1st International conference
on
P-NET fieldbus system
31st October - 1st November 1991
Scandic Hotel, Silkeborg, Denmark

PROCESS-PASCAL

a high level programming language integrated with P-NET

By John Johansen, Application engineer, Proces-Data Silkeborg ApS, Denmark.

INTRODUCTION TO PROCESS-PASCAL.

PROCESS-PASCAL is a high level programming language based on Standard Pascal.

PROCESS-PASCAL is extended from Standard Pascal with a number of facilities, which makes it possible to execute several programmes simultaneously in one computer. This is called multi-tasking.

PROCESS-PASCAL is specially developed for use in connection with P-NET, which is a local area network for transmission of data in distributed data acquisition and process control plants. Data, which are distributed in modules on the P-NET, can be defined as variables in PROCESS-PASCAL.

PROCESS-PASCAL permits automatic reports of alarms in case of error in the controller or in the interface modules. Furthermore, it is possible to automatically test all components of the plant during the starting phase.

PROCESS-PASCAL includes standard routines for interactive screen dialogue. Thus it is possible to define that a variable must be shown on the screen and be continuously updated. Data can be keyed into a variable by pointing at it using the screen cursor.

PROCESS-PASCAL programmes can be written using any general purpose editor working with ASCII files.

PROCESS-PASCAL programmes operate with several types of variables, and typecasting is automatically performed by the compiler during compilation. This makes programmes more safe and easier to develop.

The PROCESS-PASCAL compiler is a cross compiler running on a PC under MSDOS. The compiler exists in two versions:

PPCOMPIL.EXE and PPCEMS.EXE

PPCOMPIL.EXE is the compiler used for small and medium size programmes. It requires a minimum of 450 Kbyte of free memory in the PC for program execution and data allocation.

PPCEMS.EXE is an EMS (Expanded Memory Specification) version of the compiler and it is mainly used for large programmes. It requires a minimum of 400 Kbyte in the conventional memory area for program execution and basic data allocation. Additional data are allocated in the EMS area. The EMS driver must be version 4.0 or later.

The compiler generates code, P-code, that is stored in the PD3000 controller in EPROM. The operating system in the controller interprets the P-code and executes a piece of machine code for each P-code. PROCESS-PASCAL programmes can not be executed on a PC. The compiler is entirely developed by Proces-Data.

P-NET, SOFT-WIRING and **PROCESS-PASCAL** are registered trademarks of Proces-Data Silkeborg Aps.

COMPARING PROCESS-PASCAL VER. 2.0 TO ISO 7185 STANDARD PASCAL.

This list compares PROCESS-PASCAL to ISO 7185 STANDARD PASCAL as defined in the book **PASCAL USER MANUAL AND REPORT** THIRD EDITION by Kathleen Jensen and Niklaus Wirth (published by Springer-Verlag).

Exceptions to ISO 7185 STANDARD PASCAL.

In ISO 7185 STANDARD PASCAL, an identifier can be of any length and all characters are significant. In PROCESS-PASCAL, an identifier can be of any length, but only the first 30 characters are significant.

In ISO 7185 STANDARD PASCAL, a comment can begin with { and end with *), or begin with (* and end with }. In PROCESS-PASCAL, comments must begin and end with the same set of symbols.

In ISO 7185 STANDARD PASCAL, it is an error if the value of the selector in a CASE statement is not equal to any of the case constants. In PROCESS-PASCAL, this is not an error; instead the CASE statement is ignored unless it contains an ELSE clause.

In ISO 7185 STANDARD PASCAL, statements that threaten the control variable of a FOR statement are not allowed. In PROCESS-PASCAL, this requirement is not enforced.

ISO 7185 STANDARD PASCAL can operate on files. It is not possible to operate on files in PROCESS-PASCAL and in that reason the following procedures are not implemented:

Pack Unpack
Read Readln
Write Writeln
Eof(f) Eoln(f)
Get(f) Put(f)
Reset(f) Rewrite(f)
Page(f)

ISO 7185 STANDARD PASCAL can operate with pointers. It is not possible to use dynamic pointers in PROCESS-PASCAL and in that reason the following procedures are not implemented:

Dispose(q) New(p)

ISO 7185 STANDARD PASCAL can operate with recursive procedures and functions. It is not possible to use recursivity in PROCESS-PASCAL.

In ISO 7185 STANDARD PASCAL, some arithmetic functions are available. The following functions are not available in PROCESS-PASCAL:

Arctan(x) Exp(x)
Ln(x) Sin(x)
Sqr(x) Sqrt(x)

These functions can be written in PROCESS-PASCAL by using series. A number of these functions can be found in a file called MATH.INC.

In ISO 7185 STANDARD PASCAL, the WITH statement can be used. This statement is not implemented in PROCESS-PASCAL.
Conformant array schemes are not supported by PROCESS-PASCAL.

Extensions to ISO 7185 STANDARD PASCAL.

PROCESS-PASCAL is integrated with P-NET, a local area network, which allows use of distributed data.

PROCESS-PASCAL is specially designed for multitasking.

PROCESS-PASCAL implements the additional integer types LONGINTEGER, BYTE and WORD and the additional real type LONGREAL.

PROCESS-PASCAL implements the additional type TIMER, which is assign compatible with the type REAL. A variable of type TIMER will count down in real time when assigned a value.

PROCESS-PASCAL implements the additional type BUFFER, which, like an ARRAY type, has a fixed number of components of one type. A BUFFER is accessed only by the buffers identifier without any indexes.

PROCESS-PASCAL implements the additional types VIDEOBITMAP, LARGEBITMAP and SMALLBITMAP.

PROCESS-PASCAL implements string types, which differ from the packed string types defined by ISO 7185 STANDARD PASCAL in that they include a dynamic-length attribute that can vary during execution.

String constants are compatible with the PROCESS-PASCAL string types, and can contain control characters and other nonprintable characters.

String-type variables can be indexed as arrays to access individual characters in a string.

The relational operators can be used to compare strings.

PROCESS-PASCAL implements typed constants, which can be used to declare initialized variables of all types.

Variables can be declared at absolute memory addresses using an AT ADDRESS clause.

Constant, type, variable, procedure and function declarations can occur any number of times in any order in a block.

An identifier can contain underscore characters (_) after the first character.

Integer constants can be written in hexadecimal notation; such constants are prefixed by a $.

The type of an expression can be changed to another type through a value typecast.

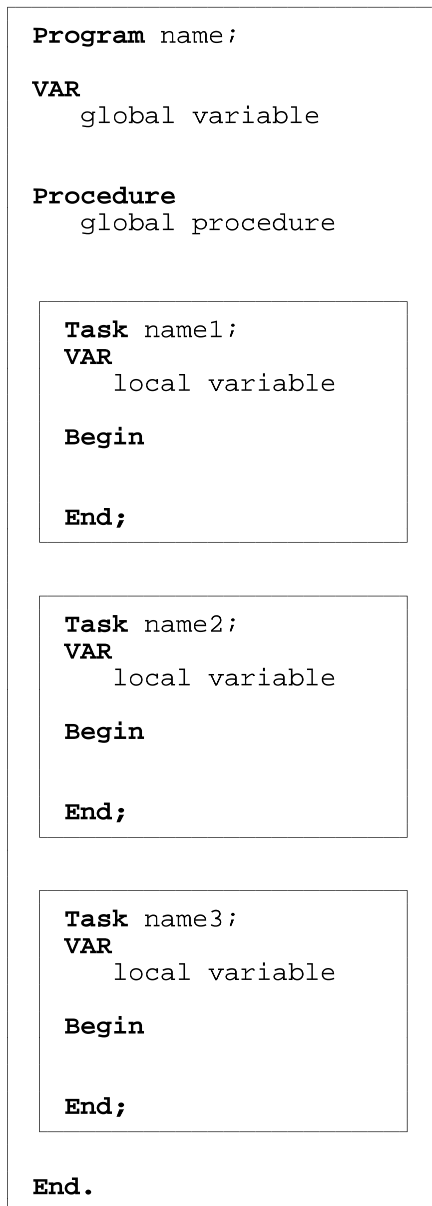
The CASE statement allows constant ranges in CASE label lists, and provides an optional ELSE part.

PROCESS-PASCAL implements the following standard procedures and functions, which are not found in ISO 7185 STANDARD PASCAL:

AlarmHornOnOff
AlarmPulseOn
BitTest
Box
BoxTo
BufferEmpty
BufferFull
ChangeTask
Clear
ClearWindow
CloseWindow
ContinueTask
ContrastControl
Convert
CursorInWindow
CursorTo
CursorToAbs
CursorWithin
CyclicTask
Disable
Display
DisplayOnOff
Enable
InitBuffer
InitPort1
InitPort2
InterruptTask
LedOnOff
LightControl
LightOnOff
Line
LineTo
MaxRunTime
MoveCursor
MovePen
MySWNo
MyTaskNo
OpenWindow
PCodeCall
PenRefTo
PenTo
PenToAbs
PerformUpdate
PointerOk
PointerToNode
Raise
RestartTask
Return
SetCharacterGenerator
SetColors
SetCursor
SetCursorColors
SetCursorType
SetInputString
SetScreen
SetVideo
SetWindow
SetWindowFrame
StopTask
StrVal
SystemCall
Tab
TestAndSet
TimedInterruptTime
TimedTask
Update
Val
Varname
ZoomIn
ZoomInHor
ZoomOut
ZoomOutHor
ZoomOutVer

PROGRAM STRUCTURE IN PROCESS-PASCAL.

Every PROCESS-PASCAL program consists of a heading and a block. The structure is illustrated below:



Program Heading.

Program heading consists of the word **Program** and a name.

Global variable declaration.

External variables in other modules accessed via the P-Net are declared with an identifier which is used within the program.

Internal variables used to exchange data between tasks, as well internal as external in other controllers.

Global procedure declaration.

Global procedures can be called from all internal tasks.
A global procedure can be called simultaneously from several tasks with different sets of parameters.

Task declaration.

Program declaration for task. Tasks are executed 'simultaneously'.

Tasks are used to monitor and control different jobs that occur simultaneously. This is done by defining each job independently in a separate task.

Data are exchanged with other tasks and 'the world outside' in global variables.

A PROCESS-PASCAL program is divided into a heading and a body, called a block. The heading gives the program name. The block consists of seven sections: LABEL-, CONST-, TYPE-, VAR-, PROCEDURES and FUNCTIONS- and TASK declaration, where any except the last may be empty.

All that is defined before tasks is called the global section and you can have as many declaration sections as you want, in any order you want, including procedures and functions declarations. But, as in standard Pascal, things must be defined before they are used otherwise, a compile-time error will occur.

Task, procedure and function declarations have a structure similar to a program; i.e. consists of a heading and a block. The symbols in the heading are different (TASK, PROCEDURE, FUNCTION instead of PROGRAM) and they end with a semicolon instead of a period. They can have their own constants, data types, and variables, even their own procedures and functions.

Tasks are different from procedures and functions at various points:

1. TASKs have their own memory area allocated for variables defined in a VAR section in the block. Termination of a task does NOT release this allocated storage.
2. TASKs have their own program counter and stack pointer and operates entirely autonomously from other tasks.
3. TASKs can not be nested.
4. TASKs are not called from a statement to execute.

TASK, AN INTRODUCTION.

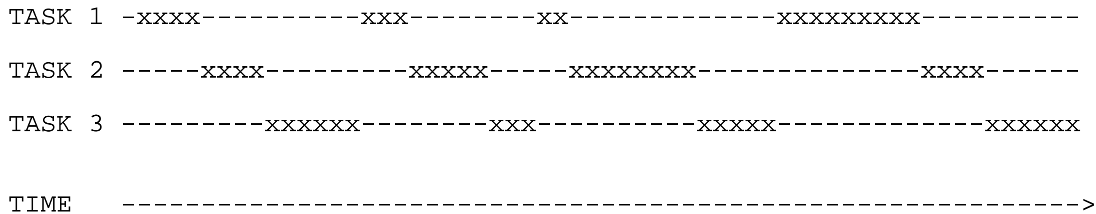
Multitasking is a facility in PROCESS-PASCAL which makes it possible to execute several sub-programmes simultaneously in the very same computer. These sub-programmes are called TASKs and are fundamental to PROCESS-PASCAL. They make it very easy to split up a program into manageable proportions where each TASK performs a distinct function.

Multitasking is very useful for process control where the process can be controlled in real time.

A TASK is a section of code which controls a part of the process, e.g. monitoring the keyboard for user input or controlling the valves on a blending unit etc. Each TASK will run and perform as much of its function as it wants to before it relinquishes control of the processor and lets another TASK run. While in reality the TASKs are not performed in parallel, the switching between them is done fast enough to make this a useful aid visualising a system in real time. Switching from one TASK to another can be done in all parts of the program, including procedures, but can advantageously be used each time a delay appears or the TASK is waiting for some actions to take place, e.g. a certain level on an input signal or a TIMER to run out. Switching to another TASK in such situations makes the program more efficient, and no time is wasted by waiting.

Swapping from one TASK to another is done by the statement CHANGETASK, which is a standard procedure in PROCESS-PASCAL. The actual TASK calling CHANGETASK stops program execution in the TASK and relinquishes control of the processor to the following TASK in which the program execution continues from where it was last interrupted (e.g. by CHANGETASK).

The principle diagram below shows how the program execution is switching between a number of cyclic TASKs



TASK-TYPES.

PROCESS-PASCAL handles 3 different types of TASKs: CYCLIC TASK, TIMEDINTERRUPT TASK and SOFTWIREINTERRUPT TASK. All 3 types of TASK can be used within the same program.

Cyclic TASKs are executed in sequence, where CHANGETASK switches to the following one in the sequence. The sequence is defined by the order of the TASKs in the program.

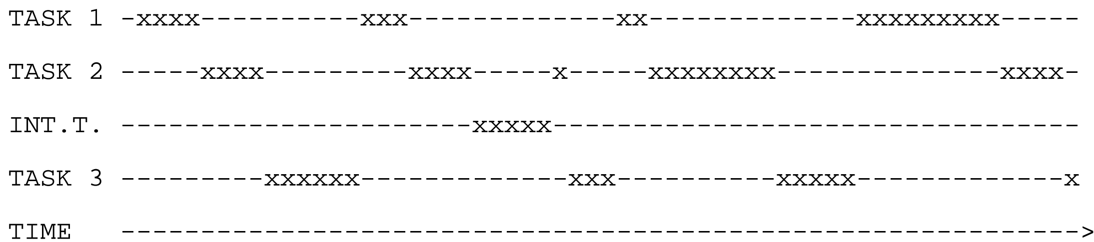
TIMEDINTERRUPT TASKs are executed at certain time intervals, controlled by the programmer. The time periods are declared in seconds and the resolution is 1/128 second.

SOFTWIREINTERRUPT TASKs are executed each time a certain defined event occurs, eg. the keyboard is activated and a TASK starts running to read which key was pressed and to undertake the appropriate action.

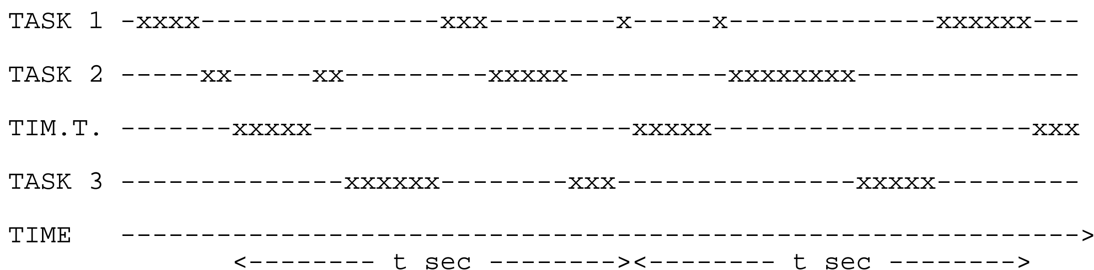
When a cyclic TASK is running and a timedinterrupt or softwireinterrupt TASK is ready to run, a CHANGETASK is forced in the cyclic TASK and control is given to the interrupting TASK. When the interrupting TASK has finished, i.e. reaches a CHANGETASK statement, this CHANGETASK makes the earlier cyclic TASK continue where it was interrupted.

A timedinterrupt or softwireinterrupt TASK can not be interrupted by other TASKs.

The principle diagram below shows how the program execution is switching when a SOFTWIREINTERRUPT TASK interrupts a number of cyclic TASKs



The principle diagram below shows how the program execution is switching when a TIMEDINTERRUPT TASK is interrupting a number of cyclic TASKs. The TIMEDINTERRUPT TASK is timed to t seconds.



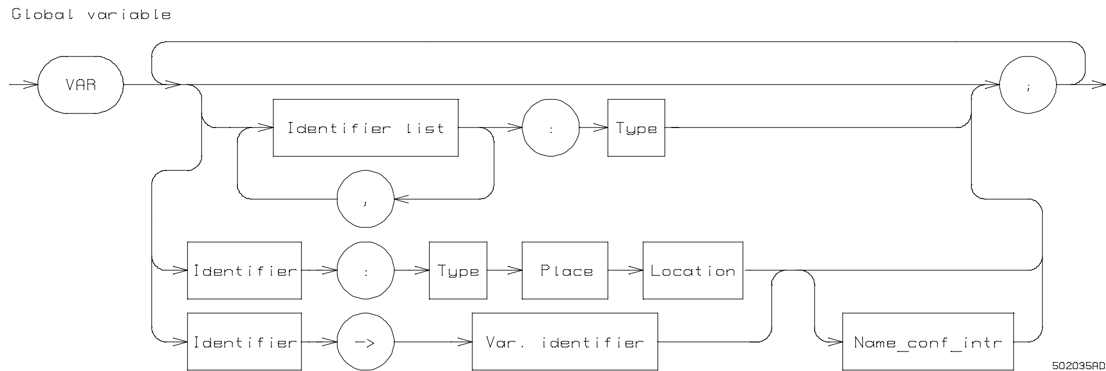
VARIABLE DECLARATION

Variables can be declared to reside inside the controller or externally in other modules at a net-address. Variables can be allocated by the compiler, or they can be declared to reside at specific memory addresses for special applications.

Variables declared before tasks, and outside procedures and functions are called **global variables** and reside in a global data section. Variables declared within a task, but outside procedures and functions are called **local variables** and reside in a local data section for the specific task. Variables declared within procedures and functions are also called local variables, but these variables are only known within the procedure or function in which they are defined.

All the global identifiers used in a PROCESS-PASCAL program are converted to a number by the compiler. These softwire numbers are used as an entry key to the softwire list which contains structured information on each individual global variable and constant, used in the particular program.

Variables of the same type can be declared by a list of identifiers, separated by a comma, followed by a colon and the type of the variables.



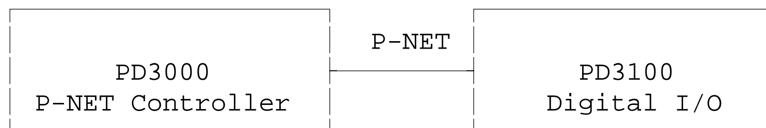
Examples of variable declarations:

```
VAR
     LineNo, PageNo : INTEGER;              (* These variables *)
     Color : BYTE;                          (* are allocated   *)
     Process_On, AlarmState : BOOLEAN;      (* by the compiler *)
     Wait, LightTime : TIMER;
     Limit : REAL;
```

Variables can be defined to reside at a specific address in memory, at a specific softwire number or at a net-address. If a variable is declared to reside at a softwire number or at a net-address, memory has already been allocated for it.

HOW TO DECLARE AN INTERFACE MODULE IN PROCESS-PASCAL.

Below is shown an example of a simple system consisting of a PD3000 P-Net Controller and a PD3100 digital I/O module.



To get access to the digital module from the controller, the digital module must be declared in the program in the controller. This is done in a normal variable declaration as follows:

```
VAR
    DigModule : PD3100 AT NET: (2, $31);
```

This single line declares the entire digital module in the program and makes it possible to get access to test/set/reset all inputs/outputs in that module.

DigModule, identifier:
  is the PROCESS-PASCAL identifier and can be used as any other identifier in the program.

PD3100, type:
  is the type of the variable and is a predeclared type specifying the construction of the digital module, the organisation of the I/O channels.

AT NET:, net address:
  specifies that the variable (DigModule) is an external variable, a variable that must be accessed via P-NET.
  The following parameters **(2, $31)** specifies where the digital module is located, the path to the module seen from the controller.
  The first parameter (2) indicates the communication port number, the PD3000 P-NET Controller utilises two communication ports, port 1 and port 2.
  The next parameter ($31) defines that the digital module has P-NET number 31 HEX.

If the module is located at another net, the net address is simply extended with additional numbers, first port number then P-NET number for the module to continue from, port number from that module and so on, and at last, P-NET number for the digital module.

The above declaration shows how to declare an entire module. But when writing a program it can be more convenient with a more detailed specification of inputs and outputs. To do this, PROCESS-PASCAL offers the facility of declaring indirect variables as a part of another variable or as a combination of other variables.

If we take the above declaration for the digital module:

DigModule : PD3100 AT NET: (2, $31);

a number of indirect variables can be declared from that variable. Example:

MotorChannel -> DigModule.CH21;

The variable **MotorChannel** is now declared as an indirect variable, the entire channel 21 of the digital module.

The indirect declaration can go even further than a channel. Example:

Motor -> DigModule.CH21.FlagReg[7];

Now the variable **Motor** is declared as a part of channel 21 of the digital module, indicated by the 7th index of the register FlagReg, which is a boolean array. The bit 7 is identical to the output transistor at the channel.

By using this type of variable declaration, the variable **Motor** becomes identical to output 21 at the digital module. In the program the output can be switched on and off in a simple way as shown below:

```
Motor:=ON;       (* switch the output on *)

Motor:=OFF;      (* switch the output off *)
```

The above statements has the same effect as if the entire module was used with channel and register selection as shown below:

```
MotorChannel.FlagReg[7]:=ON;           (* switch the output on *)
MotorChannel.FlagReg[7]:=OFF;          (* switch the output off *)

DigModule.CH21.FlagReg[7]:=ON;     (* switch the output on *)
DigModule.CH21.FlagReg[7]:=OFF;    (* switch the output off *)
```

The digital interface module is organised as a number of channels, where a channel consists of the output transistor / input sensor and a number of related register for each channel. As examples of these related registers can be mentioned: Error register, counter register and code register.

A variable can be declared with a **Name**, which declares a name as a stringconstant for the variable. This name can be used as a string when an error occurs for the variable.

Examples of variable declarations, using a name:

```
VAR
     DigModule : PD3100 AT NET: ( 1,$35)
                    NAME : 'Digital module panel 1';
     AnaModule : PD1611 AT NET: ( 1,$38)
                    NAME : 'Analog controlunit 22';
```

When using NAME on variables of interface type (modules) conforming to the Interface Declaration in the P-NET standard, each channel can get it's own name. NAME for the module belongs to channel 0, the service channel. When using NAME on other variables, each variable can get only one name.

A **CONFIG** clause can be used to assign a value to the variable when calling a CONFIG statement in the program. This is done by typing **CONFIG: Procedureidentifier** after the type and name for the variable. The procedure will be executed when a CONFIG statement is executed.

Examples of variable declarations, using CONFIG:

```
VAR
DigModule : PD3100 AT NET: ( 1,$35) NAME : 'Module at CIP unit'
               CONFIG : SetLongInteger(.Ch0.Code9, WatchDog);

AnaModule : PD1611 AT NET: ( 1,$38) NAME : 'Inlet control unit'
               CONFIG : SetLongInteger(.Ch0.Code9, WatchDog);
```

The procedure call passes the variable itself as a default parameter. When the variable is of complex type, a part of the variable can be selected as the first parameter. When the variable is an entire module, a channel or even a register can be selected to be the parameter, see the example above.

Indirect arrays can be used to assemble particular variables, or part of variables, in a structured manner. This can be used to make easyer and more understandable programmes.

Example of an indirect array of digital IO channels:

```
VAR
     DigModule1 : PD3100 AT NET (2,$51);
     DigModule2 : PD3100 AT NET (2,$52);


     Valves -> ARRAY[1..MaxNumberOfValves] OF ChDigitalIO =
               ([1] -> DigModule1.Ch31,
                [2] -> DigModule1.Ch32,
                [3] -> DigModule1.Ch33,
                [4] -> DigModule1.Ch34,
                [5] -> DigModule2.Ch31,
                [6] -> DigModule2.Ch32,
                [7] -> DigModule2.Ch33,
                [8] -> DigModule2.Ch34);
```

To access an IO channel in either of the two digital modules, i.e. a valve, an indirect element in the variable

VALVES is accessed:

```
Valves[ValveNumber].FlagReg[7]:=ON;
IF Valves[3].Counter <= 20 THEN
```

Examples of indirect variable declarations using the NAME clause:

```
VAR
     Start -> DigModule.Ch21 NAME :'Start button for production';
     WaterTemp -> AnaModule.Ch1.AnalogIn NAME :'Water temperature';
```

The name 'Start button for production' is connected to the variable Start, which means that the name can be used as a string when an error occurs in accessing channel 21 in DigModule. See the WHEN ERROR chapter how to use NAME.

The CONFIG clause can also be used on indirect variables.

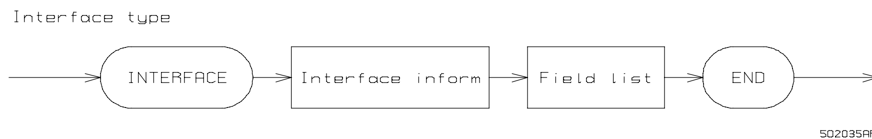
Examples of indirect variable declarations using the NAME and CONFIG clause:

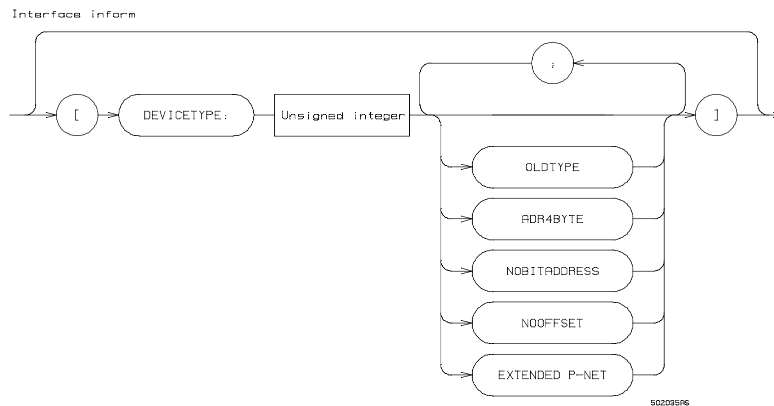
```
VAR
     Start -> DigModule.Ch21 NAME :'Start button for production'
                              CONFIG: DigitalInput;
     WaterTemp -> AnaModule.Ch1 NAME :'Water temperature'
                              CONFIG: Pt100;
```

INTERFACE

An interface type is used to define an interfacemodule or a channel in an interfacemodule as a whole structured variable. An interfacemodule is constructed with a number of channels, where each channel has 16 accessible registers. The channels can be of the same type or of different types, depending on the specific interfacemodule.

An interface type has a fixed number of components, that can be of different types. An interface type defines a channel, if all the components in the type declaration is of simple type. An interface type defines an interfacemodule, if all the components in the type declaration is of interface type or the type 'Unused'. The first component in the definition of a channel, defines register 0, the second component defines register 1 and so on. The first component in the definition of an interfacemodule, defines channel 0, the second component defines channel 1 and so on.





The interface inform DEVICETYPE is followed by a constant that denotes the moduletype. DEVICETYPE must be declared.

The interface inform OLDTYPE denotes that the module is of an old type, which means that the variables of real type are stored in a different format. Conversion to the IEEE format is done by the operating system in the controller during program execution and the user will not need to take any action for it.

The interface inform ADR4BYTE denotes the length of the SoftWireNo / abs. address when accessing the module. The length of the address can be 4 byte or 2 byte, denoted by Adr4Byte or Adr2Byte, where Adr2Byte is default.

The interface inform NOBITADDRESS denotes that the module is not able to understand bit addressing.

The interface inform NOOFFSET denotes that the module is accessed with an address without any offset.

The interface inform EXTENDED PNET denotes that the module understands complex/extended Pnet address, e.g a controller.

Example of an interface type:

```
PD1611 =INTERFACE[DeviceType:1611, NoOffset, NoBitAddress, OldType]
           ch0: Service;
           ch1: ChAnalogIn;
           ch2: ChAnalogIn;
           ch3: ChAnalogIn;
           ch4: ChAnalogIn;
           ch5: ChAnalogIn;
           ch6: ChAnalogIn;
           ch7: ChAnalogOUT;
           ch8: ChPID;
           ch9: ChBatch;
           chA: ChBatch;
           chB: ChDisplay;
         END;
```

WHEN ERROR

Some build in facilities in PROCESS-PASCAL gives you the possibility for using data distributed on the P-NET fieldbus system. To ensure your program against any erroneous data, as well internal as external in the system, PROCESS-PASCAL offers an automatic error detecting system.

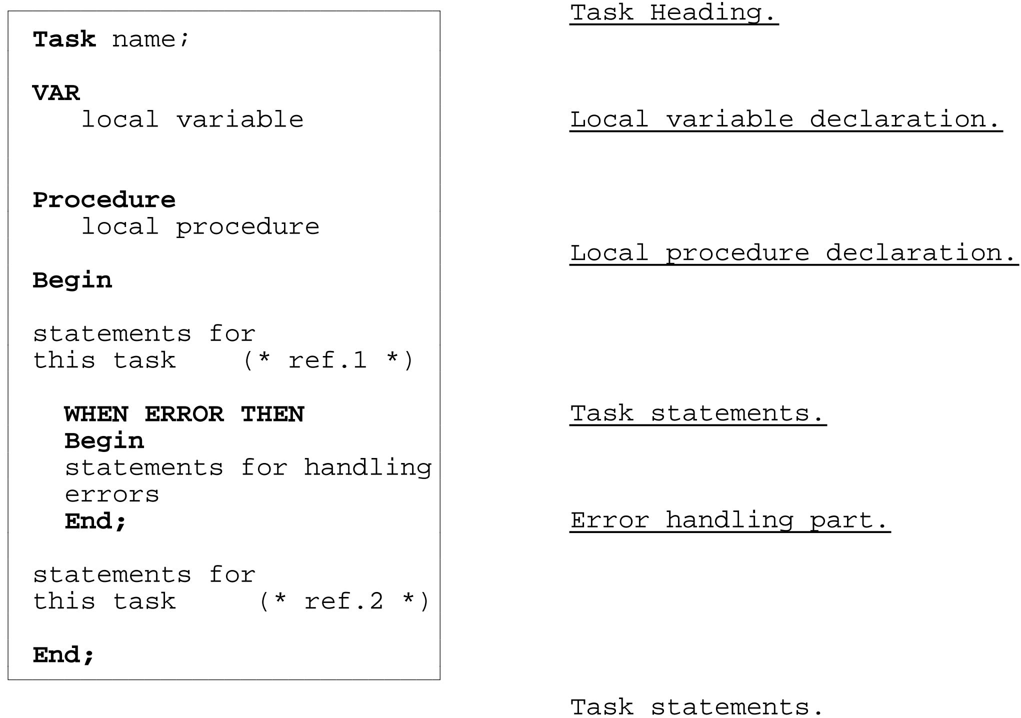
When using the P-NET to communicate with interface modules or other controllers, errors can occur. The possible errors that can appear are called INTERFACE ERRORS and can be transmission errors or errors in the interface modules. An error in a module can concern the whole module or a single channel.

Executing a program can generate some different run-time errors, caused by the operator or the programmer. These errors will NOT stop program execution but generates an error code.

The automatic error detecting system is enabled by a WHEN ERROR THEN statement. This statement should be followed by a section of statements, which handles an error condition, i.e. closing valves or stopping production. This section of program will be executed if an error occurs in the following part of the task.

The WHEN ERROR THEN statement is task dependent, meaning that the automatic error detecting system is only enabled for the tasks that have executed a WHEN ERROR THEN statement.

The figure below illustrates the structure for a task using WHEN ERROR:



If an error occurs in the first part of the task (* ref.1 *), this will make no change in the program execution, but erroneous data can be loaded and may cause problems, i.e. in calculations.

The error handling part of the program is defined in the section after the WHEN ERROR THEN statement. If an error occurs in the last part of the task (* ref.2 *), this will interrupt the program execution in the statement which caused the error, and move the program execution to the error handling part after WHEN ERROR.

The errorhandling part after WHEN ERROR THEN can end in two ways: the program continues with the statements after the errorhandling part, or it can RETURN to the statement where the error occured and continues from there.

To make the program execution return, a standard procedure RETURN must be called.
WARNING: When using RETURN, the program execution continues in the P-code AFTER the one which the error occured and there is a risk of erroneous data in the following calculations.

To enable, disable, clear and test various error states, corresponding to a number of error bits, some standard procedures/functions are available in PROCESS-PASCAL:

The different errors to clear, disable, enable, raise and test are:

PnetError,	HisError,ModuleError,	ActError,DataError,	
BufferError,	ArithmicError,	IndexError,	ConvertError

The first three errors are caused by external events:

PnetError	corresponds to transmission error on the P-NET,
HisError,ModuleError	corresponds to a historical error or a module error in the accessed module, i.e. the Error3 register is not 0,
ActError,DataError	corresponds to an actual error, a data error in the accessed module.

The next four errors are caused by internal events:

BufferError	a buffer is accessed when it is full/empty,
ArithmicError	division by zero, over-/underflow,
IndexError	array index out of bounds,
ConvertError	error in converting ASCII to numeric.

These last four errors also generates an error code in the controller errorcode, Error3 at softwire 3.

When an error is generated from the operating system, it can assign a number of parameters, a report, to the variable called InterFaceErrorBuffer. This variable is declared in the system file PD3010.SYS as a buffer with 10 elements. Each element is defined as a record of 4 fields, containing information of the variable which caused the interfaceerror. Three different errors can cause the operating system to produce this report, denoted by the following identifiers:

PnetReport, ModuleReport, DataReport.

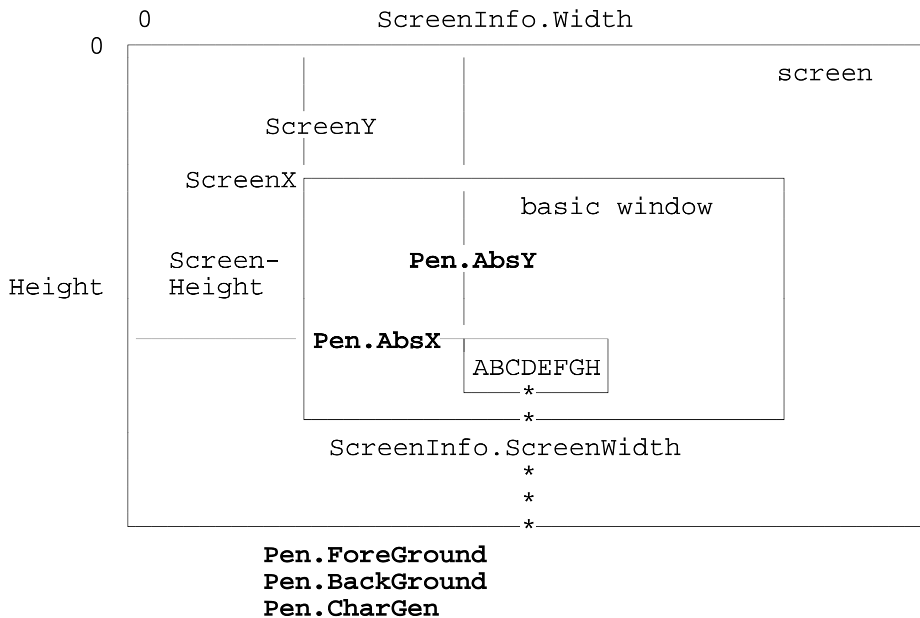
These report bits can be disabled or enabled independently by means of Disable(Error,reportbit) or Enable(Error,reportbit). The WHEN ERROR statement enables all three report bits and all error bits.

PnetReport	only errors on the P-NET will result in a store of an element in the InterFaceErrorBuffer,
ModuleReport	only errors in the accessed module will result in a store of an element in the InterFaceErrorBuffer,
DataReport	only data errors in the accessed module will result in a store of an element in the InterFaceErrorBuffer.

WRITING ON THE SCREEN.

Writing on the screen can be done with different characters and symbols in various sizes, all independent of each other and at the same screen. This means that it is simple to combine text and graphics because all text is written in graphic mode. All texts and symbols can be placed on the screen in any pixel position, so text can be written in proportional writing and with any line spacing.

All what is written on the screen, is written in windows. First the basic window is selected (the window is automatically opened by the selection) and following a number of windows can be opened from that basic window (only used in PD3020). The following description is only concerning writing in the basic window.



When you want to write at the screen, two standard procedures are available: **Display** and **Update.**

Display is used to show a bitmap, writing text or displaying the value of a variable, an expression or a function on the screen. The bitmap, text or variable is shown with the reference point for the first character in (peninfo.x, peninfo.y). When the procedure is done, peninfo.x has been moved to the right ((size * (width of one digit) for numerals, and after the last character or the number of pixels specified by FORMAT for strings), i.e peninfo.x is pointing at the first pixel after the field.

Update is a very powerful procedure, used to change a variable from the keyboard. It combines the possibility to show the current value for a variable on the screen and to assign a new value to this variable from the keyboard.
It is only possible to change, update, a variable if the cursor is inside the field on the screen where the variable is shown. If the cursor is not inside the field, the variable can not be changed from the keyboard and **Update** operates like the standard procedure **Display**.

MODULES IN PROCESS-PASCAL.

Modules are precompiled parts of a larger program with a precise external declaration part related to the host program. The external declaration part declares the constants, types, variables, procedures and functions that must be known by the module.

The idea about modules has certain advantages:

The compiling time for a program is reduced because you must not compile the entire source code.
You can include modules from a library without having the source code available.
You can develop modules to hand over to a third part, without handing over the source code.
You can divide lage programmmes into smaller logically related modules.
You can locate modules at fixed addresses in large programmes and thereby avoid exchanging all EPROM's when you make changes to small parts of the program.

Modules can consist of:

Global constants
Global types
Global variables
Global procedures
Global functions
Complete tasks
A collection of the above.

Example of a constant module:

```
MODULE;

CONST
  CH6x8 = ARRAY[$20..$A0] OF SMALLBITMAP[8] (
                   [$20]:($06, $08, $00, $00, $00, $00, $00, $00, $00, $00), (* space *)
                   [$21]:($06, $08, $20, $20, $20, $20, $00, $00, $20, $00), (* ! *)
                   . . . . . . . . . . . . . . .
                         );
END.
```

The above module is very simple and no external information is required by the module.

Example of a procedure module:

```
MODULE;
EXTERNAL
BEGIN
  CONST
    constant1 = REAL;
  VAR
    variable1 : INTEGER;
    variable2 : REAL;
END;

PROCEDURE ModulProcedure(VAR p1: LONGREAL);
BEGIN
  variable1:=variable2 * constant1;
  p1:=variable2;
END;
END.
```

The procedure in the above module operates on various external parameters. These external parameters must be declared in an EXTERNAL part excactly as they are found in the host program.

ACCESSING NOT DECLARED VARIABLES.

In larger systems with more controllers involved, you can have a need for accessing variables via P-NET which are not declared within the controller. The variables could be found in another controller or in a newly installed interface module connected to another part of the plant and therefore it may be unknown to a number of controllers.

In general it's not possible to access variables, e.g. to display a measured value, without having declared the variable first. However, the system variable NodeList, at Softwire $2C, enables you to access not declared variables.

Before you can access a variable, there are some basic elements which must be known. The elements needed to access a variable are:

- P-NET node address
- Softwire number or absolute address
- Offset
- Bit number

and of course, the type of the variable.

Besides the above, a variable can not be accessed correctly, unless the type of the module which holds the variable is known. According to the P-NET standard, you must specify if the module understands extended or complex P-NET addressing, addressing with offset and so on. Please refer to the chapter INTERFACE for further information.

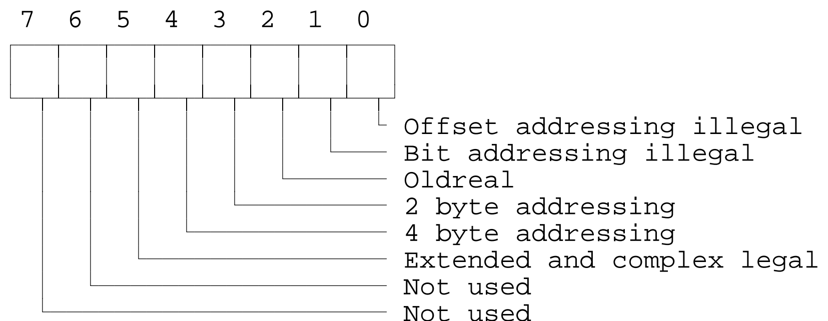
All the above information for a variable can be gathered in a pointer by means of a NODELIST and a pointerfunction. In the system file, NodeList is declared at softwire $2C as an array as shown in the following:

```
(*2C *) NodeList: ARRAY[1..10] of NodeListElement;
```

where each element is a record of the following type:

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

The field variable CODE in a nodelist element is declared as a byte with the following meaning:



Please refer to the P-NET manual for further details on the addressing facilities.

DeviceType is an integer type denoting the device type for the module you want to access.

NodeAddr denotes the P-NET node address for the module you want to access. NodeAddr is declared as a string, where NodeAddr[0] holds the number of elements to specify the entire node address including port numbers and node numbers. The total length of the string denoting the node address must not exceed 20 characters. The port numbers and node numbers in the node address are NOT ASCII characters but hexadecimal numbers in the range 0..$7F.

When you have defined an element in the nodelist, Code, DeviceType and NodeAddr, you can get access to the module specified within the list. You get access to the module by means of a pointer (see chapter POINTER TYPES how to declare and use a pointer) which is set to point to the variable you want to access.

The pointer you are using is set to point to the variable by means of the function POINTERTONODE, which is a standard function in PROCESS-PASCAL. PointerToNode operates on an element from the NodeList, a Softwire number, an offset and a bit number and it returns a pointer. The syntax for PointerToNode is the following:

```
MyPtr -> PointerToNode(Node, SWNo [, Offset [, BitNo]]);
```

Node denotes an element from the NodeList and SWNo specifies the actual Softwire number you want to access. If the variable is of a complex type, you can use the Offset to select a simple type variable. If the variable is a boolean array, you can use the BitNo to select the bit number. Offset and BitNo are optional and they are set to zero in the function call if omitted.

If Node = 0, then the pointer is set to an internal variable at the specified Softwire number.

Example where PointerToNode is used:

```
VAR
  RealPtr : POINTER TO REAL;
  NodeNo, NodeSWNo, NodeOffset : INTEGER;

PROCEDURE ShowVariable;
BEGIN
  PenTo(0,0);
  Display('Select node number ');
  Update(NodeNo:2:0);
  PenTo(0,8);
  Display('Select softwire no ');
  Update(NodeSWNo:4:-3);
  PenTo(0,16);
  Display('Select offset      ');
  Update(NodeOffset:4:0);
  RealPtr -> PointerToNode(NodeNo, NodeSWNo, NodeOffset);
  PenTo(0,24);
  Display('Value for variable ');
  Display(RealPtr:6:2);
END;
```