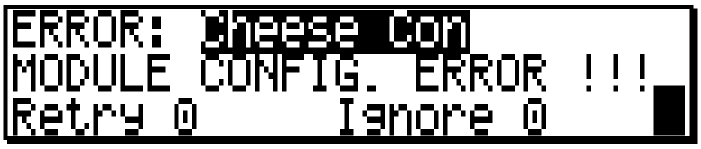
Display for PD1000 series and PD340C modules:



```
ERROR: Tank Con
Set program enable sw. ON
Retry 0     Ignore 0
```

PD1000 series and PD340C modules are write protected using a switch on the motherboard. This switch must be in the ON position to configure the module. Afterwards it must be set OFF to protect the configuration.

Display for modules with standard channels:



```
ERROR: Cheese Con
MODULE CONFIG. ERROR !!!
Retry 0     Ignore 0
```

In standard channel modules, the write protection switch is internal and controlled via the P-NET. The configuration program will automatically handle the internal write protect switch.

If the module continues to fail after setting the switch ON, there is either an invalid value specified in the Process-Pascal program, or there may be a hardware error in the module.

In addition, similar errors to those described in the initialisation may be seen. This may occur if, for some reason, the module fails during configuration.

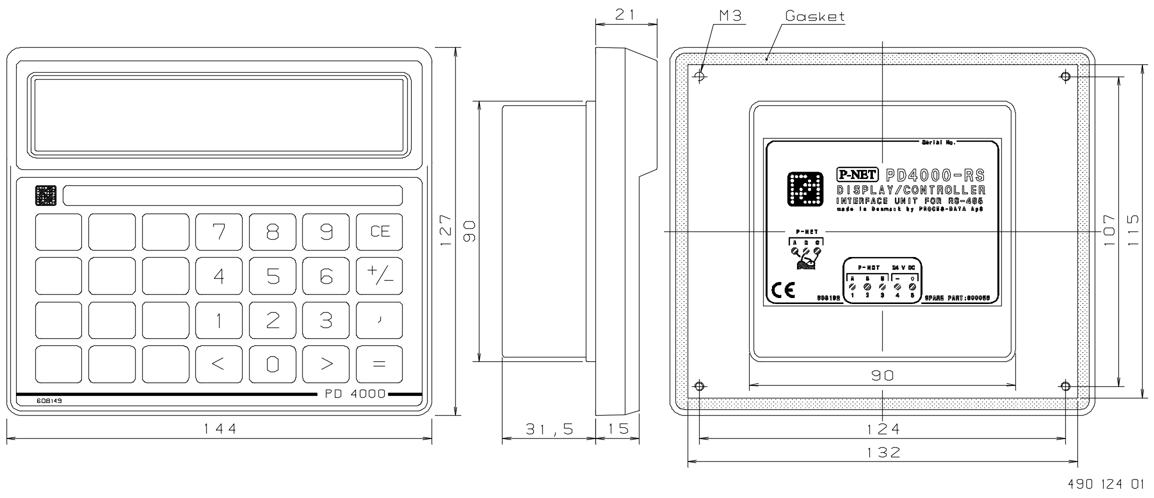
6 Construction, Mechanical

The Controller is housed in a black injection moulded plastic enclosure. The entire Controller is completely filled with silicone. This construction makes it extremely resistant to water, dust and vibration.

The controller is made up of 2 parts, the main part with built-in electronics, display and keyboard, and the power supply part, with power supply, battery for the CMOS RAM, and P-NET interface circuitry. The power supply part is attached to the main part by 4 screws.

Weight:	0.7 kg
Sealing:	IP68 @ front panel mounting
Enclosure:	Black NORYL GFN

Scale drawing (in mm):



7 Specifications

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

All specifications apply within the approved EMI conditions.

7.1 Power supply.

Power supply DC:	nom 24.0 V
	min 20.0 V
	max 28.0 V
Ripple:	max 5 %
Power consumption:	max 2.0 W
Necessary power-up current:	max 400 mA

7.2 Program storage.

RAM Memory size:	512 K bytes
FLASH Memory size (controllers with serial no. > 635600):	256 K bytes

7.3 Display.

Resolution:	150 by 20 pixels
Viewing area:	120mm * 19.2mm

7.4 Keyboard.

Membrane click-switch foil with metal domes:	28 keys

7.5 Ambient Temperature.

Operating temperature:	-25 °C to +70 °C
Storage temperature:	-40 °C to +85 °C

7.6 Approvals.

Compliance with EMC-directive no.:	89/336/EEC
Generic standards for emission:	
Residential, commercial and light industry	EN 50081-1
Industry	EN 50081-2
Generic standards for immunity:	
Residential, commercial and light industry	EN 50082-1
Industry	EN 50082-2
Vibration (sinusoidal):	IEC 68-2-6 Test Fc