Process-Pascal

Manual

GB

PROCES-DATA A/S

NAVERVEJ 8, DK-8600 SILKEBORG • Tel. +45-87 200 300 • Fax +45-87 200 301 • info@proces-data.com • www.proces-data.com

August 2007

Contents

1 Introduction to Process-Pascal.

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

Process-Pascal extends Standard Pascal with a number of facilities to make it possible to execute several programs simultaneously in one computer. This is called multi-tasking.

Process-Pascal has been specifically developed for use in connection with the P-NET fieldbus, which is a local area network for the transmission of data throughout distributed data acquisition and process control plants. Data, which are located within physically distributed modules connected to P-NET, can be defined as variables in Process-Pascal programs.

Process-Pascal permits automatic reports of alarms to be produced in the event of an error occurring in the controller or in any of the interface modules. Furthermore, it is possible to automatically test all the components that make up a plant, during the start-up 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 to it using the screen cursor.

Process-Pascal programs can be written using any general-purpose editor that produces ASCII or ANSI files.

Process-Pascal programs operate with several types of variables. The compiler automatically performs typecasting during compilation. This makes programs easier to develop.

The Process-Pascal program suite provides a debugger, which is a very powerful tool that significantly speeds up the overall process of application development.

Process-Pascal programs are compiled using a cross-compiler running on a PC under different Windows environments. The compiler generates code, P-code, which is stored in the controller in FLASH or RAM memory. The operating system in the controller interprets the P-codes and executes a piece of machine code for each one.

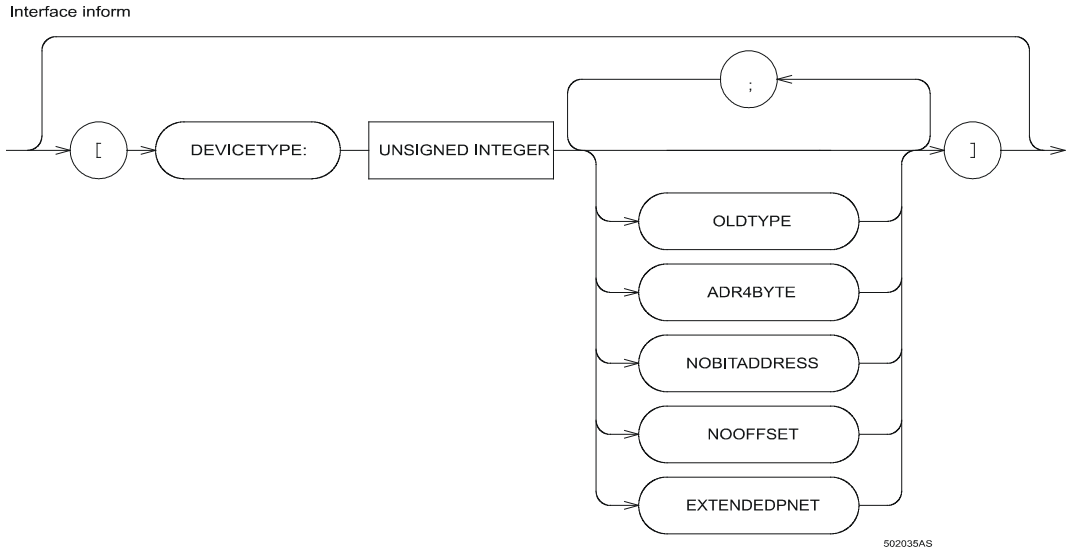
Process-Pascal programs cannot be executed on a PC. The compiler has been completely developed by PROCES-DATA A/S.

This manual has been written for programmers who have some familiariarity with the Pascal language, and understand the basic structures for data and programs.

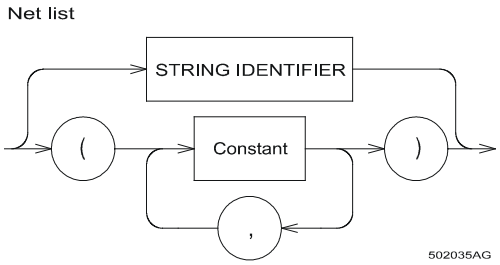
This manual should be used in conjunction with the manual for the particular controller, for which a Process-Pascal program is being written.

2 Interpretation of Syntax diagrams

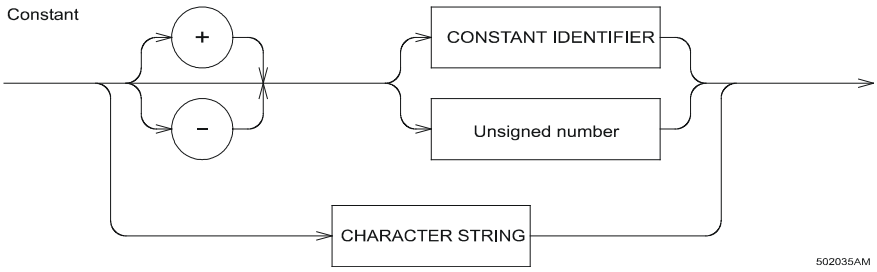
Syntax diagrams are drawings that illustrate valid Process-Pascal syntax.



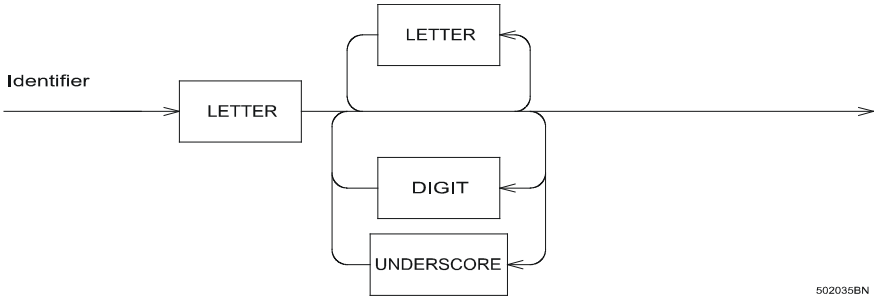
To read a diagram, trace it from left to right, in the direction shown by the arrows. Commands and other keywords appear in UPPERCASE inside ovals. Type them exactly as shown in the ovals. Operators, delimiters, and terminators appear inside circles. If the syntax diagram has more than one path, you can choose any path to travel.
Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one constant, you can go back repeatedly to choose another, separated by commas.



A rectangle with text in lowercase represents another syntax diagram. When such a rectangle is met, jump to the specific syntax diagram. When exiting from the specific syntax diagram continue after the rectangle with text in lowercase.
Constant is an example on jumping to another syntax diagram, see the following diagram.



A rectangle with UPPERCASE text is a description, which is not explained any further.
In the following syntax diagram, Digit is not explained any further



A complete alphabetically sorted list of syntax diagrams is inserted as an appendix to this manual.

3 Program Structure in Process-Pascal

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

```
Program name [objecttype, capabilities];

VAR
        global variable

 Procedure
        global procedure

Function
        global function

Task name1;
VAR
        local variable
Begin

End;

Task name2;
VAR
        local variable
Begin

End;

Task name3;
VAR
        local variable
Begin

End;

End.
```

Program Heading.

A program heading consists of the word **Program** and a name. Furthermore, it contains an objecttype and a list of capabilities for the controller.

Global variable declaration.

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

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

Global procedure and function declaration.

Global procedures and functions can be called from all internal tasks.
A global procedure/function can be called from several tasks simultaneously, using different sets of parameters.

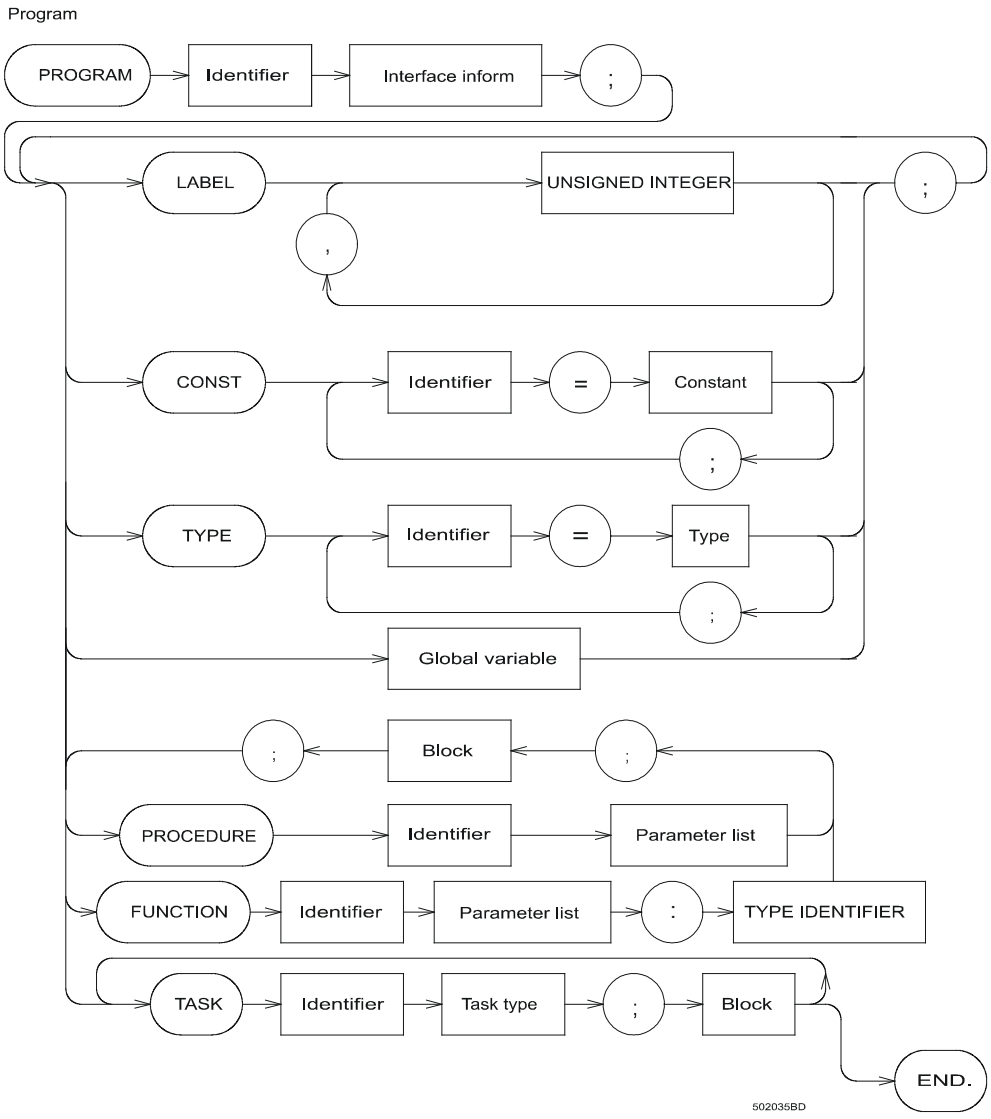
Task declaration.

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

Tasks are used to monitor and control different jobs that might occur simultaneously. Defining each independent job in a separate task achieves this.

Data are exchanged between other tasks and the ‘outside world ', via global variables.

Unlike procedures, tasks are not called, since they are always present in a task queue, and are ready to run whenever conditions dictate.



A Process-Pascal program is divided into a heading and a body. The body is called a block. The heading gives the program a name, an object-type to be used by VIGO, and the capabilities for the Controller (also used by VIGO). The block consists of seven sections: LABEL, CONST, TYPE, VAR, PROCEDURES and FUNCTIONS and TASK declaration, where any, except the last, may be empty.

Everything that is defined before any of the tasks is called the global section, and it is possible to have as many declaration sections as required, in any convenient order, including procedure and function 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. each consists of a heading and a block. The titles used in the heading are different (TASK, PROCEDURE, FUNCTION, instead of PROGRAM), and they all end with a

semi-colon instead of a period. They can have their own constants, data types and variables, and even their own procedures and functions.

Tasks differ from procedures and functions in a number of ways:

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 operate entirely autonomously from other tasks.
3. TASKs cannot be nested.
4. TASKs do not need to be called from a statement in order to execute.

The next sections describe the various usage of tasks.

4 Tasks

Multi-tasking is a facility provided in Process-Pascal to make it possible to execute several sub-programs simultaneously, within the very same computer. These sub-programs are called TASKs, and are fundamental to Process-Pascal. They make it very easy to split a program up into manageable portions, where each TASK performs a distinct function.

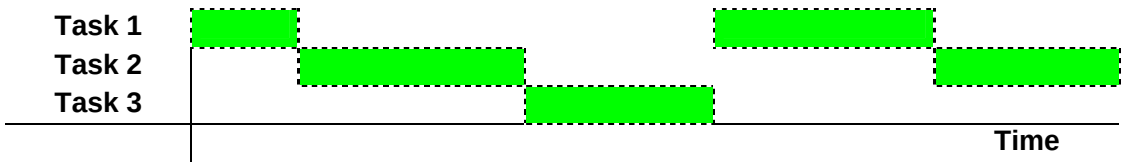
Multi-tasking is very useful for process control, where the process can be controlled in real time.

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 needs to, before it relinquishes control of the processor, and allows another TASK to run. Whilst in reality, the TASKs are not performed in parallel, the switching between them is performed fast enough for the complete system to be regarded as operating in real time. Switching from one TASK to another can be specified to occur at any points within the program, including procedures. These can advantageously be used when a delay is known or is likely to occur, or in case a TASK is waiting for some actions to take place, e.g. for a certain level on an input signal to be achieved or for a TIMER to run out.

Switching to another TASK in such situations makes the program more efficient, and wastes no time waiting.

The statement CHANGETASK, which is a standard procedure in Process-Pascal, is used to perform the switching from one TASK to another. The actual TASK that calls CHANGETASK, stops program execution in that TASK, and then relinquishes control of the processor for use by the next TASK, which continues program execution from where it was last interrupted (e.g. by a previous CHANGETASK).

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



4.1 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 a CHANGETASK switches to the following one in the sequence. The sequence is defined by the order in which the TASKs were written in the program.

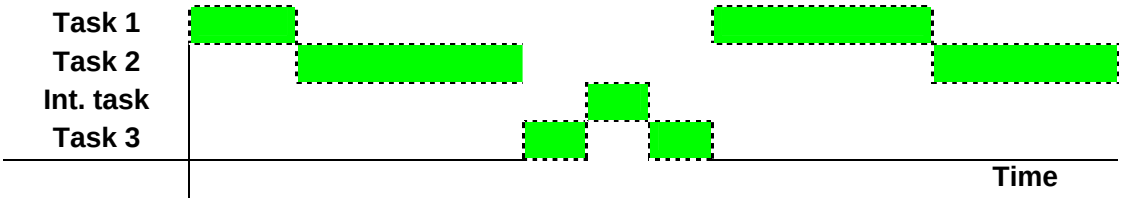
TIMEDINTERRUPT TASKs are executed at certain time intervals, as controlled by the programmer. The time periods are declared in seconds and the resolution is 1/128 second.

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

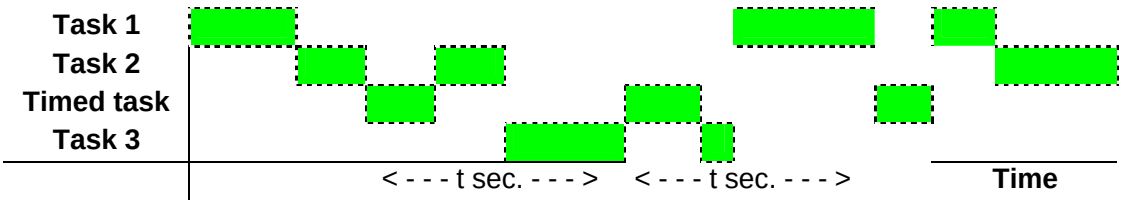
When a cyclic TASK is running and a TIMEDINTERRUPT or SOFTWIREINTERRUPT TASK is ready to run, a CHANGETASK is forced in the cyclic TASK, and control is given to the interrupting TASK. When the interrupting TASK has finished, i.e. reaches a CHANGETASK statement, this CHANGETASK makes the previous cyclic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

A TIMERINTERRUPT or SOFTWIREINTERRUPT TASK
cannot be interrupted by other TASKs.

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



The diagram below shows the principle of how program execution is switched, when a TIMEDINTERRUPT TASK interrupts a number of cyclic TASKs. The TIMEDINTERRUPT TASK is set to occur every t seconds.



4.2 How to Split a Program into Manageable Tasks

Several separate TASKs may be monitoring and controlling various devices simultaneously. A single TASK could be written to monitor the keyboard for user input. Another TASK might be responsible for what is displayed on the screen. Yet another TASK may be monitoring a flow meter, waiting for flow to start. One TASK may be affected by what happens in another one. e.g. The keyboard TASK detects a key press, which indicates that a process is to be started. This indicates to the TASK monitoring a flow meter to begin to take notice

of a flow rate, which in turn, causes the flow rate to be displayed on the screen by the display TASK.

Splitting a program up into TASKs is considered by assessing which activities need to be performed simultaneously. Each of these activities may then be implemented as an individual TASK. Taking the examples described above, it can be seen that the three tasks mentioned, i.e. monitoring the keyboard, monitoring a flow meter and displaying the data on the screen, all need to be performed simultaneously, and are therefore logical candidates for being structured as separate TASKs.

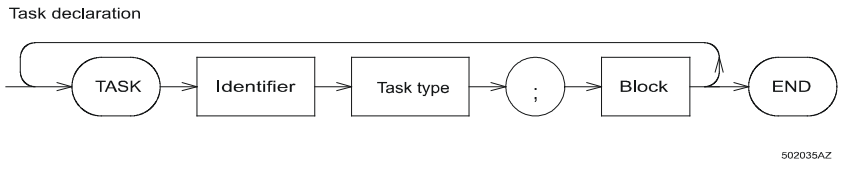
Proper *changetask* usage
Establishing the appropriate moments for switching between tasks, is an inherent part of designing a well structured and efficient Process-Pascal application. The main concept behind this is that the ***changetask*** statement should be used in any part of the task where relatively long processor usage can be expected, for example, within loops.

Differences between interrupt tasks and event handling procedures
It is very important to understand the difference between multi-tasking programming and event-driven programming.
It is a fact that event-driven programming plays a key role in MS Windows application development. In this case, a relevant procedure (function) is called when a certain event occurs. It is performed and then the control is given back to the main application.
Multi-tasking programming is based on another concept. When a cyclic task is running and a timed interrupt or SoftWire interrupt 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 from where it was interrupted. It means that timed interrupt or SoftWire interrupt tasks should always contain a LOOP statement and at least one ***changetask***. Otherwise the task will get a SUSPENDED status after the first run (if the LOOP is absent), or other tasks will never get back the control (if the ***changetask*** is absent).

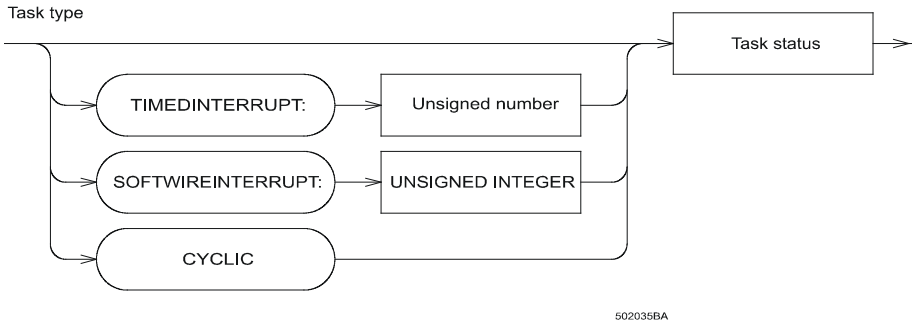
4.3 Task Declaration

Process-Pascal handles 3 different types of TASKs: CYCLIC TASK, TIMEDINTERERUPT TASK and SOFTWIREINTERRUPT TASK. All 3 types of TASK can be used within the same program.

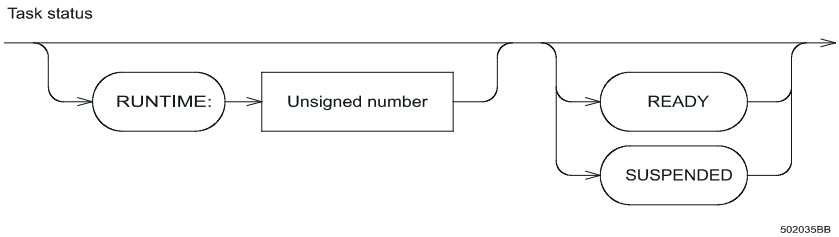
The task declaration serves to define a program part and to associate it with an identifier. The declaration has the same form as a program, a heading and a block.



The task heading names the task's identifier and specifies the task type.



A task declared without a task type attribute is given the default task type CYCLIC.



The max RUNTIME is given by a real constant and is declared in seconds with a resolution of 1/128 second. This can be used to force a changetask within the task to ensure that it will not prevent other tasks from running if it enters a "loop forever", when a changetask statement has not been included in the loop. The default RUNTIME is 300 seconds.

The max RUNTIME can be changed during program execution with the standard procedure MAXRUNTIME(time), where time must be a constant or a variable, denoting the new max runtime in seconds.

Below is an example of a timed interrupt task that runs every second:

```
TASK ProcessTime TIMEDINTERRUPT: 1.0;
VAR
  Hour, Min, Sec: Integer;
BEGIN
  Hour:=0; Min:=0; Sec:=0;
  REPEAT
    Sec:=Sec+1;
    IF Sec = 60 THEN
    BEGIN
      Sec:=0;
      Min:=Min+1;
      IF Min = 60 THEN
      BEGIN
        Min:=0;
        Hour:=Hour+1;
      END;
    END;
    ChangeTask;
  UNTIL Not Process_On;
```

```
    END; (* Task ProcessTime *)
```

The initial task status can be declared to be READY or SUSPENDED. The default task status is READY.

All the tasks are linked within a task chain system. The cyclic tasks form a cyclical ordered chain, with one task pointing to the next cyclic task, and where a CHANGETASK switches to the next one in the chain. The order of the cyclic tasks in the chain is defined by the order of the TASKs in the program, but this order can be changed if any of the tasks change status while the program is running.

The timed interrupt tasks form another chain, where the order of the tasks is determined by the defined period of activation. TimedInterrupt TASKs are executed at certain time intervals, controlled by the programmer. The time period is specified by a real constant and is declared in seconds with a resolution of 1/128 second.

The SoftWire interrupt tasks are held in a third chain, where execution is determined by the corresponding interrupt connections. The interrupt connections and interrupt conditions are declared in the global variable declaration.

When a task is included in a chain, the task status is set to READY. A SUSPENDED task will have been removed from the task chain system, and will not be able to run as long it remains suspended.

A SUSPENDED task can change to READY status, if another RUNNING task calls the standard procedure CONTINUETASK with the appropriate task identifier, CONTINUETASK (TaskIdentifier). This will insert the task in the appropriate task chain again, and enable the task "TaskIdentifier" to continue from where it was last stopped or interrupted.

A READY task can change to SUSPENDED status, if another RUNNING task calls the standard procedure STOPTASK with the appropriate task identifier, STOPTASK (TaskIdentifier).

This will remove the task from the task chain system and prevent the task "TaskIdentifier" from running any further, until it is changed to READY again from another task by means of CONTINUETASK(TaskIdentifier) statement.

If a task arrives at an END for the task block, its status is automatically changed to SUSPENDED, the task is removed from the chain and the task program counter is set to the beginning of the task. Subsequently, the task will RESTART, if another task calls the CONTINUETASK(taskidentifier) statement.

A RUNNING task can force itself to RESTART from the beginning of the task. To perform a restart for a task, the standard procedure RESTARTTASK is called. After calling RESTARTTASK, program execution will continue with the first statement within the task.

To enable tasks to be started and stopped within a program, it is required that the task identifiers are declared before they are used. This can, in some cases, be impossible. To solve this problem, a FORWARD declaration of task identifiers can be included. A FORWARD declaration must be placed as the first task(s).

Example of a forward declaration:

```
FORWARD Task WeightControl;
```

The time interval for a timed interrupt task can be changed during program execution, by means of the standard procedure TIMEDINTERRUPTTIME(time), where time can be a constant or a variable, denoting the interval time in seconds. The time is specific to the task that calls the procedure, so the procedure must be called from the task where the time must be changed, i.e. a task can only change its own time. When a task is declared to be a timed interrupt task, the time interval must be included in the task heading.

Example of a task heading for a TimedInterrupt task:

```
TASK ProcessTime TIMEDINTERRUPT: 1.0;
```

Within cyclic tasks, TIMEDINTERRUPT TASKs can be ENABLED, i.e. allowed to interrupt, or DISABLED, not allowed to interrupt cyclic tasks. ENABLE(TimedInterrupt) is a standard procedure, to be used in **cyclic** tasks, to allow timed tasks to interrupt the cyclic task. In all cyclic tasks, TIMEDINTERRUPT TASKs are ENABLED as default after a reset.

DISABLE(TimedInterrupt) is a standard procedure to be used in cyclic tasks, which disables all TIMEDINTERRUPT TASKs, i.e. denotes that no timed interrupt task is allowed to interrupt this cyclic task. Disabling the timed interrupt tasks will not change the status of these tasks. This means that they are not removed from the task chain, and when the timed interrupt tasks are enabled again, they will try to catch up with any lost time. If a timed interrupt is disabled or enabled from within a procedure or a function, the interrupt status is automatically set back to the state it held before the call, after the procedure or function has been completed.

SOFTWIREINTERRUPT TASKs are executed each time a certain globally defined variable is accessed. The conditions for activating the interrupt when accessing the variable are set by the variable declaration.

See the chapter INTERRUPT about connecting an interrupt to a variable. A number in the range 0 to 31 gives the interrupt connection. Several global variables may be connected to the same interrupt number.

Example of a task heading for a softwireinterrupt task:

```
TASK Keyboard SOFTWIREINTERRUPT:0;
```

Any task can **change task type** during program execution.

A CYCLIC task and a SOFTWIREINTERRUPT task can be changed to a TIMEDINTERRUPT task, by means of the standard procedure TIMEDTASK. Before chang-

ing the task type to TimedInterrupt, the interval time must be selected, TimedInterruptTime(time), or a default value of 255 seconds is used. Changing the task type, will not affect program execution, and the task will continue until it meets a ChangeTask statement.

A TIMEDINTERRUPT task and a SOFTWIREINTERRUPT task can be changed to a CYCLIC task by means of the standard procedure CYCLICTASK. Changing the task type will insert the task within the cyclic sequence, and subsequent program execution for the task will continue until it meets a ChangeTask statement. When the task runs again, it is still within the cyclic sequence and thus continues from the statement following ChangeTask.

A CYCLIC task and a TIMEDINTERRUPT task can change to a SOFTWIREINTERRUPT task, by means of the standard procedure INTERRUPTTASK, but only if the task was originally declared as a softwireinterrupt task. The interrupt connection is set to the initial softwireinterrupt number (declared in the task head). Changing the task type will not affect program execution, and the task will continue until it meets a ChangeTask statement. The task will continue with the next statement, when an interrupt with the corresponding interrupt connection is generated.

5 Defining Data

5.1 Variables

A variable possesses three characteristics:

1: a name
2: a type
3: a current value

The variable is identified by a name. This name is used throughout the entire program.

When a variable is declared, its type must also be stated. A variable's type circumscribes the set of values it can possess, and the operations that can be performed upon it.

The value of a variable may change during program execution. When a variable has been declared, but before a value has been assigned to it, it is said to be undefined.

5.2 Identifiers

Any names given to constants, types, variables, bounds, procedures, functions etc., are called identifiers.

They must begin with a letter, which may be followed by any combination and number of letters and digits. Corresponding upper-case and lower-case letters are considered equivalent. Letters can be in the range from 'a' to 'z', an underscore '_' and the Danish letters 'æ', 'ø' and 'å'.

Examples of identifiers:

Temperature MultiFunc ProcessTime FirstKey
ModePort1 This_Is_A_Very_Long_Identifier

Certain identifiers are reserved (word-symbols or reserved words). A reserved word cannot be used as an identifier.

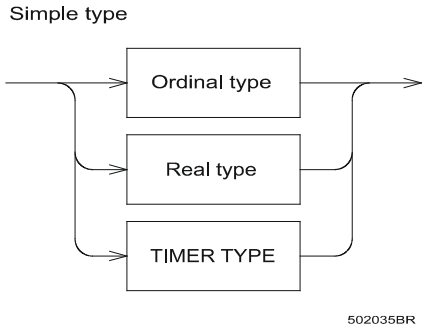
Process-Pascal provides a number of pre-declared identifiers. These pre-declared identifiers are not reserved words, but names for standard procedures, functions and so on. These names should not be used either, to avoid any mistakes. A complete list of all reserved words and pre-declared identifiers in Process-Pascal is given in chapter 35.3.

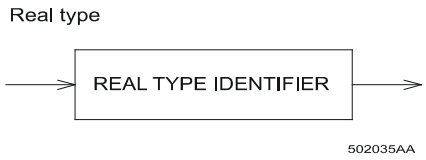
6 Simple Data Types

A program uses data of various formats and for various functions. The formats, and partly the function, are determined by the data type.

A data type defines the set of values a variable may assume and the basic operations that may be applied to it. Every variable occurring in a program must be associated with one and only one type.

Simple data types define ordered sets of values and are one of the predefined types 'REAL', 'LONGREAL', 'TIMER', 'REALDATE' or an ordinal type.





6.1 Ordinal types

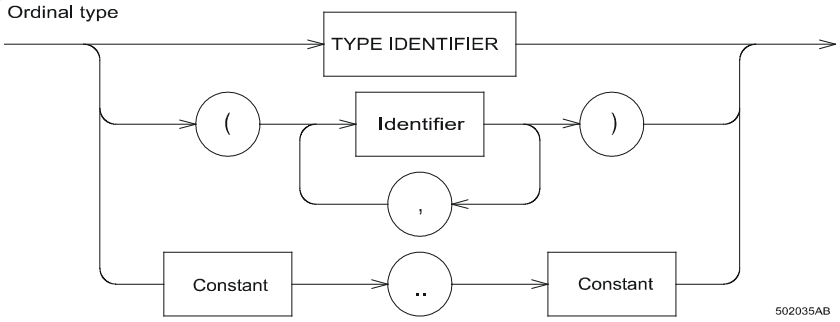
Ordinal types are a subset of simple types. Ordinal types possess the following four characteristics:

1: All possible values of a given ordinal type are part of an ordered set, and each possible value is associated with an ordinality, which is an integral value. Except for integer type values, the first value of every ordinal type has ordinality 0, the next has ordinality 1, and so on, for each value of that ordinal type. An integer type value's ordinality is the value itself. In any ordinal type, each value, other than the first, has a predecessor, and each value, other than the last, has a successor, based on the ordering of the type.

2: The standard function **Ord** can be applied to any ordinal type value to return the ordinality of the value.

3: The standard function **Pred** can be applied to any ordinal type value to return the predecessor of the value. The predecessor is defined by Pred(x) < x, and Ord(Pred(x)) = Ord(x) - 1.

4: The standard function **Succ** can be applied to any ordinal type value to return the successor of the value. The successor is defined by Succ(x) > x, and Ord(Succ(x)) = Ord(x) + 1.



Process-Pascal includes 6 predefined ordinal types: **Integer, Byte, Word, Longinteger, Boolean,** and **Char**. In addition, there are two other classes of user-defined ordinal types: enumerated types and sub-range types. These types are described in the USERDEFINED TYPES chapter.

6.2 The BOOLEAN type

A boolean value is one of the logical truth values, expressed by use of the predefined constant identifiers **false** and **true**. In Process-Pascal, the additional predefined constant identifiers **Off** equals **false,** and **On** equals **true**.

Relational operators (=, <>, <=, <, >, >=) can be used within a boolean expression, and the following relationships hold:

```
False < True
Ord(False) = 0
Ord(On) = 1
False = Off
True = On
```

Pre-declared BOOLEAN functions, i.e. pre-declared functions, which yield a BOOLEAN result, are:

BufferEmpty(buf)	true if the buffer is empty
BufferFull(buf)	true if the buffer is full
Odd(i)	true if the integer i is odd

The buffer functions are described in detail in the BUFFER chapter.

6.3 The CHAR type

This type's set of values consists of characters, ordered in accordance with the ASCII character set. The function call **Ord(ch)**, where ch is a char value, returns ch's ordinality, which is the numerical ASCII value for that character.

Any value of the type char can be generated with the standard function **Chr(value)**.

A character enclosed in apostrophes (single quotes or double quotes), denotes a value of the char type.

To represent a single quote, enclose it in double quotes. To represent a double quote, enclose it in single quotes.

Examples of values of char type:

'a' 'H' '8' "e" """" ""

6.4 The INTEGER type

There are four predefined integer types in Process-Pascal: **byte, integer, word** and **longinteger**. Each type encompasses a specific subset of whole numbers, as shown in the following table:

TYPE	RANGE	FORMAT
byte	0 .. 255	unsigned 8-bit
word	0 .. 65535	unsigned 16-bit
integer	-32768 .. +32767	signed 16-bit
longinteger	-2147483648 .. +2147483647	signed 32-bit

Arithmetic operations having an integer type operand, use 8-bit, 16-bit or 32-bit precision, according to the following rules:

1: The type of an integer constant will adopt the predefined integer type that has the smallest range and includes the value of the integer constant.

2: Binary operations can be performed with all integer types. For a binary operator, both operands are converted to their common type before the operation. The common type for a byte and a word is word, which means that a binary operation on a byte and a word converts the byte to a word, and then the operation is performed.

3: The expression on the right side of an assignment statement is evaluated dependent on the type of the variable in the expression and the type on the left side.

An integer type is converted to another integer type through typecasting. Typecasting is automatically performed during compilation.

A special typecasting can be performed to convert integer types to Boolean array types and vice versa, through a CONVERT function. The CONVERT function performs the typecasting in accordance with the following table:

INTEGER TYPE	BOOLEAN ARRAY SIZE
Byte	8
Integer	16
Word	16
longinteger	32

Examples of using the CONVERT function:

```
  TYPE
    BIT8  = ARRAY[0..7]  OF BOOLEAN;
    BIT16 = ARRAY[0..15] OF BOOLEAN;
    BIT32 = ARRAY[0..31] OF BOOLEAN;

  VAR
    Bit8Var  : BIT8;
    Bit16Var : BIT16;
    Bit32Var : BIT32;
    ByteVar  : BYTE;
    IntVar   : INTEGER;
    LIntVar  : LONGINTEGER;

  BEGIN
    ByteVar:=Convert(Bit8Var);    (* convert an 8 bit boolean array
to a byte *)
    Bit16Var:=Convert(IntVar);      (* convert an integer to a 16
bit boolean array *)
    Bit32Var:=Convert(LIntVar);   (* convert a longinteger to a 32
bit boolean array *)
```

This CONVERT function is very useful when it is required to mask out some bits, or to read a combination of bits as data in conjunction with digital inputs and outputs.
NOTE: the boolean array must start with index 0.

6.5 The REAL type

A real type has a set of values that is a subset of real numbers, which can be represented in floating-point notation with a fixed number of digits.

There are two kinds of the real type: **real** and **longreal**.

The **real** type occupies 4 bytes of memory, in a format according to the IEEE 754 standard for short real format (the same format as used in the 80x87 math processor for a single real type), providing a range of 3.4 * 10E-38 to 3.4 * 10E38 with 7 significant digits.

The **longreal** type occupies 8 bytes of memory, in a format according to the IEEE 754 standard for long real format (the same format as used in the 80x87 math processor for a double real type), providing a range of 1.7 * 10E-308 to 1.7 * 10E308 with 15 significant digits.

6.6 The TIMER type

The **timer** type occupies 4 bytes of memory and is assigned as a **real**. The value for a variable of a timer type is in seconds. A timer type variable counts down with a resolution of 1/128 second. The count down **continues** through negative values.

The timer stops counting down when the power is off.

A timer type variable can be used anywhere in the program. It is commonly used by assigning a value to it and afterwards testing whether the value of the variable is <= 0.0.

The number of defined variables of timer type has no effect on the program execution time. TIMERS cannot be set to values higher than 1.6777 * 10E7, which corresponds to 4660 hours or 194 days.

6.7 The REALDATE type

The REALDATE type is based on the same type as the longreal type.
The integral part of a REALDATE value is the number of days that have passed since 12/30/1899. The fractional part of a REALDATE value is fraction of a 24 hour day that has elapsed.

Following are some examples of REALDATE values and their corresponding dates and times:

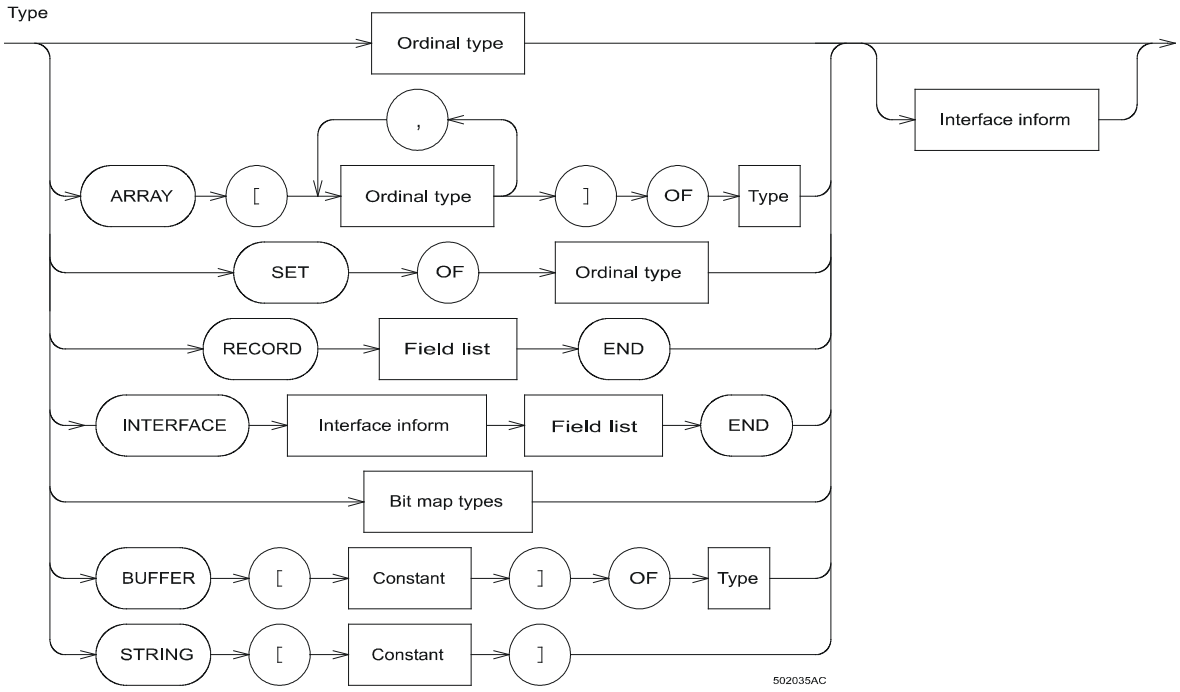
Date and time	**Representation**
30 December 1899, 00.00	0.00
1 January 1900, 00.00	2.00
4 January 1900, 00.00	5.00
4 January 1900, 06.00	5.25
4 January 1900, 12.00	5.50
4 January 1900, 21.00	5.875
4 January 1900, 21.30	5.89583333

To find the fractional number of days between two dates, simply subtract the two values.

7 Structured Types

Process-Pascal provides facilities for creating collections of data types in the form of structured types. Although data types can be quite sophisticated, each must ultimately be built up using unstructured simple types.

A structured type, characterised by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.



Each of the methods for structuring types is described in separate chapters.

8 Variable Declaration

Every variable identifier used within a program must initially be introduced using a variable declaration. This declaration must be textually introduced prior to using the variable.

A variable declaration introduces a variable identifier and its associated data type, by listing the identifier followed by the type. The type chosen for a variable can either be a type identifier previously declared in a type declaration part in the same block or in an enclosing block, or it can be a new type definition. The reserved word **VAR** heads the variable declaration part. It is permitted to type **VAR** more than once within the same variable declaration part.

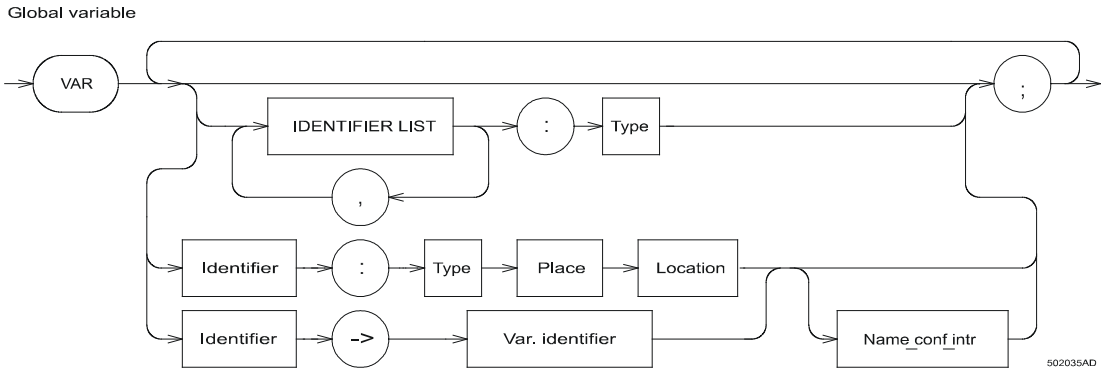
Variables can be declared to reside inside the controller, or externally in other devices at a net-address. The compiler will automatically allocate memory for internal variables, or they can be declared to reside at specific memory addresses for specific hardware applications.

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 are only recognised by the procedure or function within which they are defined.

8.1 Global variables

All the global identifiers that have been declared in a Process-Pascal program are given a number by the compiler. These are called **SoftWire** numbers (SWNo), and are used as an entry key into the **SoftWire list,** which contains information about the type and structure of each individual global variable and constant used in the particular program.

Variables of the same type can be declared using a list of identifiers, separated by a comma, followed by a colon, then stating the common type for these variables.

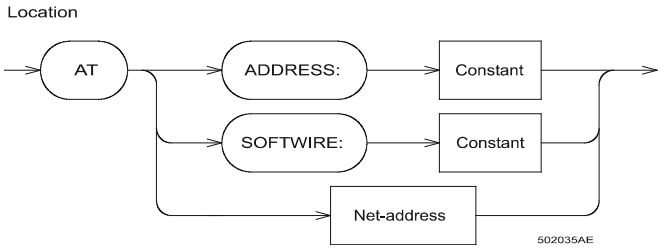


Examples of variable declarations:

```
VAR
  LineNo, PageNo : INTEGER;    (* The memory for these variables *)
  Color : BYTE;                (* are allocated   *)
  Process_On, AlarmState : BOOLEAN;   (* by the compiler *)
```

```
  Wait, LightTime : TIMER;
  Limit : REAL;
```

Variables can be defined to reside either at a specific address in memory, at a specific SoftWire number or at a net-address. When a variable has been declared to reside at a SoftWire number or at a net-address, memory will have already been allocated for it within the controller or within the remote device connected to P-NET



An **Address** clause is followed by an absolute memory address, and only one identifier can be specified.

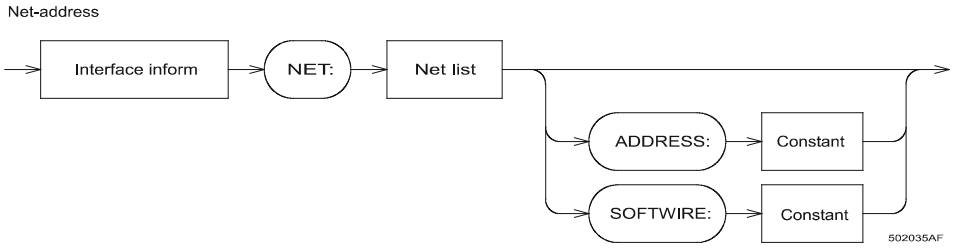
Example of a variable declaration for a specific memory address:

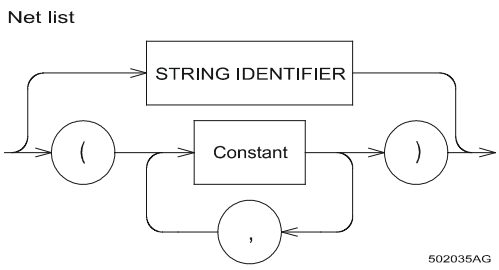
```
  VAR
    LightValue : WORD AT ADDRESS : $00FFFF08;
```

The **SoftWire** clause is followed by a specific number from the SoftWire table. The declaration is rarely used without a net-address, because a global variable has to be declared in order to generate an entry in the SoftWire table in the first place.

8.2 Variables on P-NET

Variables that are physically located within a module connected to P-NET, must be declared to reside at a certain location, defined by a net-address. When variables are declared together with a net address, no memory space is allocated within the controller.





The net-address is denoted by a net list, followed by an address, which can be an absolute address or a SoftWire number.

The **net list** holds an ordered set of numbers, which describes the path to the device, i.e. denoting the port-numbers and P-NET numbers for the module containing the variable.

The net list can also be a string-identifier. This means that the net list can be a string-variable, and the P-NET node address for the module can therefore be set or changed during program execution.

Example of a variable declaration, using a net list:

```
  VAR
    DigModule : PD3221 AT NET: ( 1,64);
```

This variable declaration defines an entire interface module of the type PD3221 including all its channels and registers, which is to reside within the P-NET environment. The device is connected to the Controller via P-NET at port 1, and the device node address is 64.

DigModule is a global identifier for the entire interface module, and can be used in the same way as any other identifier throughout the program.

PD3221 is the type of the variable, which is a pre-declared type specifying the internal organisation of the channels within the module. See PDMODULE.DEF in the Process-Pascal library

AT NET specifies that the declared variable is an external variable that is located on P-NET. Any access to that variable is performed via the network. The following parameters (1, 64) specify where the module is located, as seen from the controller. The first parameter indicates the communication port (Port 1 in this case), and the next parameter defines that the module is expected to have node address number 64.

The **ADDRESS** and **SOFTWIRE** clauses denote a specific address or a Softwire No. within the module defined by the net list.

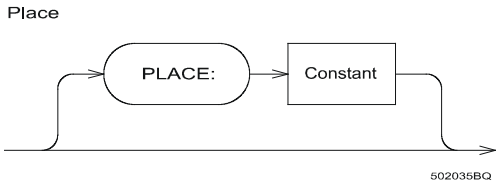
Example of a variable declaration, using a net list and an ADDRESS clause:

```
VAR
  Mixer1 :
    MixerController[devicetype:5000] AT NET:(1,37)ADDRESS: $0C00;
```

Examples of variable declarations, using a net list and a SOFTWIRE clause:

```
VAR
  BeltControl :
    BeltConType[devicetype:5000] AT NET: (1,38) SOFTWIRE: $92;
  ExtInt :
    Integer[devicetype:5000] AT NET: (2,3) SOFTWIRE: $124;
```

If it is required to declare a variable to reside at a fixed SoftWire number, e.g. a global database for a number of controllers, the **PLACE** clause must be used. (Examples of this can be found in the system files for the Controllers).

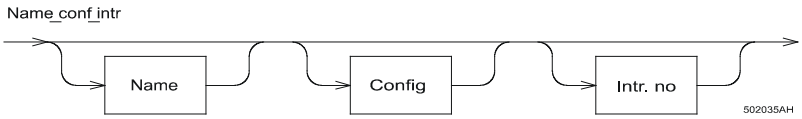


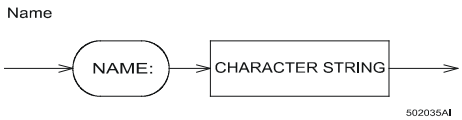
Example of a variable declaration, using the PLACE clause:

```
VAR
  DataBase : ARRAY[1.2000] OF INTEGER PLACE: 200;
```

This declaration will fix the variable at SoftWire No. 200. It must be ensured that the declaration for a specific location is made at a point before the compiler would generate the SoftWire number automatically. i.e. it is too late to place a variable at SoftWire number 200, if 300 variables have already been declared.

A variable can be declared with a **NAME**, as a string constant for that variable. This name can be used as a string when an error occurs involving the variable. See details about errors and error handling in the WHEN ERROR chapter.





Examples of variable declarations, using a name:

```
VAR
  DigModule : PD3221 AT NET: ( 1,35)
        NAME : 'Digital module panel 1';
  AnaModule : PD3240 AT NET: ( 1,38)
        NAME : 'Analog controlunit 22';
```

When using NAME with variables of interface type (modules), conforming to the section INTERFACE DECLARATION in the P-NET Standard, each channel can get its own name. NAME for the module belongs to channel 0, the Service Channel. When using NAME on variables other than interface modules, each variable can only have one name.

8.3 Config

A **CONFIG** clause can be added to a variable declaration such as those discussed in the previous section. This will enable specific sub-elements within that variable to be set (configured) to particular values. The Config clause calls a single or a list of specific procedures, which have been designed to perform the configuration, and takes the form:

```
  CONFIG: Procedureidentifier ( identifier, value);
```

The parameters given in the clause consist of the remaining part of a complete global variable identifier, and an expression denoting the value to which the variable is to be set. The particular procedure called, depends on the type of variable being configured (e.g. byte, real). All such Config procedure calls are instigated by calling the procedure ModuleConfiguration from within the main program. This in turn calls the particular procedure defined in each clause, using the defined parameters.

See the examples included in the Service and Config programs in the Examples folders in the Process-Pascal library.

Examples of variable declarations, using CONFIG:

```
VAR
DigModule : PD3221 AT NET: ( 1,35)
    NAME : 'Module at CIP unit'
    CONFIG : SetByte(.Service.ModuleConfig, WatchDog);

AnaModule : PD3240 AT NET: ( 1,38)
    NAME : 'Inlet control unit'
    CONFIG : Standard_PT100(.Analog_In_4);
```

When the variable is of complex type, a component part of this variable can be specified as part of the first parameter. When the variable is an entire module, a channel or even a register can be selected to be the parameter The procedure call passes the declared variable itself as a default and appends the remaining part of the identifier to form the complete identifier parameter. See the example above. The equivalent procedure calls for the above Config clauses would therefore be:

```
    SETBYTE(DIGMODULE.SERVICE.MODULECONFIG, WATCHDOG);
    STANDARD_PT100(ANAMODULE.ANALOG_IN_4);
```

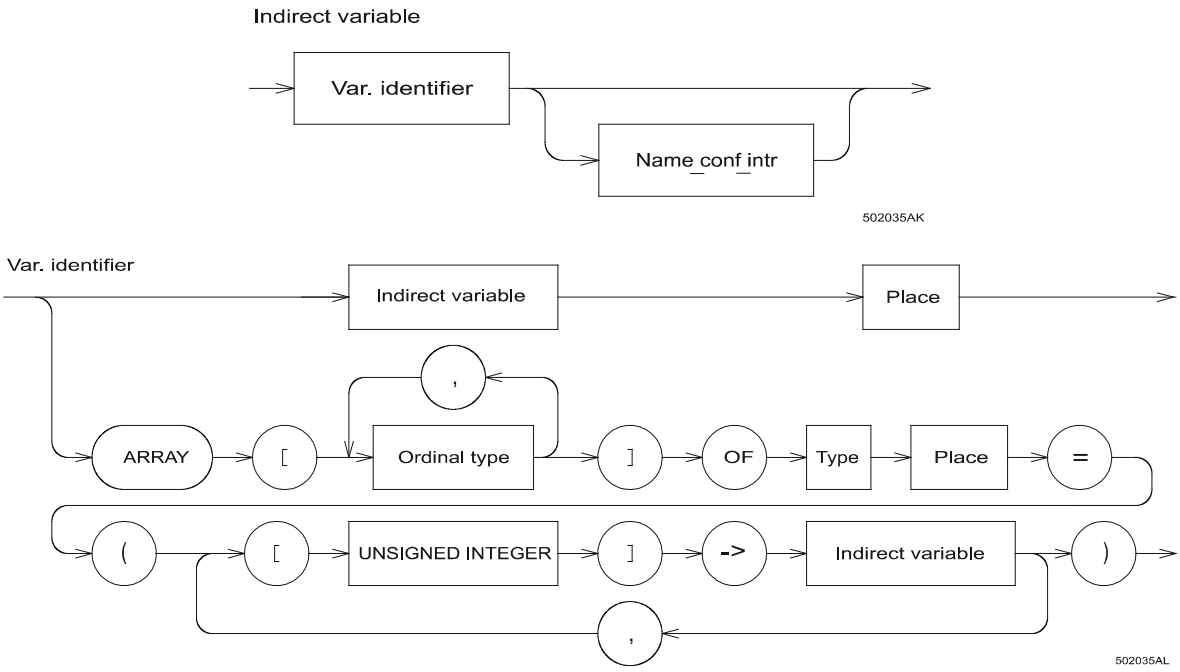
These procedure calls can be seen in the LIST file, having been substituted by the CONFIG statements. The Config procedures used are declared in the files called Config4.inc for PD 4000 and in Config5.inc for PD 5000, and should be included in a program that uses them.

8.4 Indirect variables

The previous declarations show how it is possible to declare an entire interface module. However, when writing a program, it is often more convenient to include a more detailed specification of the inputs and outputs.

Variables can be declared indirectly, which means that a new variable identifier can be declared, which will use the same location (the same memory address) as a previously declared variable, but can be accessed using the different identifier. An indirect variable identifier can also be used to access a sub-variable of a previously declared variable.

Indirect variables are declared by typing an identifier followed by -> and followed by the identifier of a previously declared variable. This previously declared variable can be of any type. When using this method of variable declaration, the newly declared identifier will have the same type and the same address as the variable on the right hand side of the -> sign.



This method of declaring variables is not a part of standard Pascal.

The **indirect variable** is a variable reference, and can be an entire structured variable, a specific component of a structured variable or a variable of simple type.

The following example demonstrates how an indirect variable declaration is achieved in practice.

```
VAR
  UPI : PD3221 AT NET: (1,64);  (* Defines a UPI slave module *)

  AgitatorCh -> UPI.Digital_IO_4;
    (* Defines the Digital_IO channel No 4 *)
  OverfillCh -> UPI.Digital_IO_6;
    (* Defines the Digital_IO channel No 6 *)

  Agitator -> AgitatorCh.FlagReg[7];
    (* Defines an Out Flag  of the Digital_ IO channel *)
```

```
  Overfill -> OverfillCh.FlagReg[6];
    (*  Defines an In Flag  of the Digital_ IO channel   *)

  TempCh -> UPI.Analog_In_1;
    (* Defines the 1st Analog Input channel *)
  Temp -> TempCh.AnalogIn;
    (* Defines the variable containing a
       value of the analog input No 1 *)
```

These variables can then be handled as ordinary variables. For example, any of the following statements will start the agitator:

```
  UPI.Digital_IO_4.FlagReg[7] := true;
  AgitatorCh.FlagReg[7] := true;
  Agitator := true;
```

And the following statement will call the *AlarmProc* procedure in the event of level detector activation:

```
  IF Overfill THEN AlarmProc;
```

Indirect arrays can be used to assemble a number of non-descriptive variables, or parts of variables, into a structured collection of meaningful identifiers. These can then be used to produce easier to control and more understandable programs.

The next example demonstrates this powerful feature.

```
VAR
    DigModule1 : PD3221 AT NET (2,51);
    DigModule2 : PD3221 AT NET (2,52);

    Valves -> ARRAY[1..MaxNumberOfValves] OF DigitalCh =
        ([1] -> DigModule1.Digital_IO_1,
         [2] -> DigModule1.Digital_IO_2,
         [3] -> DigModule1.Digital_IO_3,
         [4] -> DigModule1.Digital_IO_4,
         [5] -> DigModule2.Digital_IO_1,
         [6] -> DigModule2.Digital_IO_2,
         [7] -> DigModule2.Digital_IO_3,
         [8] -> DigModule2.Digital_IO_4);
```

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.Digital_IO_1
    NAME :'Start button for production';
  WaterTemp->AnaModule.Analog_In_1.AnalogIn
    NAME :'Water temperature';
```

The name 'Start button for production' is now associated with the variable Start, which means that the name can be used as a string when an error occurs in accessing channel 1 in DigModule. See the WHEN ERROR chapter about how to use and retrieve the declared 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.Digital_IO_1
          NAME :'Start button for production'
          CONFIG: DigitalInput;
  WaterTemp->AnaModule.Analog_In_1
          NAME :'Water temperature'
          CONFIG: Standard_Pt100;
```

8.5 Section variables

To place a variable in a defined section, i.e. FLASH, the variable definition must include a `SECTION` statement. The `PLACE` statement is in this case not the physical address of the variable, but the memory bank in which the variable is placed.
An example of a flash variable declaration is

```
  (* Identifier : Type SECTION: <SectionName> *)
  MyFlashVar : Integer PLACE: $EB SECTION: EEPROM;
```

When using FLASH-memory to store data it should be considered that reading and writing is slower than for RAM. In addition there is a (large) maximum number of times each FLASH bank can be rewritten (see manual for the device in question). This implies that data not changed too often is a candidate for a FLASH variable.

9 Pointer Types

All the previously discussed data types have the ability to hold data. A POINTER holds a different kind of information, - the location of where data are stored. Process-Pascal provides the use of pointers, as static variables, which means that the pointer variables are declared in the program and then denoted by their identifiers. They exist during the entire execution of a block (program, task, procedure or function). Pointer types cannot be allocated dynamically during program execution.

A pointer is always specific to a particular data type and it can only point to a previously declared variable of that type, or it can point to NIL. If a pointer is not initialised or pointing to NIL, the value of the pointer is undefined and an error code is generated (Error3 = $18). The standard function PointerOK can be used to test whether a pointer is valid.

A pointer holds information on a variable's SoftWire number and an offset, and occupies 14 bytes of memory.

Examples of pointer types:

```
TYPE
  RealPointer = POINTER TO REAL;

VAR
  Weight -> WeightModule.Ch1.Flow;
  Flow   -> FlowMeter.Flow;
  MeasuredValue : RealPointer;

BEGIN
  IF MeasuringModule = FlowModule THEN
    MeasuredValue -> Flow
      (* set pointer to Flow register in flowmeter *)
  ELSE
    MeasuredValue -> Weight;
      (* set pointer to Flow register in weight module *)

  Display(MeasuredValue:6:1);
    (* display flow from either flowmeter
       or weight module as measured value *)

  IF MeasuredValue > MaxFlow THEN ReduceFlow;
    (* compare MaxFlow to the value that
       MeasuredValue is pointing to *)
```

The pointer itself must be declared to reside internally, but it is permitted to point to internal as well as external variables.

A pointer type may be a part of another type, e.g. as a field in a record.

```
  MyRecordType = Record
    ASimpleVariable:  Integer;
    PointerVariable:  POINTER TO REAL;
    END;
```

10 Constants

The values 5, 1.25, -357 and TRUE, when written in a program, are called constants. A 5 in the program can only take the value 5, so 5 is a constant value. A constant cannot change value during program execution.

A constant definition introduces an identifier as a synonym for a constant. The reserved word CONST heads the constant definition part. Constant values can be a number, a constant identifier, a character, a string or a structured constant (see the STRUCTURED CONSTANT chapter).

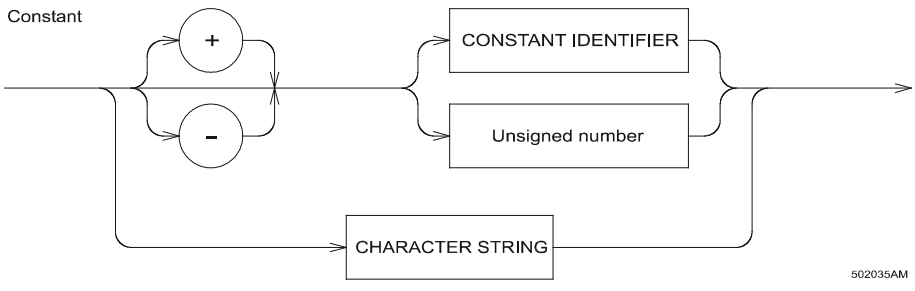
The use of constant identifiers generally makes a program more readable and acts as a convenient documentation aid. It also facilitates a grouping of machine-dependent quantities at the beginning of the program, where they can be easily changed. It is only necessary to change the value of a constant in the CONSTANT declaration part, instead of changing the constant value in all the parts of the program where it is used.

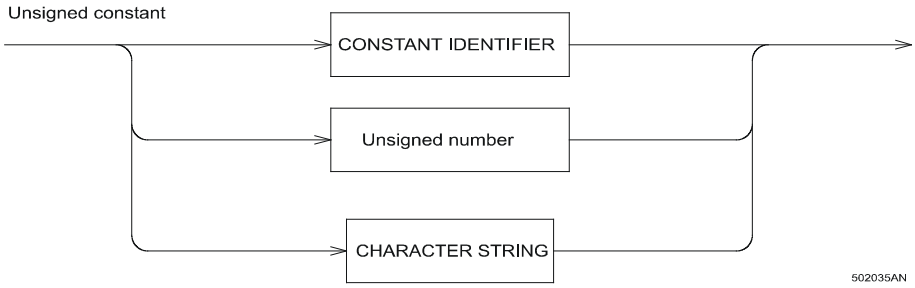
Examples of constant declarations:

```
CONST
  Max_Valves = 100;
  CursorStepX = 6;
  PageSize = 50;
  Blank = '          ';
  Manual_Set = '1.0';
  WaitTime = 2.7;
  AlarmOn = TRUE;
  CrLf = #13#10;
```

The compiler determines the type for the constant, depending on the syntax and range. However, the constant can be forced to take a specific type by using a type identifier in the declaration. e.g.:

```
PD340Type = WORD(56);
```




Unsigned constant
CONSTANT IDENTIFIER
Unsigned number
CHARACTER STRING
502035AN

11 Comments

The readability of a Process-Pascal program can be improved by inserting blanks, blank lines and notes within it. Notes can be inserted to remind the programmer (or anyone else who reads or maintains the program) as to what certain variables mean, what certain functions or procedures do, and so on. These notes are known as COMMENTS.

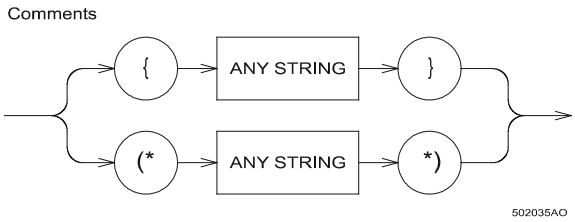
A program may contain as many comments as required and a comment may contain any sequence of characters.

A comment begins with a left curly brace { or a left parenthesis and an asterisk, (*, and ends with a matching right curly brace } or a matching asterisk and a right parenthesis, *). A comment that contains a dollar sign immediately after the opening { or (* is a compiler directive. See chapter 35.5.

You can start a comment with a left curly brace {, which signals to the compiler to ignore everything until it sees the right curly brace }. This allows for a limited form of comment nesting, because a comment beginning with a { ignores all (* and vice versa.

Example of a comment:

```
  a:=7;  (* This is a comment for the statement *)
```



It is suggested that one type of comment marker is used for program comments and compiler directives, and another type for temporary program parts. This method will prove very useful during program development and will make it easier to use comment nesting.

12 Expressions and Assignments

12.1 Expressions

An expression is a rule for calculating a value based on observing the conventional rules of algebra for left-to-right evaluation of operators and operands.

The value that is calculated depends on the value of the constants and variables that are included in the expression, and on the operators and functions that are used in the expression.

12.2 Operators

Expressions can utilise the normal arithmetic operators, logical operators and relational operators.

12.3 Arithmetic operators

The arithmetic operators are: +, -, *, and /, where * is multiplication and / is division.

These operators can be used on integer types, real types and timer types. The result type for these operations depends on the value type that is calculated. This is achieved by the use of automatic typecasting during compilation.

Examples of expressions with arithmetic operators:

```
x + y
51.8 - 2
arc * number
10 / 2.45
```

Furthermore, there are two operators, which only operate on integer operands. These are DIV and MOD.

The DIV operator performs an integer division (i.e. the value is not rounded).

Examples of the DIV operator:

Expression	**Result**
15 **DIV** 6	2
15 **DIV** 7	2
-15 **DIV** 5	-3

The MOD operator returns the remainder obtained by dividing its two operands.

Examples of the MOD operator:

expression	result
15 **MOD** 6	3
-15 **MOD** 7	-1
15 **MOD** 5	0

12.4 Logical operators

The logical operators are NOT, AND and OR. The logical operators can operate on all integer types and on BOOLEAN types.

The NOT operator performs a bitwise negation on the one operand.

Examples of the NOT operator:

Operand type	**Expression**	**Result**
byte	**NOT** $00	$FF
word	**NOT** $0101	$FEFE
boolean	**NOT** TRUE	FALSE
boolean	**NOT** FALSE	TRUE

The AND operator performs a bitwise And on the operands.

Examples of the AND operator:

Operand types	**Expression**	**Result**
byte	$55 **AND** $11	$11
word	$0202 **AND** $0101	$0000
boolean	TRUE **AND** TRUE	TRUE
boolean	TRUE **AND** FALSE	FALSE

The OR operator performs a bitwise Or on the operands.

Examples of the OR operator:

Operand types	**Expression**	**Result**
byte	$55 **OR** $11	$55
word	$0202 **OR** $0101	$0303
boolean	FALSE **OR** TRUE	TRUE

12.5 Relational operators

The relational operators are =, <>, >, <, >= ,<= and IN.

The relational operators can be used on all simple data types: boolean, byte, char, integer, longinteger, longreal, real and timer. Different types can be compared, because of the auto-

matic typecasting. Furthermore, strings can be compared according to the ordering of the extended ASCII character set. The IN operator is used to test for membership of a SET type operand. The result type is always a boolean, i.e. true or false.

Examples of relational operators:

WaitTime <= TimeOut
Weight > SetPoint
PassWord <> PassCode
InputChar IN Digits

12.6 String operator

Process-Pascal allows the + operator to be used to append two string operands. The result of the operation StrA + StrB, where StrA and StrB are of string types, will be the addition of the strings, with the first character from StrB positioned after the last character from StrA, and where the length will be the integer addition of the two string lengths. If the resulting string is longer than the result type, it will be truncated to the max string length of the result type.

The value of expressions can be converted into strings, by adding a size-specifier and a format-specifier to the expressions that are required to be converted. The syntax is as follows:

```
Str := expression : size-specifier : format-specifier
```

The size-specifier denotes the number of characters that are to represent the result of the expression, (including the decimal point, if any). The format-specifier is a value that defines how the result of the expression will be represented within the string.
If result type for the expression is TIMER, REAL or LONGREAL, the format- specifier has the following meaning:

0-..	Number of digits to be displayed to the right of the decimal point.
-1	The variable is to be represented in floating-point.
-2	The variable is to be represented with an exponent. For TIMER or REAL types, the exponent always has 2 digits and a sign. For a LONGREAL type, the exponent always has 3 digits and a sign.

If the expression result is a simple type other than TIMER, REAL or LONGREAL, the format-specifier has the following meaning:

0	Decimal representation with leading blank spaces.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If the expression contains operators, it must be enclosed in brackets.

Example:

```
(* r is a real having the value 25.61 and str is a string[35] *)
  Str := 'The value of r is : ' + r:5:2 ;

After this operation Str holds the following characters:
```

```
The value of r is : 25.61

Str:='The weight is : ' + (Weight / 1000.0):6:1 + 'T';
```

Weight is assumed to be a variable that holds the value for a weight in Kg.

12.7 Operator precedence

The operators are classified into 5 categories ordered by their precedence, with the first having the highest precedence.

The table below shows the order of operator precedence and should be referred to whenever there is any doubt as to the exact rules.

1	Unary minus	(minus with only one operand).
2	NOT operation	(boolean negation)
3	Multiplying operators	(*, /, DIV, MOD, AND)
4	Adding operators	(+, -, OR)
5	Relational operators	(=, <>, >, >, <=, >= ,IN)

13 Statements

A program is intended to perform some kind of action, using its internal and input/output data. The exact activity the program will perform, is described within statements. Statements describe algorithmic actions that can be executed. Statements are either simple or structured. Please refer to the STATEMENT syntax diagram in the chapter SYNTAX DIAGRAMS.

13.1 Simple statements

A simple statement is one that doesn't contain any other statements. Simple statements can be assignment statements, procedure statements or the empty statement. The empty statement consists of no symbols and denotes no action.

13.2 Assignment

The most fundamental of statements is the assignment statement. It specifies that a newly computed value be assigned to a variable. The value is specified by an expression. The variable may be a simple variable or an entire structured variable, located within the computer or within a module connected to P-NET. The assignment statement has the following form:

```
   identifier := expression
```

where the identifier is a variable identifier. The assignment statement is a simple statement.

Examples of the assignment statement:

```
SetPoint:= Recipe[i].Parts / 100 * Scale
DrainValve:= ON
DigitalModule.Ch20.FlagReg[7]:= OFF
Weight_Timer:= 10.0
```

13.3 Procedure statement

Another simple statement is the procedure statement, which activates the named procedure, being a subprogram specifying another set of actions to be performed on some data.

Examples of procedure statements:

```
PrintOut
Picture_11(No-Scroll)
StopMixing(MixerNo, StopCommand)
```

See the chapter "PROCEDURES AND FUNCTIONS" for more details about procedures.

13.4 Structured statements

Structured statements are constructs composed of other statements that are either to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements).

13.5 Compound statement (begin end)

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The BEGIN and END symbols act as statement brackets, and the statements are separated by semicolons. The semicolon is not a part of the statement, and is only used to separate them. An extra semicolon before an END does no harm, because an empty statement will be assumed between the semicolon and the END.

Example of a compound statement:

```
BEGIN
SetPoint:=0;
ErrorMessage:=FALSE;
PrintOut;
END;
```

13.6 Conditional statement (if then else)

The IF statement specifies that a statement will only be executed if a certain condition is true. The condition is the result of a boolean expression which produces TRUE or FALSE. If the expression produces true, then the statement following the symbol THEN is executed. If the expression produces false and the ELSE part is present, then the statement following the symbol ELSE is executed. If the ELSE part is not present, no statement is executed.

Examples of IF statements:

```
IF Sec = 60 THEN Min := Min + 1;

IF Min = 60 THEN
BEGIN
  Hour := Hour + 1;
  Min := 0
END;

IF x > z THEN
  largest := x
ELSE
  largest := z;
```

Please note that a semi-colon is never used after the boolean expression or before an ELSE, because semicolons are used to separate statements, not to end statements.

If more than one statement is to be executed after the expression, it is necessary to use the compound statement. See the second example above.

IF statements can be nested in as many levels as required. However, it is advisable not to use too many levels, because it may prove difficult to avoid getting the different IF THEN and ELSE's mixed up.

13.7 Conditional statement (case)

The CASE statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called case constants), or with the symbol ELSE. The selector can be of any ordinal type (boolean, byte, char, word or integer) but longinteger, and the ordinal values of the upper and lower bounds of that type, must be within the range -32768 to 32767. Each case constant must be associated with only one of the statements.

The CASE statement either executes the statement prefixed by the CASE constant that is equal to the value of the selector, or one prefixed with a CASE range containing the value of the selector. If no such CASE constant or CASE range exists and an ELSE part is present, the statement following the ELSE is executed. If there is no ELSE part, nothing is executed. The ordering of the case constants has no influence on the selection for execution.

The statement after the CASE constant can be a simple statement or a compound statement. When the statement has been executed, the program continues with the statement that follows the end of the CASE statement.

Examples of CASE statements:

```
CASE Number OF
  1: Figure := 2;
  2: Figure := 45;
  3, 4, 5: Figure :=0;
  6..10: Figure := 100
END;

CASE Digit OF
  '1': BEGIN
     Value :=0;
     Score :=2
       END;
  '2': Value :=3;
  '3': BEGIN
     Value :=7;
     Score :=0;
     PrintOut
   END
  ELSE PrintOut
END;
```

13.8 While statement

A WHILE 'expression' DO statement, which can be a compound statement, contains an expression that controls the repeated execution of a statement.

The result of the expression that controls the repetition must be of type boolean. The statement after the WHILE 'expression' DO, is executed zero or more times. The expression is evaluated before the statement is executed. The statement is executed as long as the expression is true, otherwise the WHILE statement terminates. If the expression is false at the beginning, the statement is not executed at all.

Because the expression is evaluated for each iteration, it is advisable to keep the expression as simple as possible.

Examples of WHILE statement:

```
While BufferEmpty(KeyBoardBuffer) DO ChangeTask;

While TO1.AnalogIn > 35.0 DO
BEGIN
  FeedBackControl;
  ChangeTask
END;
```

13.9 Repeat statement

A REPEAT statement contains an expression that controls the repeated execution of a statement sequence within that repeat statement. The general form for the repeat statement is:

```
REPEAT statement(s) UNTIL expression.
```

Note that it is a sequence of statements that the repeat statement executes.

The result of the expression controlling the repetition must be of type boolean. Unlike the WHILE statement, the statements after REPEAT are always executed at least once. Following execution of the sequence of statements, the boolean expression is then evaluated. Repeated execution is continued until the expression becomes true.

Since the expression is evaluated after each iteration, it is advisable to keep the expression as simple as possible.

Examples of REPEAT statements:

```
REPEAT
  ChangeTask;
  Difference := SetPoint - TO1.AnalogIn;
UNTIL HeatControl = OFF;

REPEAT
  Number := Number + 1;
  LoopControl := LoopControl - 1
UNTIL LoopControl = 0;
```

Note that the second example performs correctly for LoopControl > 0 when entering the loop, but if it is less than zero, the loop will repeat forever.

13.10 For statement

The FOR statement indicates that a contained statement, which can be a compound statement, is to be repeatedly executed while a progression of values is assigned to the control variable of the FOR statement.

The FOR statement has the form:

```
FOR controlvariable := initialvalue TO finalvalue DO statement
```

The control variable must be of an integer type and declared within the same scope that the FOR statement appears. The initial value and the final value must be ordinal types compatible with the control variable. The initial value and the final value can be expressions. The initial value is evaluated only once and the final value is evaluated each time, before the statement contained by the FOR statement is executed.
The statement contained by the FOR statement is executed once for every value in the range from initial value to final value. The control variable always starts off at the initial value.

A FOR statement can use TO or DOWNTO for assigning values to the control variable. When a FOR statement uses TO, the value of the control variable is incremented by one for each repetition. If the initial value is greater than the final value, the contained statement is not executed. When a FOR statement uses DOWNTO, the value of the control variable is decremented by one for each repetition. If the initial value is less than the final value, the contained statement is not executed.

The value of the control variable may be modified within the contained statement, without causing an error.

The control variable is incremented/decremented when the contained statement has been executed. Immediately after the FOR statement has been executed, the value of the control variable is undefined.

Examples of FOR statements:

```
FOR i:=1 TO NumberOfVAlves DO Valves[i].FlagReg[7]:=OFF;

FOR n:= Start TO Stop DO
BEGIN
    Recipe[n].Parts :=0;
    Recipe[n].Machine :=0
END;

FOR sl:= 50 DOWNTO 25 DO
  IF Data[sl].AlarmFlag THEN Data[sl].Counter:=0;
```

13.11 Loop statement

The LOOP statement specifies that the contained statements are to be executed repeatedly forever, and that the loop can only be broken by encountering a WHEN ERROR statement (see the INTERRUPT chapter).

The LOOP statement has the following form:

```
LOOP
     Statements
END;
```

14 Array

When handling great amounts of data, it is often convenient to store these in a structured way. An array is an example of a data structure where a group of data has been ordered into a certain pattern. An array is stored as a contiguous sequence of variables, all of the same type.

Arrays have a fixed number of components of one type, the component type. The component type follows the word **of** in the syntax for an array:

```
array_type : ARRAY[firstindex..lastindex] OF type
```

14.1 One-dimensional arrays

The index range specifies the number of elements. Valid index range specifier types include all ordinal types except longinteger and subranges of longinteger.

The index range can consist of constant identifiers or constants. The index must not include negative values.

Example of array declarations:

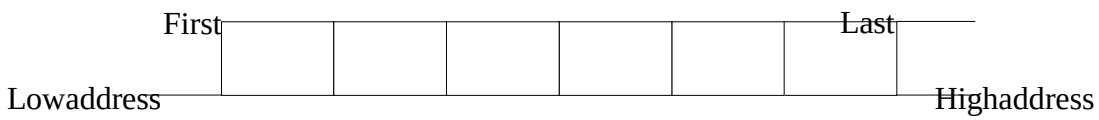
```
Data : ARRAY[1..MaxNumber] OF INTEGER;
SetPoints : ARRAY[FirstSetPoint..LastSetPoint] OF REAL;
```

An element within an array is referred to with an index, where the index can be an expression. The result of the expression must be an ordinal type, and the value should be within the specified index range. If the index value is less than the first index, then the first index is referred to. If the index value is greater than the last index, then the last index is referred to.
If the index value is out of range, an error is generated.

Examples of indexing an array:

Data[4]	denotes the fourth element in Data
Data[MaxNumber]	denotes the last element in Data

The component with the lowest index is stored at the lowest memory address, as shown below:



The values of all elements in an array can be copied to a corresponding array by using only one assignment.

Example:

```
VAR
  a,b :ARRAY[1..5] OF REAL;

BEGIN
  FOR i:=1 TO 5 DO a[i]:=0;    (* init array *)

  b:=a;            (* copy array *)
```

14.2 Multidimensional arrays

Each element within an array can itself be an array, where declared multiple index ranges will specify the number of elements, one range for each dimension of the array.

The array can be indexed in each dimension by using the values within the corresponding index range, which means that the number of elements is the total number of values within all index ranges. The number of dimensions is unlimited.

If an array's component type is also an array, the result can be treated as an array of arrays or as a single multidimensional array. The following examples are interpreted in the same way by the compiler:

```
ARRAY[1..100] OF ARRAY[1..5] OF REAL
ARRAY[1..100,1..5] OF REAL
```

An element within a multidimensional array is referred to using a number of indexes, corresponding to the number of dimensions in the array, and where each index can be an expression.

Examples of indexing a multidimensional array:

Data[2,4] denotes the fourth element in the second array element
Data[2][4] denotes the same element as above.

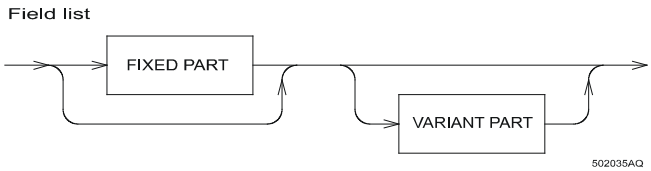
Elements within multidimensional arrays are stored in order of an increasing right-most dimension first. In the above example, this means that the values are stored in the following order: [1,1], [1,2], [1,3], [1,4], [1,5], [2,1], [2,2], [2,3], [2,4], [2,5], [3,1], [3,2], [3,3] and so on.

1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5

15 Record

A record is a structured data type, and as with the array type, it comprises a set of components. A component in a record is called a field. A field can hold values of a certain type and, unlike array types, each field can be of a different type. The type for a field can be a simple type, or a structured type, i.e. an array type or a record type.

The record type declaration specifies a type for each or collection of fields, together with an identifier that names the field. The declaration for a record begins with the symbol RECORD and terminates with the symbol END. A field list may contain a fixed part and a variant part.



The fixed part of a record type is specified within a list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

Example of a record type:

```
Square = RECORD
    x , y  : INTEGER;
    Area   : REAL;
    END;
```

15.1 Variant part

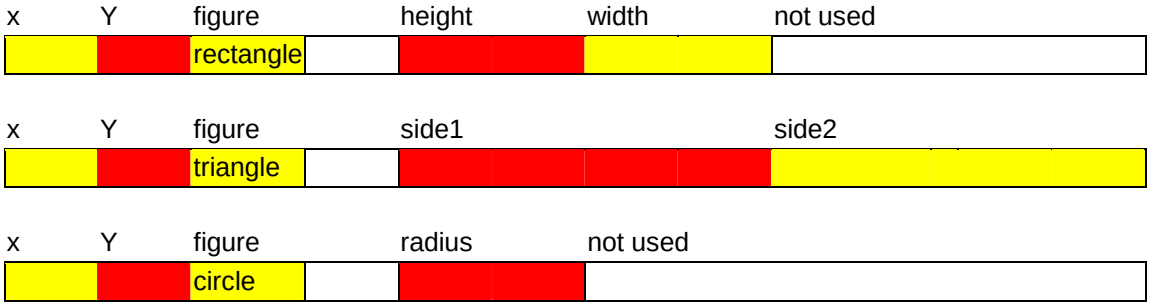
The variant part of a record type declaration provides memory space for use by more than one list of fields, so that information can be accessed in more ways than one. Each list of fields is a variant. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.

Each variant is identified by at least one constant. All constants must be distinct, and of an ordinal type compatible with the tag-field type. Variant and fixed fields are accessed in the same way.

Example of a record type with a variant part:

```
TFigure = (Rectangle,Triangle,Circle);

symbol = RECORD
  x,y : byte;
  CASE Figure:Tfigure OF
    Rectangle: (height, width : INTEGER);
    Triangle : (side1, side2: REAL);
    Circle   : (radius : INTEGER);
  END;
```

The record is shown below with the different values for the tag-field.



15.2 Accessing fields

To access a field within a record, the variable identifier for the record type is given first, followed by the field identifier. A point separates the field identifier and the record identifier.

Example of accessing a field in a record type:

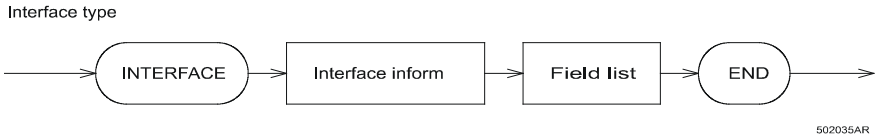
Let FORM be a record of the previously declared type SYMBOL. The fields are accessed in the following way:

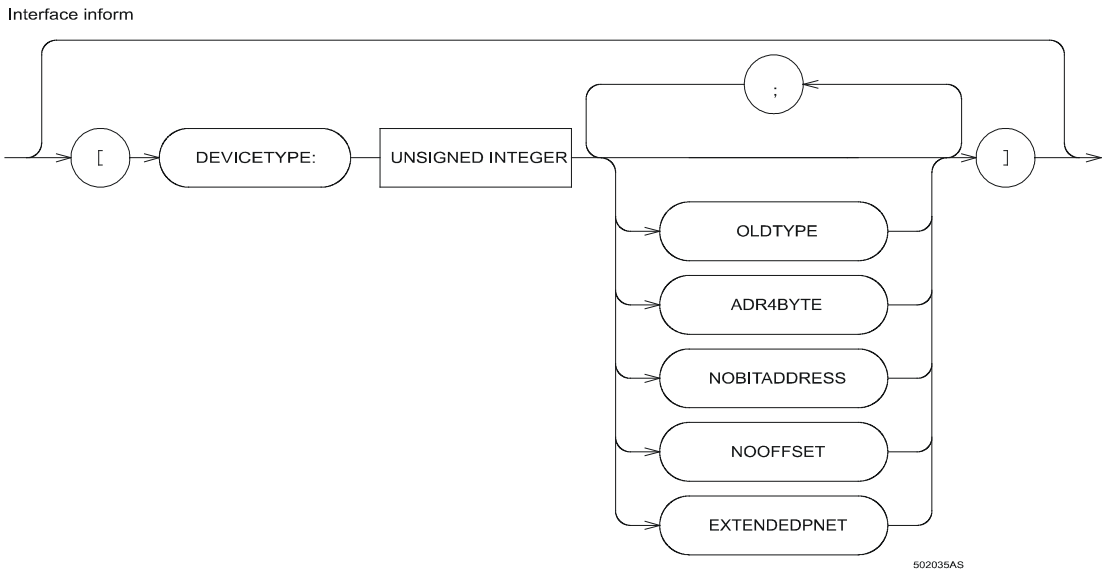
```
VAR
  Form : Symbol;

BEGIN
  Form.x := 25;
  IF Form.x = Form.y THEN ProcesSquare;
  Form.Height := 34;
  Form.Side2 := 12.22;
  Form.Radius := 200;
```

16 Interface

An interface type is used to define an interface module or a channel within an interface module, as a whole structured variable. An interface module 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 interface module.

An interface type has a fixed number of components that can be of different types. An interface type can define a CHANNEL, if all the components in the type declaration are of simple type. An interface type can define an INTERFACE module, if all the components in the type declaration are 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 interface module, defines channel 0, the second component defines channel 1 and so on.





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

The interface inform OLDTYPE denotes that the device 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 performed by the operating system in the controller during program execution, and the user does not need to consider taking any action to achieve this.

The interface inform ADR4BYTE denotes the length of the SoftWire No. / abs. address when accessing the module. The length of the address can be 4 bytes or 2 bytes, denoted by Adr4Byte or Adr2Byte respectively, where Adr2Byte is the 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 EXTENDEDPNET denotes that the module understands complex/extended P-NET addressing, e.g. a controller.

The interface inform NOOFFSETINLONG denotes that the module will not make use of the offset value in a longload or longstore command, i.e. the module calculates the offset value for itself.

Example of an interface type declaration:

```
PD3221 = INTERFACE [ DeviceType: 3221; ObjectType = 1000;
                     Capabilities = NoBitAddress, NoOffsetInLong ]
    Service        : ServiceCh;
    Digital_IO_1   : DigitalCh;
    Digital_IO_2   : DigitalCh;
    Digital_IO_3   : DigitalCh;
    Digital_IO_4   : DigitalCh;
    Digital_IO_5   : DigitalCh;
    Digital_IO_6   : DigitalCh;
    CommonIO       : CommonIO8Ch;
    Analog_In_1    : AnalogInCh;
    Analog_In_2    : AnalogInCh;
    Current_Out    : CurrentOutCh;
    PID            : PIDCh;
    Calculator     : CalculatorCh;
    PulseProcessor: PulseProcCh;
END;
```

16.1 Accessing fields

To access a field within an interface type, the variable identifier for the interface type is given first, followed by the field identifier. The field identifier and the interface variable identifier are separated by a point.

It should be noted that for variables of interface type, it is only possible to access one register at a time, and not an entire channel or module.

Example of accessing a field in a variable of interface type:

```
VAR
  TempModule : PD3221 AT NET: (1,64);

BEGIN
  IF TempModule.Analog_In_1.AnalogIn >= 45.0 THEN
    OverHeat :=TRUE;
  DigModule.Ch21.Flagreg[7]:=OFF;
  While TempModule.Analog_In_1.AnalogIn >= 35.0 DO
```

```
    ChangeTask;
```

Also see the examples in the Variable Declaration chapter about how to access variables in external devices.

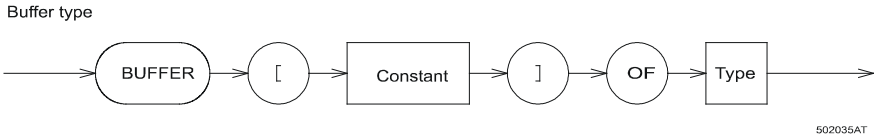
17 Buffer

A buffer can be considered as a collection of elements that are held in a queue, where elements are placed at the back of a queue when a variable is assigned to the buffer, and where the elements are removed from the front of the queue, when the buffer is assigned to a variable. This concept is known as FIFO (First In First Out).

When operating with buffers, the insertion and removal of elements applies to the entire element. This means that if the element type is a structured type, access a specific field cannot be directly accessed from the buffer. Instead, the whole element must be assigned to a variable of the same type, and then access to a particular field in that variable can be made.

Buffers have a fixed number of elements of one type, the element type. The element can be of any type except a BUFFER type, a REALDATE type or a TIMER type.

The syntax for a buffer type is:



The constant denotes the buffer size, being the max. number of elements in the buffer.

When an element has been read out from a buffer, it is deleted from the buffer and cannot be read again.

Buffers must always be initiated before they are used for the first time. This is achieved using the standard procedure InitBuffer (buffername).

Before a variable is assigned to a buffer, the program should first check whether the buffer is full. This is achieved using BufferFull (buffername), which is a standard function. The function returns a boolean, which will be TRUE if the buffer is full. If a variable is assigned to a buffer, and the buffer is already full, an error is generated, and the value will not be stored in the buffer, until an element has been removed from the buffer.

Before a buffer is assigned to a variable, the program should first check whether the buffer is empty. This is achieved using BufferEmpty(buffername), which is also a standard function. The function returns a boolean, which will be TRUE if the buffer is empty. If an empty buffer is assigned to a variable, an error is generated, and the variable will not be assigned a value until at least one element has been inserted in the buffer.

If a variable of the type BUFFER is a component of a complex variable, the buffer component variable can only be used internally within the controller.
(P-NET restriction).

Examples of buffer types:

```
  TestVarDef = RECORD
        Var1 :INTEGER;
        Var2 :REAL;
        Var3 :STRING[7];
         END;
    TestVarBuf = Buffer[10] OF TestVarDef;
```

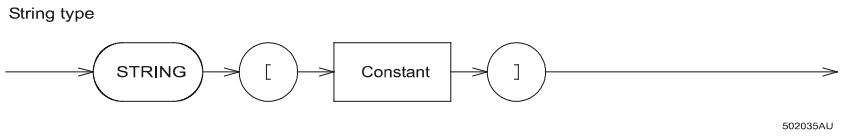
Examples of statements using buffers:

```
InitBuffer(TestVarBuf);

IF NOT BufferFull(TestVarBuf) THEN TestVarBuf:=TestVar;
(* insert an element in the buffer if it is not full *)

IF NOT BufferEmpty(TestVarBuf) THEN TestVar:=TestVarBuf;
(* remove an element from the buffer if there
   is at least one element *)

WHILE BufferEmpty(KeyboardBuffer) DO ChangeTask;

IF NOT BufferFull(Port_1.OutputBuffer) THEN Port1Output:=HeadLine;

IF NOT BufferEmpty(Port_1.InputBuffer) THEN
  BarCode:= Port_1.InputBuffer;
```

18 String

A string is a sequence of characters with a dynamic length attribute (depending on the actual character count during program execution), and a constant size attribute from 1 to 255.

The syntax of a string type:



A string can be classified as an array of characters, using the following declaration:

```
str = ARRAY[0..MaxStringLength] OF CHAR
```

Characters in a string can be accessed as components of an array.

The length attribute's current value is found in str[0].

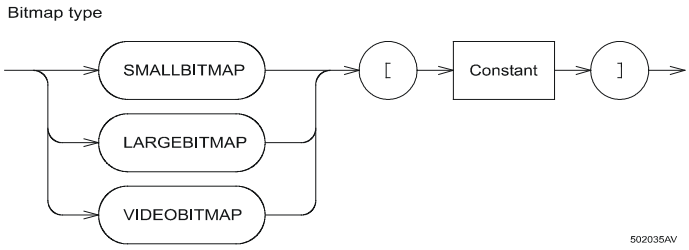
MaxStringLength is a constant in the range 0 to 255.

19 Bitmap

A bitmap defines a pixel-image as a rectangle having a width and a height. The width and height of a symbol is given in pixels, and are contained within the first elements of a bitmap type. Each of the following elements contains a byte to represent a line, or part of a line, within the pixel-image, where each bit represents the state of a pixel, starting with the most significant bit (bit7). This means that by including an ordered set of bytes within the bitmap, the inherent binary pattern will represent the pixel-image.

A bitmap type is a structured type, characterised by its component type, which is an array of booleans, and a size. A bitmap type is used to create symbols and characters that can be displayed on a screen. A charactergenerator is defined as an array of bitmap types.

Process-Pascal has three bitmap types: **smallbitmap, largebitmap** and **videobitmap**.



The size denotes the number of elements (bytes) representing the symbol.

A formula for calculating the size is given by:

```
  If width MOD 8 = 0 then a:=0 else a:=1;
  size:= ((width DIV 8) + a) * height;
```

19.1 The smallbitmap type

The smallbitmap type defines a bitmap, where the size of the symbol is less than or equal to 255 * 255 pixels (width * height).

The first byte holds the bitmap-width in pixels, and the second byte holds the bitmap-height in pixels.

A smallbitmap is referenced to the pen position on the screen at the upper left corner of the bitmap.

Example of a smallbitmap type followed by a constant declaration:

```
TYPE
  Dottype = SMALLBITMAP[1];

CONST
  Dot = Dottype($01, $01, $80);
```

The example shows a smallbitmap type with a size of '1', which means that the pixel-image will be contained within one byte. A constant defined as a smallbitmap with width '1' and height '1', is a single pixel and the pixel is on, i.e. a dot.

19.2 The largebitmap type

The largebitmap type defines a bitmap, with a size for the symbol (width * height), and an offset to a reference point.

The first two bytes hold the bitmap-width in pixels, and the third and fourth bytes hold the bitmap-height in pixels.

The fifth and sixth bytes hold an offset to a reference point in the x-direction. The seventh and eighth bytes hold an offset to a reference point in the y-direction.

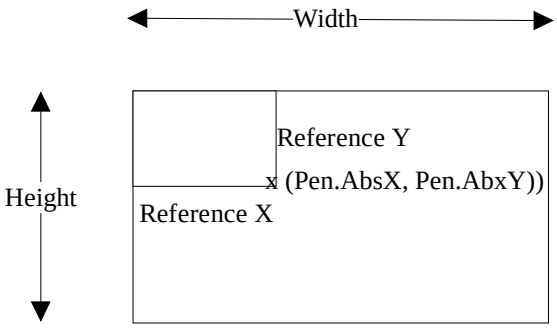
The lowest byte is the MSB for the above mentioned height, width and reference.

The bitmap will be displayed with its reference point located at the pen position on the screen (Pen.X, Pen.Y).

Example of a largebitmap type followed by a constant declaration:

```
TYPE
  Triangletype = LARGEBITMAP[4];

CONST
  Triangle = Triangletype( $00, $05, $00, $04, $00, $02, $00, $02,
                           $20, $F8, $70, $20);
```

The diagram shows the reference point (*) corresponding to the pen position for a largebitmap.



19.3 The videobitmap type

The videobitmap type is specifically used for defining video-ram, where the size denotes the capacity of the video-ram. See section SCREEN DEFINITION, for how to use videobitmap types.

20 Set

A set type provides a compact structure for recording information about the existence or combination of a collection of values having the same ordinal type.

A set type is a bit array, where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, and a set always occupies 32 bytes of RAM. A set is also a random-access structure whose elements all have the same base type.

A variable of a set type can hold from none to all values of the set.

The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range of 0 to 255.

Examples of set types:

```
Smallinteger = SET OF 0..50;
Digit = SET OF '0'..'9';
Letter = SET OF 'A'..'Z';
Colour = SET OF (red, blue, yellow, white, green, black);
```

The order of elements in a set is not significant and repetition of elements is allowed. The set (3,5..9,2,6) is equal to (2..3,5..9).

Adding new members to a set variable is simply done by adding the ordinal values to the set as follows:

```
ColourSet := ColourSet + [Red, Blue, Green];
```

Removing members from a set variable is simply done by subtracting the ordinal values from the set as follows:

```
ColourSet := ColourSet - [Yellow, Black];
```

The IN operator is used to test for membership of a SET type operand. It returns true when the value of the operand is a member of the set, otherwise it returns false.

Example:

```
IF Blue IN ColourSet THEN Display('Blue is found');

(* test if Blue is a member of the SET variable ColourSet *)
```

21 User defined Types

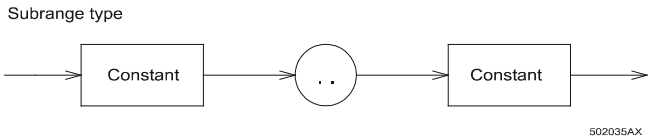
Process-Pascal has a number of pre-declared data types, which have all been described in the previous chapters. Using these types, its possible to declare new data types.

A user-defined data type is declared in the type declaration part. The name of the user-defined data type is the used identifier.

A user-defined data type can contain a previously declared type.

21.1 Subrange types

A subrange type is a range of values from an ordinal type. The definition of a subrange type specifies the least and the largest value in the subrange and includes all values in between these two values.



Both constants must be of the same ordinal type and the first one must be less than or equal to the last one.

A subrange type is mainly used to define an index range in an array structure.

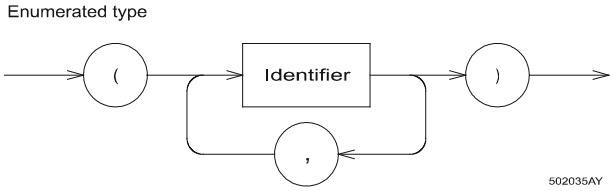
Examples of subrange types:

```
  Index20 = 1..20;  (* subrange of Integer *)
  Cap_Letter = 'A'..'Z';  (* subrange of Char *)
```

There is no index check on subrange types.

21.2 Enumerated types

Enumerated types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence in which the identifiers are enumerated. Each identifier in the list is declared as a constant for the block in which the enumerated type is declared. The data type of this constant will be of the enumerated type being declared.



An enumerated constant's ordinality is determined by its position in the identifier list in which it is declared. The first enumerated constant in a list has an ordinality of 0, the next has ordinality 1, and so on.

Example of an enumerated type:

```
  Status = ( Wait, Go, Left, Right, Stop)
```

Given these declarations, **Left** is a constant of type **Status**.

When the **Ord** function is applied to an enumerated type's value, **Ord** returns an integer that shows the position the value occupies with respect to the other values of that enumerated type. In the example above, Ord(Go) returns 1.

22 Structured Constants

A constant can be a structured type. The declaration of a constant of a structured type specifies the value of each of the elements in the structure.

The structure of a constant can be in the form of an array, a record, a set or a string type. Structured constants, which contain the types buffer or timer are not allowed.

22.1 Array constants

A declaration of an array constant specifies the values of the components. These are enclosed in parentheses and separated by commas.

Example of an array constant:

```
TYPE
  MonthsType = ARRAY[1..12] OF STRING[3];

CONST
  Months=MonthsType([1]:'Jan', [2]:'Feb', [3]:'Mar', [4]:'Apr',
                    [5]:'May', [6]:'Jun', [7]:'Jul', [8]:'Aug',
                    [9]:'Sep', [10]:'Oct', [11]:'Nov', [12]:'Dec')
```

This example defines an array constant MONTHS, which can be used to print out a 3 character string having the text corresponding to the month number.

If HEADLINE is defined as a string, the following statement

```
  HeadLine:=Months[4];
```

will produce the same result as

```
  HeadLine:='Apr';
```

Another example of an array constant is a character generator. The standard character generator, named CH6X8.CHR, is an array of bitmaps, where each character is defined as a smallbitmap. The ASCII value for the character is used as an index in the array constant.

```
TYPE
  Character6x8 = SMALLBITMAP[8];
  CG6x8 = ARRAY[$20..$9F] OF Character6x8;

CONST
   Ch6x8 = CG6x8
        ([$20]:($06,$08,$00,$00,$00,$00,$00,$00,$00,$00), (*space*)
         [$21]:($06,$08,$20,$20,$20,$20,$00,$00,$20,$00), (* ! *)
         [$22]:($06,$08,$50,$50,$50,$00,$00,$00,$00,$00), (* " *)
         [$23]:($06,$08,$50,$50,$F8,$50,$F8,$50,$50,$00), (* # *)
         [$24]:($06,$08,$20,$78,$A0,$70,$28,$F0,$20,$00), (* $ *)
         [$25]:($06,$08,$C0,$C8,$10,$20,$40,$98,$18,$00), (* % *)
         [$26]:($06,$08,$60,$90,$A0,$40,$A8,$90,$68,$00), (* & *)
         [$27] to [$9F] is not shown in this example
```

22.2 Record constants

A declaration of a record constant specifies the values of the components, separated by commas and enclosed in parentheses.

Examples of a record constant:

```
TYPE
  RecipeType = RECORD
        SesameSeed   : REAL;
        RyeFlour     : REAL;
        Water        : REAL
  END;

CONST
  RecipeDefault = RecipeType(SesameSeed : 10.0,
          RyeFlour : 65.0,  Water : 25.0);
```

A constant can also be a combination of a record type and an array, as shown in the following example:

```
TYPE
  rec1 = RECORD
    Field1 : INTEGER;
    Field2 : REAL;
  END;

  arr = ARRAY[1..2] OF rec1;

CONST
  ArrConst = rec1( [1].Field1: 0, [1].Field2: 2.34,
        [2].Field1: 4, [2].Field2: 12.40);
```

23 Procedures and Functions

When trying to resolve a problem and the program size begins to increase, it is often convenient to break it into a number of partial problems and solve each one individually. The concept of PROCEDURES and FUNCTIONS provides the means to divide each part of a problem into sub-programs.

Procedures and functions are useful in many situations, and as a guide, the following points should be considered:

1. When a certain sequence of statements is used more than once in the program, it is likely that these could be contained within a procedure. This not only conserves typing time, but also the code size in memory.
2. There should be no hesitation in formulating an action as a procedure or a function, even when it may only be called once, if doing so enhances the readability of the program. In general, shorter blocks are easier to understand than long ones.
3. General problems such as sorting, print out, weight batching and so on, should be solved in a procedure or a function. The CHANGETASK procedure can be called anywhere within a procedure or function, so a single procedure or function can remain active for hours or days without affecting the other tasks.

Procedures and functions can be global or local.

A particular global procedure or function can be called from a number of independent TASKS. This means that the same procedure or function can solve a problem for many tasks simultaneously, without affecting the other tasks (unless they are using the same global variables).

Before calling a procedure or function within a program, it is required that the procedure or function identifier be declared before it is used. This can, in some cases, be impossible. To solve this problem, procedures and functions can be "FORWARD" declared. This Forward declaration provides information to the compiler that the identifier needs to be used now, but the declaration will be found later in the program. A FORWARD declaration can be placed anywhere in the program where it is allowed to declare procedures and functions.

Example of a forward procedure declaration:

```
  Procedure CloseValve(ValveNo : byte); FORWARD;
…
  Procedure CloseValve;
  Begin
    …
  End;
```

23.1 Procedures

A procedure declaration involves defining a section of program and then associating this with a procedure identifier, so that it can be activated using a procedure call within a later statement. The declaration has the same form as a program, consisting of a heading and a block. Variables declared within a procedure are said to be local variables, and these variables are

undefined at the beginning of the statement part, whenever the procedure is activated. The local variables do not exist any further once the procedure has terminated.

Examples of procedures:

```
Procedure CloseValves;
BEGIN
InletValve[7]:=OFF;
OutletValve[7]:=OFF;
ShuntValve[7]:=OFF;
ValvesClosed:=TRUE
END;

Procedure WaitOneMinute;
VAR
    DelayTimer : TIMER;
BEGIN
    DelayTimer:=60;
    Repeat
   ChangeTask
    Until DelayTimer <= 0;
END;
```

If the procedure is required to operate on various parameters, these parameters must be introduced within the procedure heading, as part of the procedure declaration. The parameter details are inserted immediately after the procedure identifier, as a formal parameter list.

The parameter list includes the name of each formal parameter followed by its type.

A procedure statement, characterised as one that includes the procedure's identifier, together with any expected parameter values or identifiers, activates the procedure using the parameters given. These parameters are called *actual* parameters, and are substituted for the corresponding *formal* parameters that were defined in the procedure declaration. The correspondence between the formal parameters (in the procedure heading) and the actual parameters (in the procedure statement), is established by the positioning of the parameters in the list of actual and formal parameters. The inclusion of parameters therefore provides a substitution mechanism that allows a process to be repeated using a variety of arguments.

There are two kinds of parameters: value parameters and variable parameters. The kind of parameters to be used is determined by the structure of the formal parameter list in the procedure heading. Both kinds can be used within the same parameter list.

23.2 Reference parameters

When the symbol VAR heads a parameter section of the list, the parameters of this section are said to be variable parameters. In this case, the actual parameters (in the procedure statement) must be variables. The correspondingly positioned formal parameters (in the procedure heading) become synonyms for the actual variables throughout the entire execution of the procedure. Any operation involving the variable parameters is then performed directly on the actual parameters. The procedure may then change the value of these actual variables

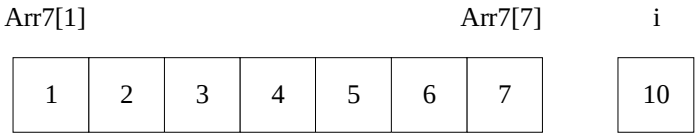
through assignments. Hence, a variable parameter can be used to represent the result of a computation.

Example to show the differences between value and variable parameters:

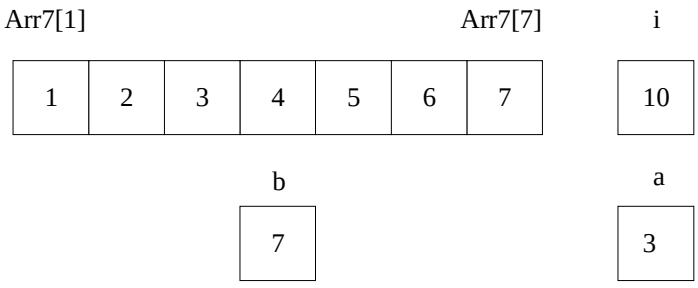
```
TYPE
  Arr7 = ARRAY[1..7] OF INTEGER;
VAR
  Arr : Arr7;
  i : INTEGER;

Procedure ValEx(a,b:integer);  (* proc. with value parameters *)
BEGIN
  a:=3;
  b:=7;          (* point B *)
END;

Procedure VarEx(VAR a,b:integer); (* proc. with variable parame-
ters*)
BEGIN
  a:=3;
  b:=7;          (* point C *)
END;

TASK Example;
BEGIN
  FOR i:=1 TO 7 DO Arr7[i]:=i;
  i:=10;          (* point A *)
  ValEx(i,Arr7[4]);
  VarEx(i,Arr7[4]);
END;
```

When the program executes, the data memory will be as follows:
a was initially a copy of *i*, but the assignment altered it to 3. Should follow “Point B:”

Point A:

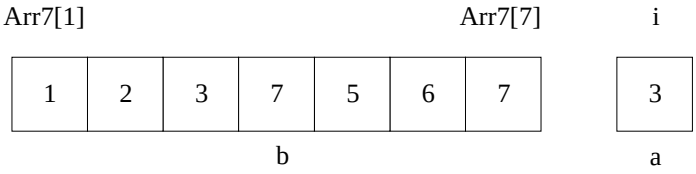


Point B:



b was initially a copy of *Arr7[4]*, but the assignment altered it to 7. Should follow “Point C:”

Point C:



Example of a procedure that controls a weight batching with a CHANGETASK procedure statement included:

```
PROCEDURE WeightBatching(    FirstSilo: INTEGER;
                             LastSilo : INTEGER;
                             VAR DataSilo : Silos;
                             VAR DoseValve: IOChannels;
                             VAR Weight: WeightChannel );
VAR
  i : INTEGER;
  DelayTimer: TIMER;
BEGIN
  FOR i:=FirstSilo TO LastSilo DO
  BEGIN
    Weight.Weight1:=0.0;
    DoseValve[i].FlagReg[7]:=On;
    REPEAT
      WeighOut:=Weight.Weight0;
      ChangeTask;
    UNTIL DataSilo[i].WeighOut <= DataSilo[i].Setpoint-
                                          DataSilo[i].Tails;
    DoseValve[i].FlagReg[7]:=Off;
    DelayTimer:=5;
    WHILE DelayTimer >= 0 DO
    BEGIN
      ChangeTask;
      DataSilo[i].WeighOut:=Weight.Weight0
    END
  END
END;
```

23.3 Value parameters

When no symbol heads a parameter section of the list, the parameters of this section are said to be value parameters. In this case, the actual parameters (in the procedure statement) must be an expression (of which a variable is a simple case). The correspondingly positioned formal parameters represent local variables within the activated procedure. This means that the local variables receive the current values of the actual parameters (the value of the expression at the time of procedure activation), as initial values. The procedure may then change the value of these local variables through assignments, but this will not affect the value of the

actual parameters. Hence, a value parameter can never be used to represent the result of a computation performed by a procedure.

A degree of caution should be applied when working with large data structures (e.g. an array with a large number of elements). The copying operation (the value parameters are copied to the local variables in the procedure), could be relatively expensive in computing time, and the amount of data storage needed to hold the copy would be as large as the value parameter itself (the array). When the procedure terminates, the data storage used by the local variables is released.

23.4 Functions

Functions are program parts (in the same sense as procedures), which compute a single ordinal or real value for use in the evaluation of an expression. The declaration has the same form as a program, having a heading and a block.

The function heading specifies the identifier for the function, the formal parameters (if any), and the function result type. The result data type for a function can only be a simple type. The variable and value parameters are discussed in the previous section PROCEDURES.

A function call is made by using the function's identifier and any actual parameters required by the function. A function call appears as an operand in an expression. When the expression is evaluated, the function is executed, and the value of the operand becomes the value returned by the function.

A function is generally used when only a single function value needs to be returned.

The block within the function declaration should contain at least one executed assignment statement that assigns a value to the function identifier. This assignment returns the result of the function. The result of the function is the last value assigned before the function terminates.

Example of a function:

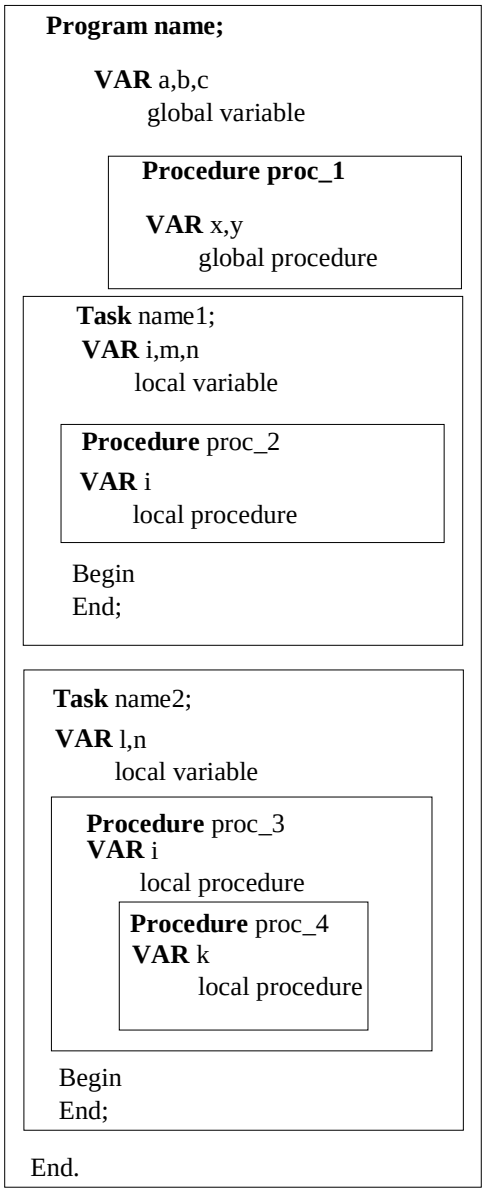
```
FUNCTION Max(VAR a:IntegerArray):INTEGER;
VAR
  i, x : INTEGER;

BEGIN
  x:=a[1];
  FOR i:=2 TO 10 DO
    IF x < a[i] THEN x:=a[i];
  Max:=x
END;
```

The function returns the largest value in an array and it is called in a statement by its identifier.

```
IF Max(Numbers) > 100 THEN .........

LargestNumber:=Max(Numbers);
```

24 Scope

In Process-Pascal, all identifiers need to be declared before they can be used or accessed. This means that an identifier is accessible within the block in which is has been declared and in any following blocks.



Only the variables **a, b, c, x and y** are available to **proc_1**.

The variables **a, b, c, i, m and n** are available to task **name1**.

The variables **a, b, c, i, m and n** are available to **proc_2**, where the variable **i** is the local variable for the procedure. The variable **i** for the task is not available to **proc_2**.

The variables **a, b, c, l and n** are available to task **name2**. **n** is not the same as **n** in task **name1**.

The variables **a, b, c, l, n and i** are available to **proc_3**.

The variables **a, b, c, l, n, i and k** are available to **proc_4**.

25 Interrupt

Process-Pascal offers facilities for generating interrupts and executing interrupt tasks. There can be 32 different interrupts, each denoted by a number in the range from 0 to 31.

An interrupt can be generated by accessing a specific global variable, which has been declared with a softwireinterrupt connection, given by an interrupt number. Interrupts can only be generated by internal variables. A task can be declared as a softwireinterrupt task with an interrupt connection. This means that an interrupt task with an interrupt number, is executed when the variable with the same interrupt number is accessed.

The interrupt condition for accessing the variable is set to “any access" as default. The interrupt condition could be specified to be one or several of the following: INTERNLOAD (the controller itself loads the variable), INTERNSTORE (the controller itself stores a value in the variable), EXTERNLOAD (the variable is loaded via the P-NET from another controller or PC) or EXTERNSTORE (a value is stored in the variable via the P-NET).

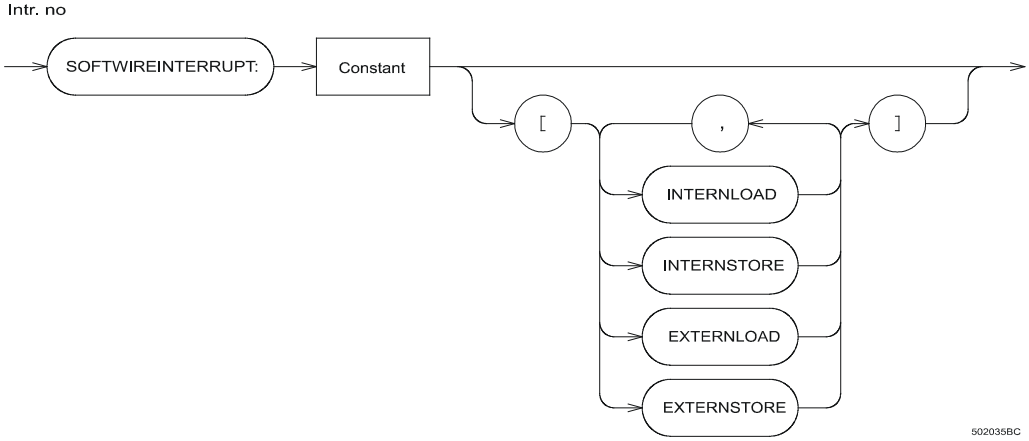
If more than one interrupt occurs at the same time, the corresponding interrupt tasks will be executed in priority, according to the interrupt number, i.e. the highest number will have the highest priority.

Example of a variable declaration with interrupt:

```
VAR
  KeyboardBuffer :Buffer[10] OF BYTE SOFTWIREINTERRUPT:0
          [INTERNSTORE, EXTERNSTORE];
```

The above declaration of the keyboardbuffer connects the variable to interrupt number 0. The interrupt condition is set for any internal or external store in the variable. The priority of this interrupt is set to the lowest priority.

Softwire interrupts can be ENABLED, i.e. allowed to interrupt, or DISABLED, not allowed to interrupt, from within cyclic tasks. ENABLE(SoftwireInterrupt) is a standard procedure to be used in cyclic tasks, to allow SoftWire interrupt tasks to interrupt the cyclic task. In all cyclic tasks, SOFTWIREINTERRUPT TASKs are ENABLED as default after a reset. If interrupt is disabled or enabled in a procedure or in a function, the interrupt status is automatically set back to the state it held before the call, after the procedure or function has been completed.



DISABLE(SoftwireInterrupt) is a standard procedure that inhibits any SoftWire interrupt in a task. The variables with an interrupt connection are not affected by DISABLE, but if a variable with an interrupt condition has been accessed while the interrupt is disabled, the corresponding interrupt task will be activated if the interrupt is enabled again.

It is possible to relate a variable of type BUFFER to another variable with an interrupt connection. Each time an interrupt related to the variable is generated, an element is stored in the buffer variable. The buffer element holds information on the SWNo which caused the interrupt (there might be more variables with the same interrupt) and an offset in bytes to the part of the variable which was accessed.

The buffer element is of the following type:

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

Example for connecting the buffer to an interrupt variable:

```
VAR
  IntBuffer:  Buffer[10] OF IntRecordType;
  DataBase :  DataBaseType SOFTWIREINTERRUPT: 3
        [ExternStore, InternStore] IntBuffer;
```

26 WHEN ERROR

Some built in facilities in Process-Pascal provide the possibility of using data that is distributed throughout a P-NET fieldbus system. To protect programs against any erroneous data that might occur externally as well as internally within a system, Process-Pascal offers an automatic error detecting system.

When using a network, such as P-NET, to communicate with interface modules or other controllers, errors can occur. The possible appearances of such errors are called INTERFACE ERRORS, and might be transmission errors relating to the network, or data errors relating to the interface modules. An error in a module can be related to the whole module or a single channel.

When executing a program, various run-time errors might be generated, which have been caused by the operator or the programmer. These errors will NOT stop program execution, but will generate 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 to handle the error condition, e.g. closing valves or stopping production. This section of program will only be executed if an error occurs in the succeeding 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:

```
Task name                                   Task Heading.
VAR
    local variable                          Local variable declaration.

Procedure
    local procedure                         Local procedure declaration.

Begin
    statements for this task    (* ref.1 *)   Task statements.

    WHEN ERROR THEN                         Error handling part.
    Begin
        statements for handling errors
    End;
    statements for this task    (* ref.2 *)   Task statements.
End;
```

If an error occurs in the first part of the task (* ref.1 *), this would not affect normal program execution, but erroneous data could be loaded and may cause problems, e.g. 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 program execution in the statement which caused the error, and move the program execution to the error handling part after WHEN ERROR.

The error handling part after WHEN ERROR THEN can end in three ways:

1. the program continues with the statements after the error handling part,
2. the program execution can RETURN to the statement where the error occurred and continue from there, i.e. proceed with the next P-code,
3. the program execution can return to the statement where the error occurred and retry the P-code.

To make the program execution return, the standard procedure *RETURN* must be called.

WARNING: When using RETURN, the program execution continues with the P-code AFTER the one in which the error occurred, and there is therefore a risk of using erroneous data in the succeeding calculations.

To make the program retry the P-code that caused the error, a standard procedure *`RetryIfLegal`* must be called.

WARNING: When using *`RetryIfLegal`*, the program execution retries the P-code in which the error occurred and there is a risk of an infinite loop, or a very slow system in the event of many errors. If using the *`RetryIfLegal`* procedure, a counter should always be implemented, and a maximum value for the counter chosen, to avoid the program locking up. The *`RetryIfLegal`* procedure can only be executed if the "WHEN ERROR program" was invoked by a transmission error.

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:

26.1 WHEN ERROR THEN [Disable]

The WHEN ERROR THEN statement activates the automatic error detecting system, which is enabled for all error conditions, i.e. enables all error bits. When an error occurs, program execution is interrupted and moved to the WHEN ERROR part. The [Disable] parameter is optional, and makes it possible to disable changetasks (interrupts from other tasks), to protect the program execution in the WHEN ERROR section. If [Disable] of ChangeTask is used in the "WHEN ERROR program", call *`Enable(ChangeTask)`* inside the WHEN ERROR block to enable ChangeTasks in the program.

A bit specification can be used to specify the error bits to be cleared, disabled, enabled, raised or tested.

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

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 generate an error code in the controller errorcode.

BITTEST

Bittest is a function used for testing error bits, generated by the automatic error detection system or the P-NET operating system. The function returns a boolean.

```
BitTest (Error [,errorbit, .., errorbit]);
```

Using Bittest on ERROR, enables a test to be performed on the error bits generated by the automatic error detection system. If the bit specification is omitted, Bittest is true if any of the error bits is true, otherwise the specified error bits are tested.

NOTE: To ensure only current error bits are tested, error bits should be cleared after the WHEN ERROR part, since the operating system will not clear these.

```
BitTest (Transmission, TransmissionErrorBit);
```

Using Bittest on TRANSMISSION, enables a test to be performed on the error bits generated by the P-NET operating system. Bittest is true if the corresponding error bit is true. The error bits correspond with the bits in the fieldvariable ErrorCode from the InterFaceErrorBuffer (see the following pages).

CLEAR

Clear is used to clear error bits, generated by the automatic error detection system. If the bit specification is omitted, all error bits are cleared, otherwise the specified error bits are cleared.

```
Clear(Error [, errorbit, .., errorbit])
```

DISABLE

Disable is used to disable all errors or specific errors, generated by the automatic error detection system. Disabling error bits will prevent the WHEN ERROR part being executed when the corresponding errors occur.

```
Disable(Error [, errorbit, .., errorbit])
```

ENABLE

Enable is used to Enable all errors or specific errors, to be generated by the automatic error detection system.

```
  Enable(Error [, errorbit, .., errorbit])
```

RAISE

Raise is used to force an error state, ignoring the automatic error detection system. An error can be raised in a specific task denoted by TaskIdentifier, or the error can be raised within the task, by calling the Raise procedure.

```
  Raise([TaskIdentifier, ] Error [, errorbit, .., errorbit])
```

26.2 ERROR REPORT

When an error is detected by the operating system, it can assign a number of parameters, contained in a report, to the global variable called InterFaceErrorBuffer. This variable is declared in the system file for the controller in question, as a buffer with 10 elements. Each element is defined as a record of 4 fields, containing information on the variable that caused the interface error. Three different errors can cause the operating system to produce this report, denoted by the following identifiers:

```
    PnetReport,    HisReport,    ActReport.
```

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 communication errors on the P-NET will insert an element in the InterFaceErrorBuffer,
HisReport	only historical errors within the accessed module will insert an element in the InterFaceErrorBuffer. *ModuleReport* can be used instead of *HisReport*
ActReport	only data errors within the accessed module will insert an element in the InterFaceErrorBuffer. *DataReport* can be used instead of *ActReport*.

The declaration of the InterFaceErrorBuffer is shown below.

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

VAR
  InterFaceErrorBuffer : BUFFER[10] OF InterFaceErrorRecord;
```

Since the variable InterFaceErrorBuffer is of type buffer, it is not possible to read a field in an element. A new variable of the same type as the elements in the buffer must be declared.

Having done so, the entire element can be assigned, and then each field in the new variable can be accessed.

Also see the example in the procedure WhenErrorRoutine or the task Error_In_InterFace, about how to use InterFaceErrorBuffer.

NOTE: When activating the automatic error detecting system and a report element is stored in the buffer, an appropriate section of program must be written to read this report element from the InterFaceErrorBuffer, in order to prevent the buffer from overflowing.

The fieldvariable **SWNo** holds a SOFTWIRE number for the interfacemodule variable that caused the interfaceerror. The standard function **VARNAME(SOFTWIRENo)** returns the stringconstant after NAME for the module variable, if it is declared. Also refer to the chapter VARIABLE DECLARATION to see how to assign a name string to a variable.

The fieldvariable **VARAddr** holds a logical address in the interfacemodule for the variable. For simple interfacemodules (I/O modules), the contents of VARAddr is a number, which is a combination of the channel number and the register number of the variable. If the module is a controller, VARAddr holds the SOFTWIRE number of the variable from the controller that caused the interfaceerror.

The fieldvariable **VAROFFSET** holds an offset to the variable (in the interfacemodule that caused the interfaceerror). The field variable VAROffset can be used to locate a variable in a complex variable.

The fieldvariable **ErrorCode** holds the errorcode relating to the interfaceerror. The field is declared as a word. The meaning of each bit is described in the specific controller's manual:

A typical structure for the error handling section is shown in the example below:

WHEN ERROR THEN [Disable]
BEGIN (* The error detection is automatically disabled when the WHEN ERROR part is entered. This prevents the error program entering a loop forever if new errors occur during the error handling procedure. *)

(* Start the error handling procedure by, for example, closing valves or stopping production. The error handling procedure should bring the process back to a well defined state, from which it can continue. A RETURN or RETRYIFLEGAL may be used to return to the program section from where the error was detected. Use Enable(ChangeTask) before leaving the WHEN ERROR block to allow changetask *)

END; (* End of the error handling procedure. The error detection is automatically enabled again, and will call this error handling section when the next error occurs. The program will continue with the statement following this "END". *)

Below is an example of the WHEN ERROR statement using RETURN:

```
WHEN ERROR THEN [Disable]  (* disable Changetask *)
```

```
BEGIN
  WhenErrorRoutine(GlobalErrorString);
  Enable (ChangeTask);
  Return;
END;
Disable(Error);
Enable(Error, PnetError, HisError, PnetReport, HisReport);
```

The example above will call the common error handling routine that returns the error information in a global error string. Please refer to the files “When_Error.inc” in the Process-Pascal library for additional details.

It is also possible to connect an interrupt to the InterFaceErrroBuffer and to call a common task, activated when an interfaceerror occurs, using a softwireinterrupt. The interrupt is connected to the variable INTERFACEERRORBUFFER, as described in previous pages. Please refer to the file “Intererr.inc” in the Process-Pascal library for additional details. The task generates a string, containing an error message. The ErrorText variable is assumed to be a global string.

The examples in the Process-Pascal library provide a common procedure to find the error information.

26.3 ERRORCODES

Some of the above mentioned errors can set an error code in the controller. Please refer to the manual for the device in question, to see a complete list of errors and the related error codes.

27 The SoftWire List

The SoftWire list is synonymous with a list containing information about the "wiring" of the plant. It has a table-like construction. The compiler converts each global identifier used in a Process-Pascal program into a number. These SoftWire numbers are used as an entry key to the SoftWire list, which contains structured information about each individual global variable and constant that is used in the particular program.

The SoftWire list contains the following information:

1. P-NET number and type of the unit - possibly internal - where the variable is stored.

2. The data type of the variable, such as integer, real, array, record, etc. It is worth noting that a record can represent a complete channel in a P-NET module.

3. The address of the variable. If the variable is available internally, the list will contain a physical address, whereas the list will contain a SoftWire number or logical address if the variable is external.

4. Name of alarm. In cases where an error is detected in a variable, for instance, in an analogue measurement channel, the name of the alarm will be included in the automatic error report (application program, written in Process-Pascal).

The SoftWire list is generated by the Process-Pascal compiler, based on the global variables declared in the Process-Pascal program. The contents of the SoftWire List can be seen in the MAP file.

During compilation, the compiler also generates a list of SoftWire numbers that are associated with all the external devices and channels that have been declared within the program. This list is stored in a global constant, declared in the PDxxxx.sys file, called PDBoxDefinition.

During starting up of a program or during configuration of a plant, it is possible to arrange to check that all the connected units are available and are equivalent to the types specified in the SoftWire list (as an optional program to include in the application program). See INITBOX4.INC or INITBOX5.INC, in the Process-Pascal library.

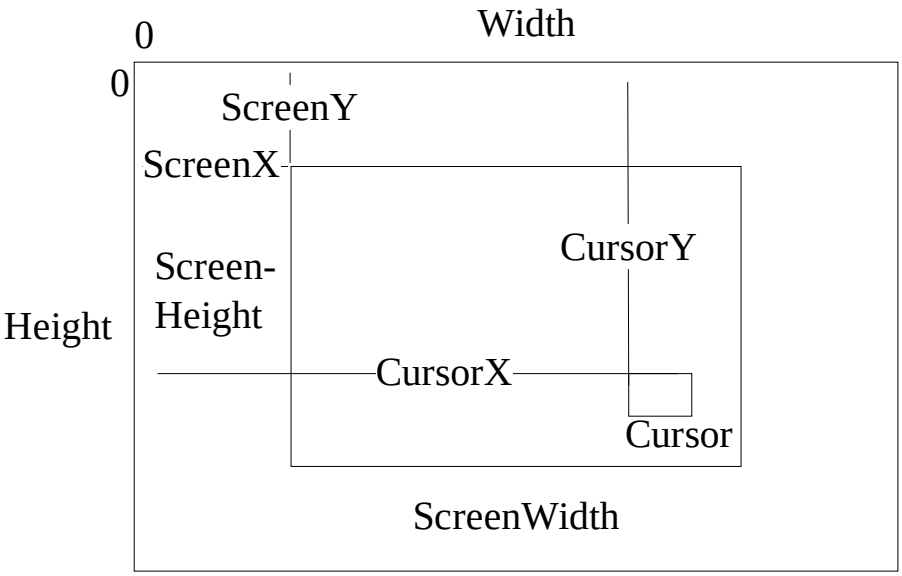
27.1 The purpose of the SoftWire list.

The SoftWire list enables the global variables that have been declared in individual controllers, to be available all over the network, thus enabling several controllers to co-operate.

When data are required to be sent from one controller to another, the identifiers in each will not be known by the other. Consequently, such data and local identifiers must be related to some numbers in a list, the SoftWire List.
The number of P-NET interface modules and the addresses of the variables can be changed within the SoftWire list without re-compiling the programs in the other controllers.

28 Screen Setup and Definition

A screen can be defined to be of a certain size (ScreenInfo.Width * ScreenInfo.Height) in pixels. The diagram shows a section of the screen, referenced to the upper left corner by ScreenInfo.ScreenX and ScreenInfo.ScreenY. This section, called the basic window, is set by



using the standard procedure SetWindow.

The system files for the various controller types declare a variable called ScreenInfo, which is a record type that holds information about the picture and the screen, and has the following structure:

```
ScreenInformationType = RECORD
            Video: BitMapPtr;       screen definition
            Width: INTEGER;
            Height: INTEGER;
            CursorX: INTEGER;      cursor definition
            CursorY: INTEGER;
            CursorForeGround: BYTE;
            CursorBackGround: BYTE;
            Cursor:  BitMapPtr;
            ScreenX: INTEGER;      basic window definition
            ScreenY: INTEGER;
            ScreenWidth: INTEGER;
            ScreenHeight: INTEGER;
        END;
```

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

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

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

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

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

ScreenX and *ScreenY* are used in controllers capable of having many windows. The standard procedure *SetWindow(x,y)* will insert x in *ScreenX* and y in *ScreenY*. Refer to elsewhere in this manual for further details.

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

To access the screen, a corresponding videoram is declared as a variable holding information about the size and address for this videoram, as shown below:

```
VAR
  Picture : VIDEOBITMAP['picturesize'] AT ADDRESS: $'adr';
```

where 'picturesize' is size for the actual screen in bytes.
'adr' is the address of the videoram for the actual screen.

Example:

```
    Screen : VIDEOBITMAP[$4000] AT ADDRESS: $4C0001;
```

The standard procedure **SETSCREEN** selects a variable of the type **VIDEOBITMAP** for the actual screen and a pointer is generated to the field variable ScreenInfo.Video.

A variable of type BitMapPtr is used by the operating system to locate the variable in memory. All variables of type BitMapPtr should not be accessed in the program by the user, because they are changed and used by the standard procedures with the name SET....., e.g. : SETVIDEO.

The standard procedure **SETVIDEO** clears the screen to the background colour, and passes its parameters to ScreenInfo.Width and ScreenInfo.Height and sends these parameters to the videocontroller. Assigning values to ScreenInfo.Width and ScreenInfo.Height has no influence on the picture, unless **SETVIDEO** is called.

The cursor position is defined by a pixel position relative to the screen origin, and is given by ScreenInfo.CursorX and ScreenInfo.CursorY. The screen origin is the pixel position (0,0) (upper left corner).

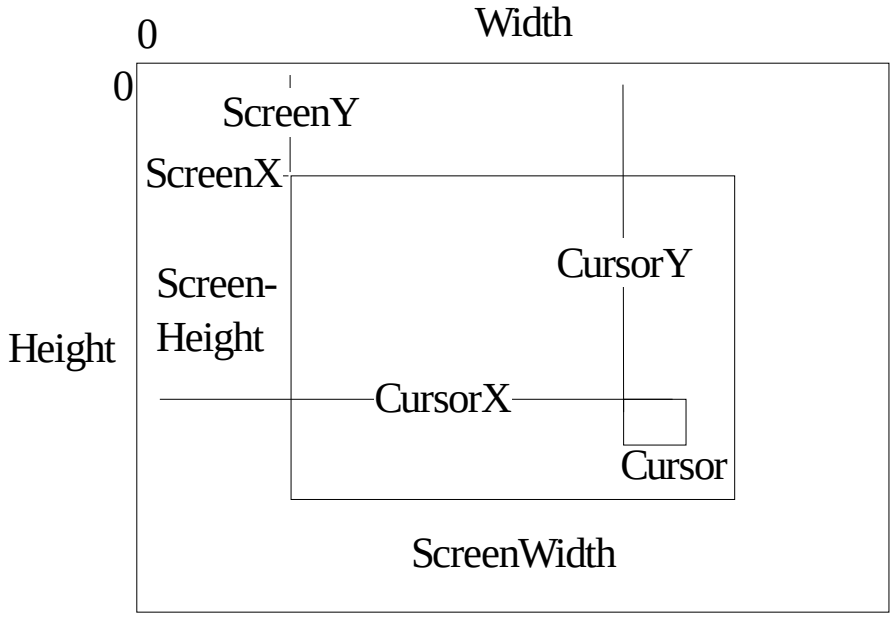
The standard procedure **SETCURSOR** selects a variable of the type bitmap for the actual cursor, and a pointer is generated to the variable ScreenInfo.Cursor. The pointer for ScreenInfo.Cursor must be generated at least once in the program, if a cursor is used within that program. The cursor is NOT used for writing onto the screen, it is only used to point to variables on the screen when the user wants to change their value from the keyboard.

The standard procedure **SETWINDOW** selects a section of the screen to be shown, and which is the basic window. The window is defined by ScreenInfo.ScreenX and ScreenInfo.ScreenY to be the upper left corner of the screen, when ScreenInfo.ScreenWidth and ScreenInfo.ScreenHeight are to be set to match the actual number of display pixels available in the used hardware.

29 Writing on the Screen

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

Anything that is written onto the screen, is referenced to particular windows. Firstly, the basic window is selected (this window is automatically opened by the selection) and following this, a number of windows can be opened from that basic window (only used within the PD5020). The following description only concerns writing within the basic window.



When it is required to write on the screen, two standard procedures are available: **Display** and **Update.** When these procedures are called, different parameters must be passed to them. Of these parameters, the first one holds information about the character generator, writing-mode and pen position.

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

Before the above mentioned procedures are called, a character generator, a foreground and a background colour and a pen position must be assigned to the variable of type Pen-InformationType.

A Pen is declared as a record type having the following structure:

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

It is possible to define as many variables of the type PenInformationType as needed. Typically, it is sensible to define one variable for each task that writes on the screen. This is necessary to ensure that no other tasks change the character generator, colour or pen position, if they interrupt the task just after setting up the required parameters.

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 SETCHARACTERGENERATOR. The figure below shows an example of the character ”A” from a 6 x 8 character generator.

The colours on the screen are selected with the field variables **ForeGround and BackGround** in variables of type PenInformationType. The colours for a pen can be set by means of the standard procedure **SETCOLORS**. ForeGround and BackGround can take the following values:

16 different colours	(Only used in a PD5020. The variety of colours are defined in the PD5000 manual. Black and White are used in the other controllers)
1 transparent colour	
inverse writing mode	

The difference between the foreground and the background writing is shown below for the character **A**.

The foreground colour corresponds to the pixels defined to be ON in the bitmap specification (symbolised by 1 below), and the background colour corresponds to the pixels defined to be OFF (symbolised by 0 below).

Also see the previous chapter - BITMAP.

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

If the foreground colour is set to transparent, only the background colour is written.

If the background colour is set to transparent, only the foreground colour is written.

If both the foreground colour and the background colour are set to transparent, nothing is written at all.

The 16 different colours are represented by 4 bits. When using the inverse writing mode, each of the 4 bits is inverted to give the resulting colour. (Only used in PD5020. Black and white can be inverted in the other controllers).

Writing to the screen is performed with reference to a pixel position. This absolute pixel position is defined by the field variables **AbsX** and **AbsY** within record variables of type PenInformationType. This pixel position is not the same as the cursor position, and should not be confused with that.

The pen position X and Y can be assigned directly, or by the standard procedures **PenToAbs**, **PenTo**, **MovePen** and **PenRefTo,** followed by the variable concerned. The variable must be of the type PenInformationType. If the variable is omitted, the variable called DefaultPen will be used.

The standard procedures for changing the pen position and the corresponding results are listed below:

MyPen is used as the pen variable.

STANDARD PROCEDURE	RESULT
`PenToAbs (MyPen,a,b)`	`MyPen.AbsX = a`
	`MyPen.AbsY = b`
`PenTo (MyPen,a,b)`	`MyPen.AbsX = MyPen.RefX + a`
	`MyPen.AbsY = MyPen.RefY + b`

MovePen (MyPen,a,b)	MyPen.AbsX = MyPen.AbsX + a
	MyPen.AbsY = MyPen.AbsY + b
PenRefTo (MyPen,a,b)	MyPen.RefX = a
	MyPen.RefY = b
	MyPen.AbsX = a
	MyPen.AbsY = b

When writing on the screen, the AbsX for the pen is changed according to the current pen position.

Example:

```
VAR
  SmallChar : PenInformationType;
BEGIN
  SetCharactergenerator(SmallChar, SmallCharGenerator);
  (* select SmallCharactergenerator to be the character
     generator when writing on the screen with the
     screeninformation variable called SmallChar *)

  SmallChar.ForeGround:= Black;  (* set foreground colour black *)
  SmallChar.BackGround:= White;  (* set foreground colour white *)

  PenToAbs(SmallChar,0,40);
  (* assign the pen position to AbsX=0 and AbsY=40 in the
     variable SmallChar *)

  Display(SmallChar,'This text is written with small characters');
  (* the text is written with characters from the character
     generator SmallCharGenerator. The first character
     is positioned with the reference point in absolute
     pixel position 0,40 *)

  PenRefTo(1,1);
  (* assign the reference point to (1,1) and the absolute
     pen pos. to (1,1) for DefaultPen *)

  PenTo(MyPen,36,8);
  (* move the pen position in MyPen to position (36,8) relative
    to the reference point (MyPenRefX, MyPen.RefY) *)

  MovePen(0,8);
  (* move the pen position in DefaultPen relative to the
     absolute pen pos. for DefaultPen *)
```

The cursor is NOT used for writing on the screen. It is only used to highlight variables on the screen when it is required to change their value from the keyboard. In other words, the cursor is used for entering and changing data via the keyboard.

The symbol for the cursor is a bitmap, and the bitmap is selected by the standard procedure **SetCursor**. In addition, this procedure will display the cursor on the screen. Typically, a cursor is only selected once within a program.

The size of the bitmap representing the cursor must not exceed the allocated memory space for the variable CursorHide. CursorHide is declared in the system file. The size of CursorHide must be at least the same size as the bitmap for the cursor.

If a colour graphic screen is used, CursorHide must be at least 4 times the size of the bitmap for the cursor. If the size of CursorHide is insufficient, an errorcode will be generated.

Before the cursor is selected, the colours for the cursor bitmap must be set. The colours are set within the record variable **ScreenInfo** by the field variables CursorForeGround and CursorBackGround. The specific colours depend on the selected cursor and the used display type. Furthermore, the initial cursor position on the display must be selected. The field variables CursorX and CursorY must be set to the absolute pixel position for the cursor.

Three different standard procedures can be used to set or move the cursor to a specific position: **CursorToAbs**, **MoveCursor** and **CursorTo**.

The standard procedures for changing the cursor position are listed below, together with the corresponding results:

MyPen is used as the pen variable.

STANDARD PROCEDURE	RESULT
`CursorToAbs(a,b)`	`ScreenInfo.CursorX = a`
	`ScreenInfo.CursorY = b`
`MoveCursor(a,b)`	`ScreenInfo.CursorX = ScreenInfo.CursorX + a`
	`ScreenInfo.CursorY = ScreenInfo.CursorY + b`
`CursorTo(MyPen,a,b)`	`ScreenInfo.CursorX = MyPen.RefX + a`
	`ScreenInfo.CursorY = MyPen.RefY + b`

30 Keyboard

The keyboards used for the different Controllers are designed as a number of user-defined keys. The key functions depend upon the type of application, and may be defined by the Process-Pascal program.

Each key has its own keycode, starting with code 1, up to number of keys on that keyboard.

The KeyBoardBuffer 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 *KeyboardBuffer* via P-NET. (An example of this can be found in VIGO for the PD 4000 Controller, by selecting the program called ‘Show PD4000 Controller’ from the right mouse menu).

The standard keyboard task 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” can be stored in the *KeyboardBuffer* via P-NET.

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.

Please refer to the specific Controller manuals for details about the number of keys available and their associated positional key codes.

31 Real-time Clock and Calendar

A CONTROLLER incorporates a real-time clock and calendar. The real-time clock and calendar is accessed through a system variable called DateTime, which is defined as SOFTWIRE 6. (In a PD 5000, the real-time clock is found in Service.DataTime).

For a PD 4000, DateTime is defined as an array of bytes in the system file.
For a PD 5000, DateTime is defined as a Record with eight fields, each defined as a byte in the system file.
Each byte or field in DateTime is defined below:

PD 5000	PD 4000	Description
Second	DateTime[0]	This byte holds the Seconds. The decimal range for seconds is from 0 to 59.
Minute	DateTime[1]	This byte holds the Minutes. The decimal range for minutes is from 0 to 59.
Hour	DateTime[2]	This byte holds the Hours. The decimal range for hours is from 0 to 23 in 24 hour mode, or from 1 to 12 in 12 hour mode.
Day	DateTime[3]	This byte holds the Day of the Week. The range for day of the week is from 1 to 7, where Sunday = 1.
Date	DateTime[4]	This byte holds the Date. The range for date is from 1 to 28/29/30/31, depending on the month. The real-time clock provides automatic leap year recognition.
Month	DateTime[5]	This byte holds the Month. The range for month is from 1 to 12, where Jan = 1 and Dec = 12.
Year	DateTime[6]	This byte holds the Year. The range for years is from 0 to 99.
Code	DateTime[7]	The following values are used to select hour mode: DateTime[7]:=$00; (* 24 hour mode *) DateTime[7]:=$80; (* 12 hour mode, AM *) DateTime[7]:=$A0; (* 12 hour mode, PM *)

The 12/24 hour mode, indicated by Code, is not supported in the PD 5000.

In 12 hour mode, DateTime[7] can be read to indicate if the time is AM or PM. It is recommended that DateTime[7] is set to one of the example values in the program, to ensure appropriate operation for the real-time clock and calendar.

The operating system in the Controller synchronises the DateTime with the real-time clock chip each time the Seconds change to 0. This can cause a problem when the time is set to 'just before midnight', 23:59:SS, where the time does not reset to 00:00:00, but changes to the previous time setting. When the time is set to 23:58:SS, it will reset to 00:00:00 correctly.

32 Accessing Undeclared Variables

In larger systems with many controllers involved, there may be a need to access variables via P-NET, which have not been declared within the controller. The variables might reside in another controller or in a newly installed interface module connected to another part of the plant, and therefore they may be unknown to a number of controllers.

In general, it's not possible to access such variables, e.g. to display a measured value, without having declared the variable first. However, the system variable called NodeList provides a way to access undeclared variables. Reference should be made to the .SYS file for the controller in question, to find the SoftWire number for NodeList.

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

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

and of course, the type of the variable.

In addition to the above, a variable cannot be accessed properly, unless the type of the module which holds the variable is also known. According to the P-NET standard, it must be specified whether 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 pointer function. The system variable NodeList is declared as an array of NodeListElement, as shown in the following:

```
NodeList: ARRAY[1..10] of NodeListElement;
```

where each element is a record type having the following structure:

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

The user can change the number of elements within NodeList. The size is declared to be 10 elements as default. The size of NodeList can be changed to match the actual application.

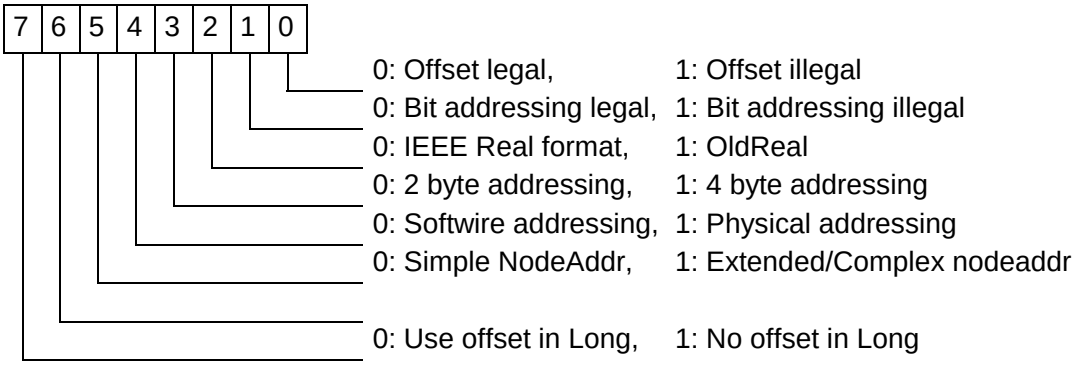
StdChannel must be set to TRUE, if the device incorporates channels that follow the structure for general purpose channel types, as defined within the P-NET Standard.

DeviceType is an integer type denoting the device type of the module that it is required to access.

NodeAddr denotes the P-NET node address of the module. NodeAddr is declared as a string, where NodeAddr[0] holds the number of bytes that are needed 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 25 characters. The port numbers and node numbers in the node address are NOT ASCII characters but hexadecimal numbers in the range 0..$7F, corresponding exactly to port numbers and node addresses.

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



Please refer to the P-NET standard for further details about addressing facilities.

Reference could also be made to the VIGO Users Manual (Ref. No. 502 086), in the Capabilities table, to find a list of values to use for *Code,* for various device types. (*Code* is the same as Capabilities in VIGO).

When an element in the nodelist, Code, DeviceType and NodeAddr has been defined, access to the module specified within the list can be achieved. Access to the module is made by means of a pointer, which is set to point to the variable required to be accessed, (see the chapter - POINTER TYPES, about how to declare and use a pointer).

The pointer being used 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(NodeListIndex, SWNo [, Offset [, BitNo]]);
```

NodeListIndex is an index to an element in NodeList and SWNo specifies the actual Softwire number from where access is wanted. If the variable is of a complex type, the Offset can be used to select a simple type variable. If the variable is a boolean array, the BitNo can be used to select the bit number. Offset and BitNo are optional, and if omitted, they are set to zero in the function call.

If Node = 0, i.e. the index is outside the NodeList array, then the pointer is set to an internal variable at the specified Softwire number. PointerToNode will also create a pointer to an internal variable if the NodeAddr field in the selected nodelist element is not specified (an empty string).

The PointerToNode function is used both in the Service and Config programs, and in the example program for WHEN ERROR. These files can be found in the Process-Pascal library.

Example of how PointerToNode could be 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;
```

33 PD GATEWAY

PD Gateway is an advanced option in Process-Pascal, which enables communication protocols used for control and regulation equipment (PLC's etc.) from different manufacturers, to be integrated directly into a Process-Pascal program.

By using the PD Gateway option, a "non P-NET compatible device" can be regarded as a P-NET interface unit, and can be accessed by the entire P-NET system, as a "normal P-NET device", including error detection and error handling. On the other hand, the P-NET system can also be regarded as a part of the "non P-NET system" as just another unit, when it is accessed from the "non P-NET" side.

The "non P-NET system" is connected to P-NET via a PD 5000 P-NET Controller, using either an RS-232 or RS-485 communication port. The PD 5000 P-NET Controller is connected to the "non P-NET system" via one of the physical ports, Port1 to Port3.

The principle is, that instead of accessing the device directly following the P-NET standard, a Process-Pascal task is started from the operating system. This program uses a virtual port - the Gateway port, and a physical port to which the device is connected. The task converts to and from the appropriate protocol for the actual device. When an answer from the device is received, the Process-Pascal program returns it to the operating system, in exactly the same manner as if the answer was received from a normal P-NET transmission. The Process-Pascal task must be declared as a SoftwireInterruptTask.

The variables that are to be accessed from the "non P-NET device", are declared in the P-NET master units where the access is required by the application, exactly as a normal variable declaration. The variables are declared as global variables with a NET address, including a net list and, depending on the communication protocol, an address specification. The net list for these variables must involve Port5 in the Gateway Controller to activate the Process-Pascal communication program.

A GatewayRecord is declared in the system file for a PD 5000, which is used to transfer data from the operating system to the Process-Pascal programme that is interfacing with the "NON P-NET DEVICE" (CALL). After the Process-Pascal program has performed the "NON P-NET" transmission, the same RECORD is used to transfer data from the Process-Pascal program to the operating system (ANSWER).

A GatewayRecord is declared in the following way:

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

The fields *NodeAddress*, *Control_Status*, *InfoLength* and *Info,* correspond to the same fields that are described in the P-NET standard. The operating system handles all NodeAddress conversion.

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

NodeAddress
This field is only used to transfer data from the operating system to the Process-Pascal program (CALL). The length of the string indicates how many numbers the string contains. If, for example, a variable is declared - AT NET:(01, 04, 02, 33) the length of the string will be 3, and the contents will be 04, 02, 33 (The first 01 indicates Port1, and is not transferred in the CALL).

Control_Status
This field is used to transfer data in CALL as well as in ANSWER. First, it is used to transfer the instruction (as it is defined in the P-NET standard) in the CALL. In the ANSWER, it is used to transfer instruction/status to the operating system - again according to the P-NET standard, thus enabling the Process-Pascal program to return "Data Error" etc. to the operating system.

InfoLength
This field is also used in both directions. It is used to determine how many databytes are involved in the transmission. For example, if the transmission is a "LOAD 8 BYTE", datalength will be 8 in CALL as well as ANSWER. If the transmission is a "STORE 4 BYTE", datalength will be 4 in CALL, and 0 in ANSWER. So, the datalength is used to tell the Process-Pascal program how many databytes are to be loaded/stored etc., and it is used to tell the operating system the number of databytes in the answer. The datalength must not exceed 56.

Info holds the entire info field, which consist of the internal address (Softwire no.), offset, bit number, data and error code as described below:

Addr
Addr is only used in CALL. Addr contains the internal address, as it is known from the P-NET standard. According to the P-NET standard, the address can be 2 or 4 bytes long, and it can be a Softwire number or a physical address. However, the address in Addr will always be sign extended to 4 bytes. If the address in the P-NET block was 4 bytes, FLAGS[0] will be TRUE.

Offset
Offset is only used in CALL, according to the P-NET standard. If there was no offset in the P-NET block, the value 0 will be transferred in Offset. If there was an offset in the P-NET block, FLAGS[1] will be TRUE.

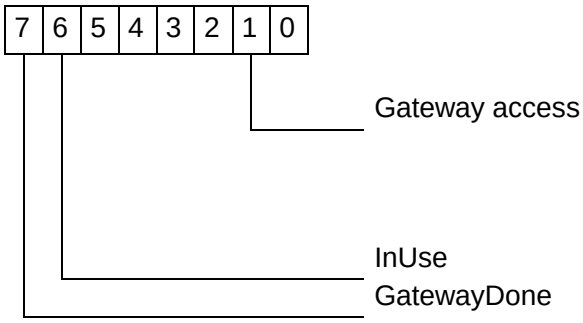
BitNo

BitNo is only used in CALL. If the transmission is a bit-transmission (indicated in the P-NET block by the MSB in a 4 byte address being SET), BitNo will contain the bit number, 0..7, and FLAGS[2] will be TRUE.

DATA
The Data field is used in CALL as well as ANSWER, and contains data as defined in the P-NET standard.

ErrorCode
This field is only used in ANSWER. If the "NON P-NET" transmission results in an error of some kind, the Process-Pascal program can return an error code in this field. Refer to the chapter "WHEN ERROR", for information on which error codes should be used. The operating system inserts 0 in this field in the CALL. If the ErrorCode is not 0 in the ANSWER, the operating system assumes there was a communication error, and does NOT use any data. Instead, the information "P-NET Error" is returned to the master.

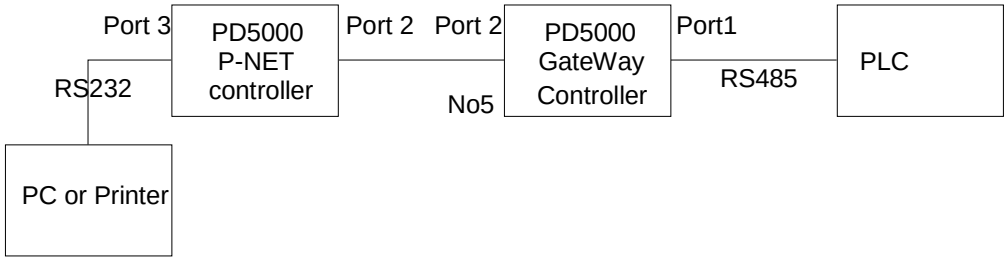
The *Flags* variable has the following meaning:



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

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

Example:



The PLC system consists of 2 PLC's, with node numbers 1 and 2.

The set up for the Gateway controller should be as follows:

Port_1.ActualMode.Protocol to be set to *DatamodeInOut*,
Gateway.GatewayInterrupt to be set to 5,
Port_1.ActualMode.BaudRate must be selected to match the PLC.

In the Gateway controller, the PLC-data are declared in this way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                    AT NET:(5,1) SOFTWIRE:$1234;
PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                    AT NET:(5,2) SOFTWIRE:$1234;
```

In the P-NET controller, the PLC-data can be declared in 2 different ways. This way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                 AT NET:(2,5,5,1) SOFTWIRE:$1234;
  PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                 AT NET:(2,5,5,2) SOFTWIRE:$1234;
```

Or this way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                      AT NET:(2,5) SOFTWIRE:aaaa;
PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                                      AT NET:(2,5) SOFTWIRE:bbbb;
```

where aaaa and bbbb are the SoftWire numbers for PLC1Data and PLC2Data declared in the Gateway controller (from the Gateway controller MAP-file).

When a LOAD transmission of PLC1Data with index $31 is initiated from the P-NET controller, the Gateway controller returns "answer comes later" to the P-NET controller. Then the following data are transferred to the Info field in the GatewayRecord in the Gateway controller:

```
  NodeAddress = $01
  Instruction = $02                (LOAD)
  DataLength  = $02                (INTEGER size)
  Addr        = $00001234
  Offset      = $0062              (index $31 * 2)
  BitNo       NOT DEFINED
  Data        NOT DEFINED
  ErrorCode   = $0000
  FLAGS       = 0/1/0/0/0/0/1/0
```

After inserting these data, the task with interrupt number 5 is started. This task must perform a transmission to the PLC, and insert the result into the Info field within the GatewayRecord:

```
  NodeAddress  Don't care
  Instruction = $82               (module error in the PLC)
  DataLength  = $02
  Address     Don't care
  Offset      Don't care
  BitNo       Don't care
  Data        = 2233            (the data from PLC integer $31)
  ErrorCode   = $0000         (no transmission errors occurred)
```

```
FLAGS          = 0/1/0/0/0/0/1/0
```

After inserting these data, the Process-Pascal program in the Gateway controller must insert TRUE in FLAGS[7]. This makes the operating system return the answer to the P-NET controller, with the data from the PLC.

Thus, the variables PLC1Data and PLC2Data can be accessed from the P-NET controller or from the Gateway controller, just as if they were variables inside a normal P-NET device.

34 Process-Pascal Reference Lookup

The following list describes all the procedures and functions in Process-Pascal that are extensions to ISO 7185 STANDARD PASCAL.

[ ] denotes that the enclosed parameter is optional. If not used, the compiler inserts a default value.

34.1 Task handling

34.1.1 CHANGETASK

```
ChangeTask;
```

ChangeTask is a procedure without parameters, and it is used to switch program execution to deal with another task. ChangeTask can be used within all types of task.

Example:

```
LOOP   (* LOOP forever   *)
  ..  (*                *)
  ..
  ChangeTask;   (* Change to another task  *)
END;
```

34.1.2 CONTINUETASK

```
ContinueTask(TaskIdentifier);
```

A SUSPENDED task can be changed to READY status, if another RUNNING task calls the standard procedure CONTINUETASK together with the appropriate task identifier, (TaskIdentifier). This will insert the task back into the appropriate task chain and when instigated, will continue from where it was last stopped or interrupted.

34.1.3 CYCLICTASK

```
CyclicTask;
```

A TIMEDINTERRUPT task and a SOFTWIREINTERRUPT task can be changed to a CYCLIC task by means of the standard procedure CYCLICTASK. Changing the task type will insert the task in the cyclic sequence, and the program execution for the task will continue until it meets a ChangeTask statement.

Example:

```
Task FlowControl TimedInterrupt:1.0;
(* the task is declared as a timed interrupt task *)
BEGIN
  . . .
  CyclicTask;
(* change the task type to a cyclic task *)
  . . .
END;
```

34.1.4 DISABLE

```
Disable(TimedInterrupt);
```
or
```
Disable(SoftwireInterrupt);
```
or
```
Disable(Interrupt);
```
or
```
Disable(ChangeTask);
```
or
```
Disable(AutoChangeTask);
```

Disable(TimedInterrupt)
Disable(TimedInterrupt) is used in a cyclic task to prevent a timed interrupt task from interrupting. All timed interrupt tasks are disabled by this procedure. Disabling the timed interrupt tasks will not change the status of these tasks. This means that they are not removed from the task chain, and when the timed interrupt tasks are enabled again, they will try to catch up with any lost time.

If a timed interrupt is disabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call occurred, following the completion of the procedure or function.

Disable(SoftwireInterrupt)
Disable(SoftwireInterrupt) is used in cyclic task, to prevent a SoftWire interrupt task from interrupting it. All SoftWire interrupt tasks are disabled by this procedure. Disabling the SoftWire interrupt tasks will not change the status of these tasks.

If a SoftWire interrupt is disabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call occurred, following completion of the procedure or function.

Disable(Interrupt)
Disable(ChangeTask)
A Disable(Interrupt) or a Disable(ChangeTask) will disable both TimedInterrupt and SoftwireInterrupt tasks (see description above).

Disable(**AutoChangeTask**)
Use `Disable(AutoChangeTask)` to prevent ChangeTask while a transmission takes place. AutoChangeTask is disabled by default. This command only affects the task in which it is stated.

34.1.5 ENABLE

```
Enable(TimedInterrupt);
```
or
```
Enable(SoftwireInterrupt);
```
or
```
Enable(Interrupt);
```
or

```
Enable(ChangeTask);
```

or

```
Enable(AutoChangeTask);
```

Enable(TimedInterrupt)

Enable(TimedInterrupt) is a standard procedure used in a cyclic task, to enable a timed interrupt task to interrupt. All timed interrupt tasks are enabled by this procedure. Timed interrupt tasks that ought to have run, will execute after enabling, and try to catch up with any lost time. Timed interrupt tasks are, by default, enabled in cyclic tasks.

If a timed interrupt is enabled within a procedure or a function, the interrupt status is automatically set back to the state it was before the call occurred, following completion of the procedure or function.

Enable(SoftwireInterrupt)

Enable(SoftwireInterrupt) is a standard procedure used in a cyclic task, to enable a SoftWire interrupt task to interrupt it. All SoftWire interrupt tasks are enabled by this procedure. Softwire interrupt tasks that ought to have run, will now continue to execute in priority order. Softwire interrupt tasks are, by default, enabled in cyclic tasks.

If a SoftWire interrupt is enabled within a procedure or function, the interrupt status is automatically set back to the state it was before the call occurred, following the completion of the procedure or function.

Enable(Interrupt)

Enable(ChangeTask)

Enable(Interrupt) and Enable(ChangeTask) will enable both Timed and SoftwireInterrupt tasks. (See description above).

Enable(AutoChangeTask)

Use `Enable(AutoChangeTask)` to enable ChangeTask while a transmission takes place. This might increase performance as other tasks can be working while the task performing P-NET communication is waiting for the response. AutoChangeTask is disabled by default. This command only affects the task in which it is stated.

34.1.6 INTERRUPTTASK

```
InterruptTask;
```

A CYCLIC task and a TIMEDINTERRUPT task can be changed to be a SOFTWIREINTERRUPT task, by means of the standard procedure INTERRUPTTASK, as long as it was originally declared as a softwireinterrupt task. The interrupt connection is set to the initial softwireinterrupt number (declared in the task head). The task continues from the next statement.

Example:

```
Task CalculateTotals SoftwireInterrupt:12;
(* the task is declared as a SoftWire interrupt task *)
BEGIN
```

```
  …
  CyclicTask; (*change task type and perform the calculations *)
  (* calculate totals *)

  InterruptTask;
  (*change the task back to a SoftWire interrupt task *)
  …
END;
```

34.1.7 MAXRUNTIME

```
  MaxRunTime(time);
```

The MAXRUNTIME for a task is initially given by a Real constant, and is declared in seconds with a resolution of 1/128 second. The default MAXRUNTIME is 300 seconds.

The Max Runtime can be changed during program execution with the standard procedure MAXRUNTIME(time), where time must be a constant or a variable, denoting the new max runtime in seconds.

Examples:

```
  MaxRunTime(NewRunTime);
```

```
  MaxRunTime(0.3);
```

34.1.8 MYTASKNO

```
  MyTaskNo:INTEGER
```

This function returns the task number of the task calling the function. The function is an integer type.

Example:

```
  Display(MyTaskNo:3:0);
```

34.1.9 RESTARTTASK

```
  RestartTask;
```

A RUNNING task can force itself to RESTART from the beginning of the task. To perform a restart of the task, the standard procedure RESTARTTASK is called. After calling RESTARTTASK, program execution will continue with the first statement within the task.

34.1.10 STOPTASK

```
  StopTask (TaskIdentifier);
```

A READY task can be changed to SUSPENDED status, if another RUNNING task calls the standard procedure STOPTASK with the appropriate task identifier, StopTask(TaskIdentifier). This will prevent the task "TaskIdentifier" from running any further, until it is changed to READY again from another task, by means of a CONTINUETASK(TaskIdentifier) statement. When a task is SUSPENDED, i.e. stopped or has come to an END for the task, it is removed

from the task chain. This means that a timed interrupt task will not attempt to catch up with any lost interrupts when starting again after it had been stopped.

34.1.11 TIMEDINTERRUPTTIME

```
TimedInterruptTime (time);
```

The time interval for running a TimedInterrupt task can be changed during program execution by means of the standard procedure TIMEDINTERRUPTTIME(time), where time can be a constant or a variable, denoting the interval time in seconds. The time is specific to the task that calls the procedure, so the procedure must be called from the timed interrupt task that is required to have its time changed. If the procedure is called from a cyclic task or a SoftWire interrupt task, it has no effect.

34.1.12 TIMEDTASK

A CYCLIC task or a SOFTWIREINTERRUPT task can be changed to a TIMEDINTERRUPT task by means of the standard procedure TIMEDTASK. Before changing the task type to TimedInterrupt, the interval time must be selected using TimedInterruptTime(time), or a default value of 255 seconds will be used. Changing the task type will not generate a ChangeTask, and the task will continue with the next statement.

Example:

```
  TimedTask;
```

34.2 Error handling

34.2.1 BITTEST

```
  BitTest (Error [,errorbit, .., errorbit]):BOOLEAN
```

or

```
  BitTest(Transmission, TransmissionErrorBit):BOOLEAN
```

Bittest is a function used for testing error bits, generated by the automatic error detection system in the P-NET operating system. (Also see WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR), ENABLE(ERROR) and RAISE(ERROR)). The function returns a boolean.

BitTest (Error [,errorbit, .., errorbit])

Using Bittest on ERROR allows a test to be made on the error bits generated by the automatic error detection system. If the bit specification is omitted, Bittest is true if any of the errors are true, otherwise only the specified errors are tested.

The error bits available for test are:

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

Example:

```
  IF BitTest(Error) THEN ErrorFound:=TRUE;
  (* test if any error bits are set *)
```

```
    IF Bittest(Error, IndexError, BufferError) THEN
      InternalError:=TRUE;
     (* test if the error was caused by an index error
        or a buffer error *)
```

BitTest(Transmission, TransmissionErrorBit)

Using Bittest on TRANSMISSION allows a test to be made on the error bits generated by the P-NET operating system. Bittest is true if the corresponding error bits are true. TransmissionErrorBit is a mask (16 bits), where the bits correspond to the error bits within the field variable ErrorCode from the InterFaceErrorBuffer (see the WHEN ERROR chapter).

Example:

```
  IF BitTest(Transmission,$0020) THEN
    ShortCircuitError:=TRUE;
  (* test if the P-NET is short-circuited *)
```

34.2.2 CLEAR

```
  Clear (Error [,errorbit, .., errorbit]);
```

Clear is used to clear error bits, generated by the automatic error detection system. (Also see WHEN ERROR, DISABLE(ERROR), ENABLE(ERROR)). If the bit specification is omitted, all error bits are cleared, otherwise the specified error bits are cleared.

The different error bits are:

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

Example:

```
Clear(Error);
  (* clear all error bits *)
Clear(Error, BufferError, IndexError);
  (* clear buffer error and index error bits *)
```

34.2.3 DISABLE

```
  Disable (Error [,errorbit, .., errorbit]);
```

Disable(Error)

Disable is used to disable errors generated by the automatic error detection system. (Also see WHEN ERROR, CLEAR(ERROR), ENABLE(ERROR)). If the bit specification is omitted, all errors are disabled, otherwise the specified errors are disabled.

The various errors to disable are:

```
PnetError,  HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,    IndexError,    ConvertError,
PnetReport,  ModuleReport,      DataReport
```

Example:

```
Disable(Error);  (*disable all errors. i.e. Disable the entire
                   automatic error detection system*)

Disable(Error, ModuleError, DataError);       (* disable module and
                  data errors detected during P-NET transmission *)
```

34.2.4 ENABLE

```
  Enable (Error [,errorbit, .., errorbit]);
```

Enable(Error)

Enable is used to Enable errors that are generated by the automatic error detection system. (Also see WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR)). If the bit specification is omitted, all errors are enabled, otherwise the specified errors are enabled.

The different errors that can be enabled are:

```
PnetError,  HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,    IndexError,    ConvertError,
PnetReport,  ModuleReport,      DataReport
```

Example:

```
Enable(Error);     (*enable all errors. i.e. Enable the
                     entire automatic error detection system *)
```

```
Enable(Error, PnetError, PnetReport);
(* enable P-NET errors detected during P-NET transmission and pro-
duce a  report. i.e. The operating system stores an element in the
InterFaceErrorBuffer *)
```

34.2.5 RAISE

```
Raise ([TaskIdentifier,] Error [,errorbit, .., errorbit]);
```

Raise is used to force an error state, ignoring the automatic error detection system. (Also see WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR) and ENABLE(ERROR)). If the bit specification is omitted, all errors will be raised, otherwise only the specified errors will be raised. An error can be raised in a specific task denoted by TaskIdentifier, otherwise the error will be raised within the task that called the procedure.

The various errors that can be raised are:

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

Examples:

```
Raise(CommTask, Error);
(* raise all errors in the communication task. i.e. Force an error
state and move program execution to the latest WHEN ERROR part next
time the task  runs *)
```

```
Raise(Error, PnetError);
(* raise P-NET error. i.e. Force an error state and move program
execution to the last WHEN ERROR part *)
```

34.2.6 RETRYIFLEGAL

```
    RetryIfLegal;
```

In a situation where the program has detected a transmission error and program execution has been moved to the WHEN ERROR part, the program can retry the P-code that caused the error. To do so, a standard procedure RetryIfLegal must be called.

WARNING: When using RetryIfLegal, the program execution retries the P-code in which the error occurred, and there is a risk of an infinite loop, or a very slow system in the event of many errors. If using the RetryIfLegal procedure, a counter should always be implemented, and a maximum value for the counter tested, to avoid program locks. The RetryIfLegal procedure can only be executed if the "WHEN ERROR program" was invoked by a transmission error.

34.2.7 RETURN

```
    Return;
```

The procedure is used to return program execution from a WHEN ERROR part. The program execution returns to the statement that caused the error and continues with the P-code immediately following. The procedure can only be called from within a WHEN ERROR THEN program part.
WARNING: When using RETURN, there is a risk of erroneous data in the succeeding calculations.

Example:

```
  When Error THEN
  BEGIN
    i:=i+1;
    IF i > MaxTries THEN Return;
  END;
```

34.3 Display handling

34.3.1 BOX

```
  Box([peninfo,] SizeX, SizeY);
```

This procedure draws a box figure as a rectangle. The box is drawn with FOREGROUND COLOR for the specified pen, starting in the current pen position. The size of the box is specified in SizeX and SizeY. When the procedure has been completed, the pen position will have moved SizeX pixels in the X-direction and SizeY pixels in the Y-direction.

Example:

```
  Box(MyPen, 25, 50);
  (* A box is drawn on the screen with a size of 25 by 50 pixels *)
```

34.3.2 BOXTO

```
  BoxTo([peninfo,] PosX, PosY);
```

This procedure draws a box figure as a rectangle from the current pen position to (PosX,PosY). PosX and PosY are relative to the upper left corner of the window. The box is drawn with FOREGROUND COLOR for the specified pen. The size of the box is determined by the current pen position and PosX and PosY. The pen position is not changed.

Example:

```
BoxTo(MyPen, 150, 240);
```

```
(* A box is drawn on the screen starting at the current pen position
and ending at position (150,240) relative to the upper left corner
of the window *)
```

34.3.3 CURSORWITHIN

```
  CursorWithIn([peninfo,] Width, Height):BOOLEAN
```

CursorWithIn checks to see whether the cursor is positioned within a certain field. The upper left corner of the field is specified by peninfo. The size of the field (width and height) is defined in pixels. If the cursor is positioned within the field, the function returns TRUE, otherwise FALSE is returned.

Example:

```
IF CursorWithIn(MyPen,20,20) THEN Found:=TRUE;
(* check if the cursor is positioned within a field of size 20 by 20
pixels relative to the current pen position*)
```

34.3.4 CURSORTO

```
CursorTo([peninfo], x, y);
```

The procedure moves the cursor position by (x,y) relative to the reference point for peninfo. If peninfo is omitted, then the pen variable within the block called DefaultPen is used. The cursor is automatically removed from the old position and appears at the new position. If a cursor has not been previously selected with the SetCursor procedure, an error is generated.

Example:

```
CursorTo(MyPen, CursorStepX, CursorStepY);
(* The cursor is moved to position (CursorStepX, CursorStepY) rela-
tive to the reference point for MyPen *).
```

34.3.5 CURSORTOABS

```
CursorToAbs (x, y);
```

The procedure sets the absolute cursor position to (x,y). The cursor is automatically removed from the old position and then displayed at the new position. If a cursor has not been previously selected with the SetCursor procedure, an error is generated.

Example:

```
CursorToAbs(CursorOffsetX, CursorOffsetY);
(* The cursor is moved to position (CursorOffsetX,CursorOffsetY) *)
```

34.3.6 DISPLAY

```
  Display([peninfo,] information: size [:format]);
```

Display is used to show a bitmap, for writing text or for displaying the value of a variable, an expression or a function on the screen. The bitmap, text or variable is displayed with the

reference point for the first character being peninfo.x and peninfo.y. When the procedure has been completed, peninfo.x will have been moved to the right by “size” (width of one digit) for numerals, or to after the last character or by the number of pixels specified by FORMAT for strings). i.e peninfo.x will be pointing to the first pixel following the field. If peninfo is omitted, then the pen variable within the block called DefaultPen will be used.

INFORMATION must be of simple type or a string or bitmap. The Information can be declared as internal to the controller, or it can be declared to be located on the P-NET network.

If the information is a **string type**, the parameter SIZE is optional and denotes the maximum **field width**, in pixels, for representing the string on the screen. If the field width is larger than the actual string, the remaining field is filled with blank pixels (background colour), otherwise the string is written until the maximum field width has been exceeded. If size is omitted, the string is written using the actual number of characters. When the procedure has been completed, PenInfo.X will have been moved SIZE pixels to the right, or to the pixel following the last written character if SIZE is omitted.

If the information is a type other than a string, SIZE denotes the **number of characters** that are to be written for the variable. The parameter FORMAT is a value defining how to present the information on the screen.

If *information* is an expression, a variable or a result of type TIMER, REAL or LONGREAL, format has the following meaning:

0-..	The number of digits to be displayed to the right of the decimal point. The default value is 2.
-1	The variable is presented in floating-point format.
-2	The variable is displayed with an exponent. For a type TIMER or REAL, the exponent will consist of 2 digits and a sign. For a type LONGREAL, the exponent will consist of 3 digits and a sign.

If *information* is a variable or a result of type BYTE, WORD INTEGER or BOOLEAN, format has the following meaning:

0	Decimal representation. This is the default value.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If *information* is a variable or a result of type CHAR or BYTE, format has the following additional meaning:

-6	ASCII representation.

If *information* is of type bitmap or string, format is not used.

Example:

```
  (* b is a BYTE, r is a REAL *)
```

```
  (* gives the following format:  *)

  b:=255;
  Display(b:3:0);   (* 255 *)
  Display(b:4:-3);  (* 00FF *)
  Display(b:8:-4);  (* 11111111 *)

  r:=12.345678;
  Display(r:7:2);   (*   12.34 *)
   Display(r:7:-1);  (* 12.3456 *)
  Display(r:7:-2):  (* 1.2e+01 *)

  Display('Process-Pascal');     (* Process-Pascal *)
  Display('Process-Pascal':60);  (* Process-Pa *)
      (* each character is 6 pixels wide *)

  Display(LargeChar,'Process-Pascal');
  Display(InputString);
  Display(ValveSymbol);
```

34.3.7 LINE

```
  Line([peninfo,] OffsetX, OffsetY);
```

This procedure draws a line to a point that is a relative distance from the current pen position. The line is drawn using the FOREGROUND colour for the specified pen, starting from the current pen position. The finishing point of the line is specified by OffsetX and OffsetY as a relative pixel distance from the current pen position. The thickness of the line is 1 pixel. When the procedure has finished, the pen position will have moved OffsetX pixels in the X-direction and OffsetY pixels in the Y-direction. The pixel at the final pen position is not drawn.

Example:

```
Line(MyPen, 50, 0);
(* Draws a vertical line on the screen with a length of 50 pixels*)
```

34.3.8 LINETO

```
  LineTo([peninfo,] PosX, PosY);
```

This procedure draws a line from the current pen position to position (PosX,PosY). PosX and PosY are relative to the upper left corner of the window. The line is drawn using the FOREGROUND colour for the specified pen. The thickness of the line is 1 pixel. The pixel at position (PosX,PosY) is not drawn. The pen position is not changed.

Example:

```
LineTo(MyPen, 250, 40);
(* A line is drawn on the screen starting at the current pen posi-
tion and ending at position (250,40) relative to the upper left cor-
ner of the window. *)
```

34.3.9 MOVECURSOR

```
MoveCursor (x, y);
```

The procedure moves the cursor position by (x,y) relative to the absolute cursor position. The cursor is automatically removed from the old position and displayed at the new position. If a cursor is not previously selected with the SetCursor procedure, an error is generated.

Example:

```
MoveCursor(CursorStepX, CursorStepY);
(* The cursor is moved CursorStepX pixels in the x-direction and
CursorStepY pixels in the y-direction *).
```

34.3.10 MOVEPEN

```
MovePen ([peninfo,] x, y);
```

The procedure moves the absolute pen position of peninfo to a position (x,y) pixels from the original absolute pen position. If peninfo is omitted, then the pen variable within the block called DefaultPen is used.

Example:

```
MovePen(36, 16);
(* The absolute pen position for DefaultPen is moved 36 pixels in
the x-direction and 16 pixels in the y-direction *).
```

34.3.11 PENREFTO

```
PenRefTo ([peninfo,] x, y);
```

The procedure sets the reference point and the absolute pen position within peninfo to position (x,y). If peninfo is omitted, then the pen variable within the block called DefaultPen is used.

Example:

```
PenRefTo(MyPen, 0, 0);
(* The reference point and the absolute pen position for MyPen is
set to the top left corner of the screen *).
```

34.3.12 PENTO

```
PenTo ([peninfo,] x, y);
```

The procedure moves the absolute pen position within peninfo to a position (x,y) relative to the reference point. If peninfo is omitted, then the pen variable within the block called DefaultPen is used.

Example:

```
PenTo(60, 8);
```

```
(* Moves the absolute pen position for DefaultPen 60 pixels in the
x-direction and 8 pixels in the y-direction, relative to the refer-
ence point *).
```

34.3.13 PENTOABS

```
  PenToAbs ([peninfo,] x, y);
```

The procedure sets the absolute pen position within peninfo to position (x,y). If peninfo is omitted, then the pen variable within the block called DefaultPen is used. The reference point is not changed.

Example:

```
  PenToAbs(20, 32);
  (* The absolute pen position for DefaultPen is set to (20,32) *)
```

34.3.14 PERFORMUPDATE

```
  PerformUpdate;
```

The procedure is used to pass a value from **InputString** (a globally declared variable into which characters from the keyboard are placed), to another variable. The procedure converts the digits in InputString and stores these data in the variable in the appropriate format. The procedure will only pass the value to the variable if the cursor is positioned within the field of the displayed variable, and only if that variable has been displayed on the screen using the procedure UPDATE. If the cursor is not positioned within the field of a variable, the procedure has no effect.

Example (keyboard task):

```
  InputString  := '123';
  Performupdate;
(*Will pass the numeric value to the variable pointed to by the cur-
sor, and display it in the format defined in UPDATE*)
```

34.3.15 SETCURSOR

```
  SetCursor(CursorRef);
```

The procedure selects CursorRef to be the current cursor.
CursorRef could be defined as a LARGEBITMAP, where ReferenceX and ReferenceY denote an offset from the upper left corner of the cursor to a reference point, which is the point that must be inside the field on the screen when a variable is to be updated. If CursorRef is defined as a SMALLBITMAP, the upper left corner of the cursor is used as the reference point (0,0). Before SetCursor is called, the colours and its desired position on the screen should be selected. Calling SetCursor will automatically display the cursor according to its previous settings. If a cursor had already been previously selected with SetCursor, it will be removed from the screen before the new cursor is displayed.

Example:

```
  SetCursor(BlackCursor);
```

34.3.16 SETVIDEO

```
SetVideo(peninfo, Width_of_screen, Height_of_screen);
```

The procedure clears the screen by setting the entire videoram to background colour. Furthermore, it passes the width and height for the screen to ScreenInfo.Width and ScreenInfo.Height in the video controller. The cursor is automatically displayed on the screen again after clearing.

It is recommended that this procedure be used to clear the screen when selecting between various screen layouts, since it will clear information about valid cursor positions within an update field on the previous display from the operating system. This will prevent updating data belonging to a previous display after selecting the new display. In the PD5020 controller, all windows are automatically closed.

Example:

```
SetVideo(DefaultPen, ScreenWidth, ScreenHeight);
```

34.3.17 UPDATE

```
Update([peninfo,] variable: size [: format] [: UpdateValid]);
```

This is a very powerful procedure, which can be used to change the value of a variable from the keyboard. It combines the ability to continuously display the current value of a variable on the screen and to assign a new value to this variable from the keyboard. The variable can be declared as an internal variable within the controller, or it can be declared as located somewhere within the P-NET network.
It is only possible to change or update a variable, if the cursor is positioned within the field on the screen where the variable is shown. If the cursor is not inside the field, the variable cannot be changed from the keyboard, and **Update** operates like the standard procedure **Display**. (Som i PP 4.0-manualen, ellers er det ikke forståeligt)

Update can be used on simple types and string types. The variable is displayed with the reference point for the first character at (peninfo.x, peninfo.y).

If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

If the variable is a **string type**, the parameter SIZE denotes the maximum **field width, in pixels**, for presenting the string on the screen. If the field width is larger than the actual string, the remaining field is filled with blank pixels, otherwise the string is written until the maximum field width has been exceeded. When the procedure has been completed, PenInfo.X will have been moved SIZE pixels to the right.

If the variable is not a string type, SIZE denotes the **number of characters** that are to be displayed as the variable. The parameter FORMAT is a value for defining the way that information is presented on the screen. When the procedure has been completed, peninfo.x will have been moved by (size * width of one digit) to the right. (peninfo.x will be pointing to the first pixel after the field).

If the variable is of type TIMER, REAL or LONGREAL, format has the following meaning:

0-..	The number of digits to be displayed to the right of the decimal point. 2 is the default value.
-1	The variable is presented in floating-point format.
-2	The variable is displayed with an exponent. For a type TIMER or REAL, the exponent is always displayed with 2 digits and a sign. For a type LONGREAL, the exponent is always displayed with 3 digits and a sign.

If the variable is of type BYTE, WORD INTEGER or BOOLEAN, format has the following meaning:

0	Decimal representation. This is the default value.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If the variable is of type CHAR or BYTE, format has the following additional meaning:

-6	ASCII representation.

If the variable is of type string, format is not used.

UpdateValid is a boolean or a booelan expression. If *UpdateValid* is ON, the variable can be changed from the keyboard. If *UpdateValid* is OFF, the procedure operates like the procedure **Display**. *UpdateValid* can be any boolean or boolean expression defined by the user and is independent of the cursor position. The default value for *UpdateValid* is ON.

Examples:

```
  (* b is a BYTE, CharVal is a CHAR, r is a REAL *)
  (* Str is a string[20] and PassWordOK is a boolean *)
  (* The values can be presented in the following forms:  *)

  b:=255;
  Update(b:3:0);   (* 255 *)
  Update(b:4:-3);  (* 00FF *)
  Update(b:8:-4);  (* 11111111 *)

  Update(CharVal:1:-6);
  (* the first character from the input string is moved directly to
the variable without any conversion *)

  r:=12.345678;
  Update(r:7:2);   (*   12,34 *)
   Update(r:7:-1);  (* 12,3456 *)
  Update(r:7:-2);  (* 1,2e+01 *)

  Update(Str:120:PassWordOK);
```

34.4 Miscellaneous

34.4.1 AND

```
And(variable, expression);
```

This procedure performs a logical AND instruction directly to the variable using the expression parameter. The variable can be declared as internal or external.

Example:

```
And(FlagReg, $55);
  (* The FlagReg variable is And'ed with $55 *)
```

34.4.2 BUFFEREMPTY

```
BufferEmpty(buffername):BOOLEAN
```

Before a buffer is assigned to a variable, the program must check whether the buffer is empty. This is achieved using BufferEmpty(buffername). The function returns a TRUE boolean if the buffer is empty. If an empty buffer is assigned to a variable, an error is generated, and the value of the variable will be undefined.

Example:

```
IF BufferEmpty(KeyboardBuffer) THEN ChangeTask
```

34.4.3 BUFFERFULL

```
BufferFull(buffername):BOOLEAN
```

Before a variable is assigned to a buffer, the program must check whether the buffer is full. This is achieved using BufferFull(buffername). The function returns a TRUE boolean if the buffer is full. If a variable is assigned to a buffer and the buffer is already full, an error is generated, and the value will not be stored in the buffer.

Example:

```
While BufferFull(KeyboardBuffer) DO ChangeTask;
```

34.4.4 CONVERT

```
Convert(variable):integer_type
```

or

```
Convert(variable):boolean_array_type
```

A special typecasting can be performed to translate integer types into boolean array types and vice versa, using the CONVERT function. The function is called with the variable of the type that is to be converted, and then the function transforms it into the other type.

If the function is called with an integer type, then the result type must be a boolean array type, having a size in bytes corresponding to the integer type.

If the function is called with a boolean array type, then the result type must be an integer type, having a size in bytes corresponding to the boolean array type.
NOTE: the boolean array must start with index 0.

Example:

```
  VAR
   Bit8Var  : ARRAY[0..7] OF BOOLEAN;
   Bit16Var : ARRAY[0..15] OF BOOLEAN;
   ByteVar  : BYTE;
   IntVar   : INTEGER;

BEGIN
ByteVar:=Convert(Bit8Var);
  (* convert a 8 bit boolean array to a byte *)

Bit16Var:=Convert(IntVar);
  (* convert an integer to a 16 bit boolean array *)
```

34.4.5 INITBUFFER

```
InitBuffer(buffername);
```

Buffers must always be initiated before they are used for the first time. This is instigated by using the standard procedure InitBuffer(buffername).

Example:

```
InitBuffer(KeyboardBuffer);
```

34.4.6 MYSWNO

```
MySWNo(identifier, SoftWireNo, VarOffset);
```

This procedure finds the SoftWire number of a global constant or variable and returns this number in SoftWireNo. If the identifier denotes a field in a complex variable, the actual offset for this field, in bytes, is returned in VarOffset. SoftWireNo and VarOffset must both be integer type variables.

Examples:

```
MySWNo(Recipe[Last].Stirring, SWNo, VarOffset);

MySWNo(DigitalModule, SWNo, VarOffset);
```

34.4.7 OR

```
Or(variable, expression);
```

This procedure performs a logical OR instruction directly to the variable using the expression parameter. The variable can be declared as internal or external.

Example:

```
Or(FlagReg, $80);
  (* The FlagReg variable is Or'ed with $80 *)
```

34.4.8 POINTEROK

```
PointerOK(ptr):BOOLEAN
```

This function is used to test whether a pointer is set to point at a variable of the correct type. The function returns TRUE if Ptr is valid. The function is a boolean type.

Example:

```
IF NOT PointerOK(MyPtr) THEN MyPtr -> MyDefaultValue;
```

34.4.9 POINTERTONODE

```
PointerToNode(Node, SWNo [, Offset [, BitNo]])
```

When it is required to access variables that are not declared within the controller, PointerToNode is used to set a pointer to point at the variable specified by the parameters in the function call. Node denotes an index for a node element in the NodeList, which specifies the node address for the module and the module type. SWNo denotes the Softwire number for the variable it is required to access in the module specified by Node. Offset denotes an offset in bytes, if access is to be made to a complex variable. BitNo denotes a bit number, calculated from the Offset. See the chapter ACCESSING NOT DECLARED VARIABLES, for further information.

Example:

```
MyPtr-> PointerToNode(NodeNo, SoftwireNo, IndexNo*4);
```

34.4.10 STRVAL

```
StrVal(str[:mode]) : Result
```

This function converts the numeric value depicted by a string type expression, STR, into a numeric equivalent. MODE denotes the format in which the string will be represented and it is an integer type. The values for Mode are described below. If Mode is omitted, the default value is 0. If the character sequence in the string is illegal according to the specified mode, an error is generated (ConvertError), and the result is stored as 0 (zero). If the string represents a real value including a decimal point, the character for the decimal point must correspond with the selected CountryCode, otherwise an error will be generated. (Please refer to the manual for the controller in question, for further details about CountryCode). The function type is the same type as that on the left side of the statement, or the same type as the other operands within the expression.

If the result type of the function is of type TIMER, REAL or LONGREAL, mode is not used.

If the result type of the function is a simple type other than TIMER, REAL or LONGREAL, mode has the following meaning:

0	The string is represented in decimal with leading spaces.
-3	The string is represented in hexadecimal.
-4	The string is represented in binary.
-5	The string is represented in decimal with leading zeros.

Example:

```
  RealRead:=StrVal(LoadString);
(*convert a loaded string from external equipment to a real value*)
```

34.4.11 TAB

```
  Tab (Position [,Char]):STRING
```

This function is used to fill out a string with a specified character, up to a selected position within that string. If the string length is less than POSITION, the string will be appended with the character CHAR until the string length is equal to POSITION, otherwise the function does nothing. If CHAR is omitted, a space character will be used as default.

Example:

```
  Str:='Setpoint ' + Tab(25,'.') + SetPoint:6:1 + ' kg';
```

If SetPoint is equal to 135.2 kg, Str will be as follows:

```
  Str = 'Setpoint ............... 135.2 kg'
```

34.4.12 TESTANDSET

```
  TestAndSet(bool):BOOLEAN
```

This function is used to test a boolean value, bool, and, if the boolean value is FALSE, SET it to TRUE. If the boolean value is already TRUE, the function returns TRUE and the boolean is not affected. The function returns the value of the boolean as a result of the TEST part of the function. This function can be used to test a variable to see if it is FALSE (free), and if so, then set it to TRUE (reserve it), all in one instruction. This facility is very useful in multitasking systems, when many tasks have access to the same variables. BOOL must be a global variable. The function is a boolean type.

Example:

```
  While TestAndSet(PrinterReserved) DO ChangeTask;
  (* Wait until the printer is available and then reserve it *)
```

34.4.13 VAL

```
  Val(x)
```

This function is used to change the value of the expression x to another type. The expression x must be an ordinal type. The function type is the same type as that on the left side of the statement, or the same type as the other operands in the expression.

Example:

```
TYPE
  ColourType = (Red, White, Green, Blue, Black, Yellow);
VAR
  Colour : ColourType;
BEGIN
  Colour:=Val(4);  (* Colour is set to Black *)
```

34.4.14 VARNAME

```
VarName (SoftWireNumber, [Offset]):STRING
```

This standard function is used in connection with the automatic error detection system, and it returns the string constant that has been declared after NAME within the global variable declaration part. The function is called with the number that is the SoftWire number of the variable. The fields ModuleSWNo and VarSWNo in an element from the InterFaceErrorBuffer hold the SoftWire numbers for the variables that caused the error. If the variable at SoftWire-Number has been declared without a NAME, the function returns an empty string.

OFFSET is used to get the NAME defined for a channel in an interface module, where the channel number is used as the offset. The default value is 0.

Example:

```
ErrorString := 'Error in ' + VarName(ErrorBlock.SWNo);

ChannelNo:= ErrorBlock.VarAddr DIV $10;
ErrorText:= VarName(ErrorBlock.SWNo, ChannelNo);

Str := VarName(ErrorBlock.SWNo);
```

34.5 Standard constants

OFF has the same meaning as **FALSE**.

ON has the same meaning as **TRUE**.

MAXINT = 32767, the maximum integer value.

NIL is a constant for a pointer. A pointer value set to NIL does not point to anything.

35 Comparing Process-Pascal with ISO 7185 Standard Pascal

This list compares Process-Pascal with 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).

35.1 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 100 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, there 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 permitted. 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 for 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 for 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, a number of 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 the Application folder, 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.

35.2 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 has been especially 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 buffer's 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 non-printable 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 initialised 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, within a block.

An identifier can contain underscore characters (_) after the first character. Integer constants can be written in hexadecimal notation, where 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.

35.3 Standard Procedures and Functions

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

AlarmHornOnOff
AlarmPulseOn
And
BitTest
Box
BoxTo
BufferEmpty
BufferFull
ChangeTask
Clear
ClearWindow
CloseWindow
ContinueTask
ContrastControl
Convert
ConvertErrorCode
CursorInWindow
CursorTo
CursorToAbs
CursorWithin
CyclicTask
Disable
Display
DisplayOnOff
Enable
InitBuffer
InitPort
InitPort1
InterruptTask
LedOnOff
LightControl
LightOnOff
Line
LineTo
MaxRunTime
MoveCursor
MovePen
MySWNo
MyTaskNo
OpenWindow
Or
PCodeCall
PenRefTo
PenTo
PenToAbs
PerformUpdate
PointerOk
PointerToNode
Raise
RestartTask
RetryIfLegal
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

35.4 Reserved words in Process-Pascal

ADDRESS
AFTER
AND
ARRAY
AT
BEGIN
BITMAP
BOOLEAN
BUFFER
BYTE
CASE
CHANNEL
CHAR
CONFIG
CONST
CYCLIC
DEFINE
DIV
DO
DOWNTO
ELSE
END
ERROR
EXTERNAL
FALSE
FOR
FORWARD
FROM
FUNCTION
GOTO
IF
IMPORT
IN
INITIALIZE
INTEGER
INTERCOM
INTERFACE
INTERNAL
INTERRUPT
LABEL
LARGEBITMAP
LONGINTEGER
LONGREAL
LOOP
MAXINT
MOD
MODULE
NAME
NET
NIL
NOT
OF
OFF
ON
OR
PLACE
POINTER
PROCEDURE
PROGRAM
READY
REAL
REALDATE
RECORD
REPEAT
RETURN
RUNTIME
SET
SMALLBITMAP
SOFTWIRE
SOFTWIREINTERRUPT
STRING
SUSPENDED
TASK
THEN
TIMEDINTERRUPT
TIMER
TO
TRUE
TYPE
UNTIL
UNUSED
VAR
VIDEOBITMAP
WHEN
WHILE
WITH
WORD

35.5 Compiler directives

Compiler directives control some of the compiler's functions, and are introduced as comments having a special syntax. Process-Pascal permits compiler directives to be inserted wherever comments might be used.

A compiler directive starts with a $ as the first character after the opening delimiter. The $ is immediately followed by a letter that designates the particular directive.

{$L-}
LISTING OFF
This directive is a switch directive that turns OFF the listing of the source file and error messages in the .LST file.

{$L+}
LISTING ON
This directive is a switch directive that turns ON the listing of the source file and error messages in the .LST file.

{$I'filename'}
INCLUDE FILE
This directive instructs the compiler to include the named file in the compilation. The file is inserted in the compiled text after the directive. If **filename** does not specify a directory, then the current directory is searched.

There are no restrictions on the use of include files. This means that an include file can be specified in the middle of a statement part.

Process-Pascal allows, at most, five input files to be open at any given time. This means that include files can be nested up to five levels deep.

{$P=nn}
LINES per PAGE in LST FILE
This directive determines how many lines per page there shall be in the .LST file. The default value is 60 lines per page. **nn** is an integer value.

{$MIB property}
MIB PROPERTY
This directive is used to set one or more default properties to be used in VIGO. The Compiler can automatically generate a SMB file that holds a description of all constants and variables declared within the Process-Pascal program. Each constant and variable has its own set of properties to describe visibility, backup requirement, simulation, permission for read access, permission for write access or protected write access.

The following properties are available:

MIB_Invisible	MIB_Visible

MIB_NoBackup	MIB_Backup
MIB_Simulation	MIB_NoSimulation
MIB_NoReadAccess	MIB_ReadAccess
MIB_NoWriteAccess	MIB_WriteAccess
MIB_NoProtectedWriteAccess	MIB_ProtectedWriteAccess

The *property* following the $MIB directive can be one or more of the above properties in any combination.

Example:

```
  {$MIB MIB_Visible, MIB_Backup }
```

This will set the default values for MIB properties for the succeeding declarations to be Visible and with the Backup property set.

{$IFDEF x}…{$ELSE}…{$ENDIF}
Two basic conditional compilation constructs closely resemble Process-Pascal's if statement. The first construct

```
{$IFDEF x}
  ...
{$ENDIF}
```

causes the source text between {$IFDEF x} and {$ENDIF} to be compiled only if the condition specified in {$IFDEF x} is True. If the condition is False, the source text between the two directives is ignored.
The condition is true if x is inserted as a compiler directive in the option window in Process-Pascal.

The second conditional compilation construct:

```
{$IFDEF x}
  ...
{$ELSE}
  ...
{$ENDIF}
```

causes either the source text between {$IFDEF x} and {$ELSE} or the source text between {$ELSE} and {$ENDIF} to be compiled, depending on the condition specified by the {$IFDEF x}.
The condition is true if x is inserted as a compiler directive in the option window in Process-Pascal.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Large}
   Version:=Large;
```

```
{$ENDIF}

{$IFDEF Large }
  Version:=Large;
{$ELSE}
   Version:=Small;
{$ENDIF}
```

{$RAMSTART=nn}
The RAMSTART directive provides a possibility to tell the compiler the address of the RAM. Default value is $FFE000.

{$SW=nn}
Use this directive to make the next used SoftWire number nn. Following variables are placed according to nn. It is possible to set nn to a value less than the current SoftWire number. The next variable is placed on the first unused SoftWire number.

{$DEFINE_SECTION <Name>, <StartBlock>, <SectionSize>, <SectorSize>, <MinFreeSpace>, <MemoryType>}
Use this directive to declare memory sections.
<Name> : name of the declared section
<StartBlock> the first block in this memory area
<SectionSize> total size of the memory area
<SectorSize> the size of each sector
<MinFreeSpace> the space needed for the operating system to maintain the section
<MemoryType> The type of memory, e.g. FLASH, RAMINITEEPROM, EEPROM etc.
Declaring a section to hold variables in flash:

```
(* Flash section for data, StartAddr: 1, size: 2.5 MByte, sector-
size: 2 kByte, min. free space: 128 kByte, memorytype: FLASH *)
{$DEFINE_SECTION 'Flash', 1, $280000, $800, $20000, FLASH}
```

To place a variable in the declared section

```
Var
  MyFlashVar : integer SECTION: ‘FLASH’;
```

See variable declaration.

{MAXCODESIZE nn}
Use this directive if the compiler should generate an error if the code size becomes larger than nn. This is useful to discover a too large code file at compile time instead of at download.

{MAXDATASIZE nn}
Use this directive if the compiler should generate an error if the data becomes larger than nn in size. This is useful to discover a too large data block at compile time instead of at download.

36 Restrictions in Using Process-Pascal

When programming a controller, the following restrictions must be considered:

When using a controller with a display unit, a cursor must always be defined in the program, to avoid unexpected flicker on the screen.

If a variable of the type BUFFER or TIMER is a component of a complex variable, this component variable may only be used internally in the controller. (P-NET restriction).

When one complex variable is assigned to another complex variable, at least one of the variables must be declared internally in the controller. (P-NET restriction).

It is not permitted to use recursive procedures/functions.

Using UPDATE on a local variable declared in a procedure is not allowable, but can be detected, either by the compiler or by the operating system. If it is detected by the operating system, an error is generated (Update not allowed).

If a local variable, declared in a **global** procedure, is passed to another **global** procedure as a VAR parameter, and the variable is UPDATE'd, then neither the compiler, nor the operating system can detect the failure, and the result of the performed update will be unpredictable.

If an external variable is accessed (via P-NET and a SoftWire number), this SoftWire number must not be declared as an indirect array variable.

If a Net address is declared to be a string, the string must not be declared as an indirect variable pointing to another string.

Interrupt on indirect variables is not allowed.

There is a 32 kB limit for one variable in the following controllers: PD30XX, PD4000 and PD50XX.

A STRING cannot be appended to another string and stored to itself, example:

```
Str:='str2';
Str:='text' + Str;
```

then Str = 'texttext', but it really should be 'textstr2'.

If two BOOLEANs are compared, and one is a part of a variant record, which shares the memory location with e.g. a BYTE, a COMPARE will only result in TRUE if the BYTE values are equal.

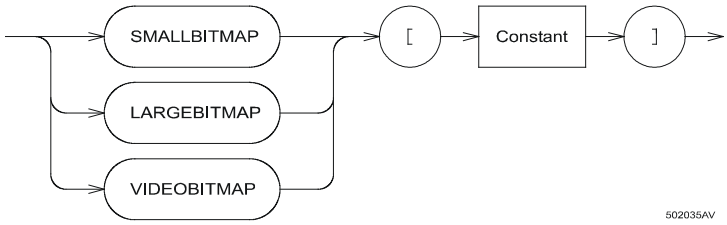
Please note that the error codes in a PD 5000 are not the same as in a PD 4000, and the use of the Bittest function is therefore not compatible between the two controllers.

When a TIMEDINTERRUPT task is used with an interrupt time set to less than 2 sec. to access data on another network through a gateway controller, this may cause the controller to behave “slowly” when there are transmission errors in the “reply-request” issued by the gateway controller. A cyclic task should be used instead.

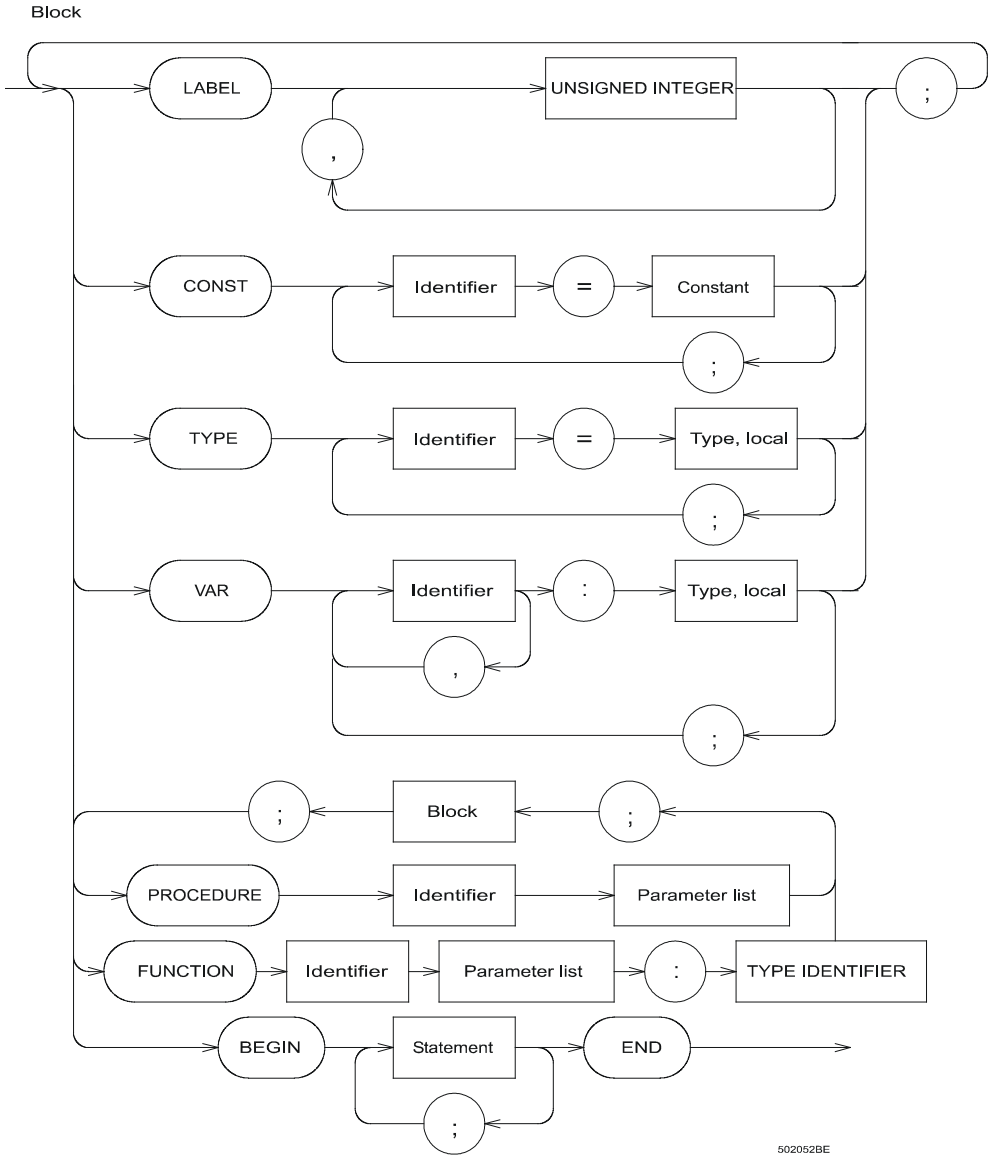
It is not possible to declare Constant strings that include a null character. If a string such as UnderOff = #$1B#$2D#$00 (esc-0) is needed, UnderOff should be declared as a variable and then each character assigned to the string by means of the Char function.

37 Syntax Diagrams

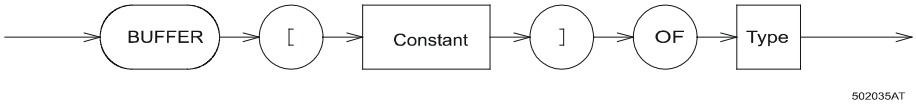
Bitmap type



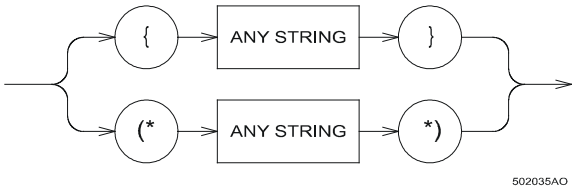
Block



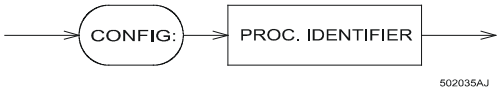
Buffer type



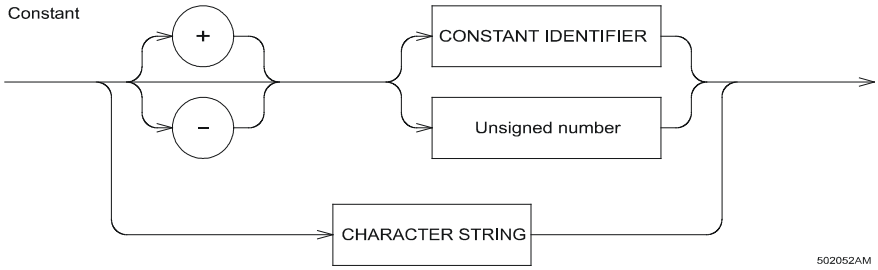
Comments



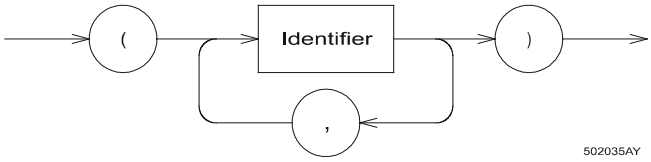
Config



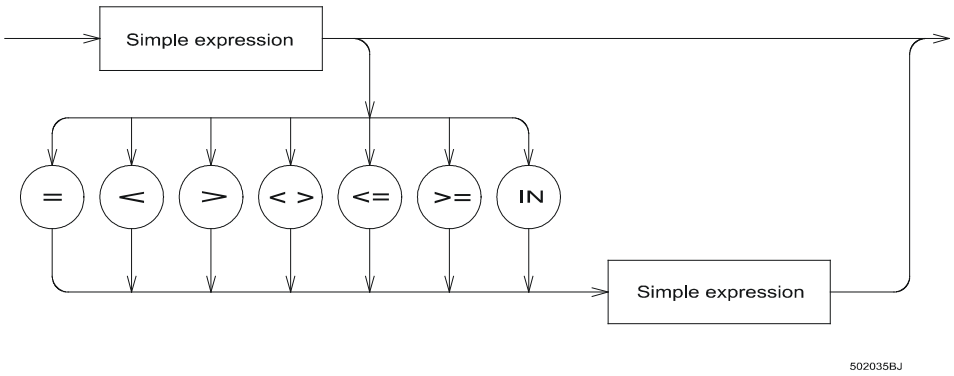
Constant

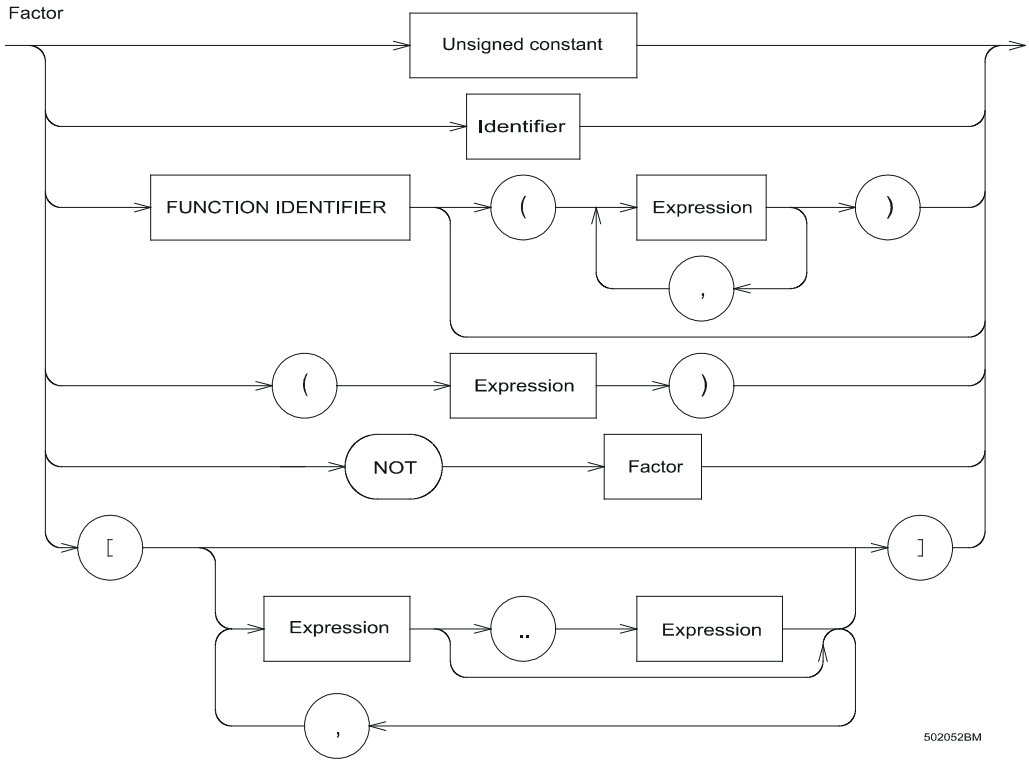


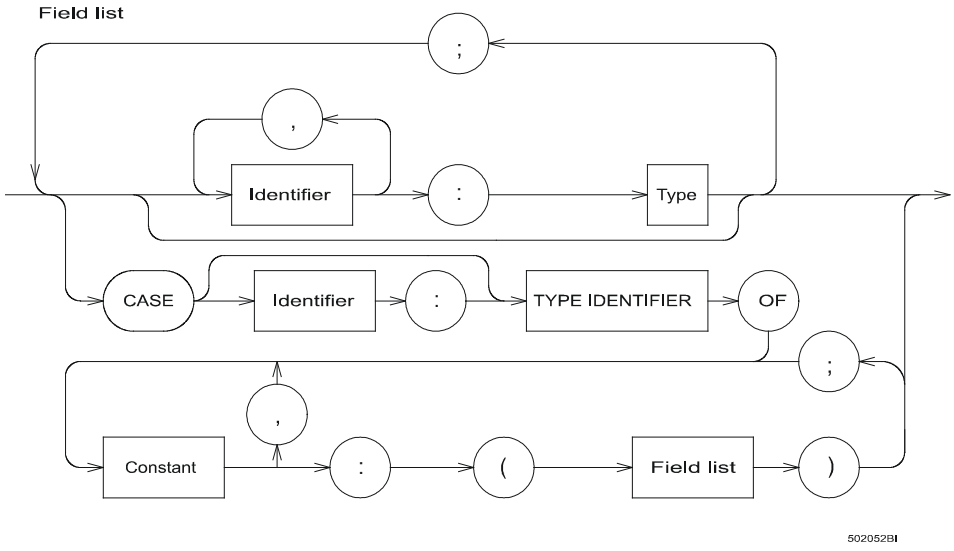
Enumerated type



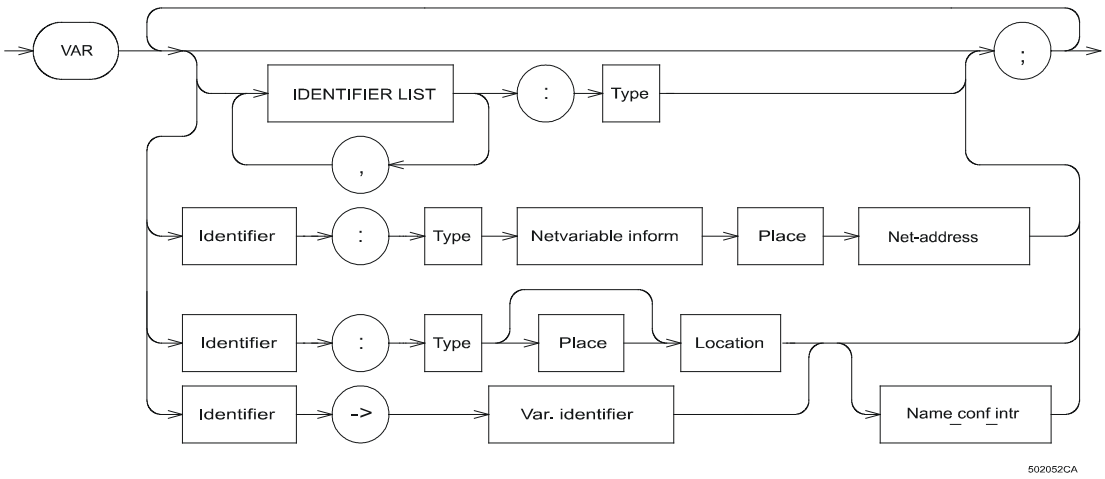
Expression



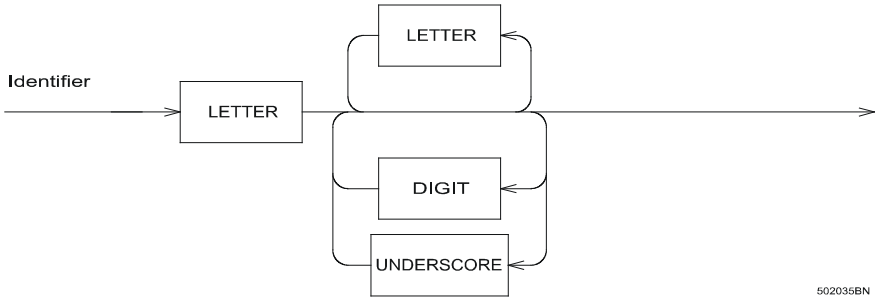

Factor
Unsigned constant
Identifier
FUNCTION IDENTIFIER
(
Expression
)
,
(
Expression
)
NOT
Factor
[
]
Expression
..
Expression
,
502052BM


Field list
;
,
Identifier
:
Type
CASE
Identifier
:
TYPE IDENTIFIER
OF
;
,
Constant
:
(
Field list
)
502052BI

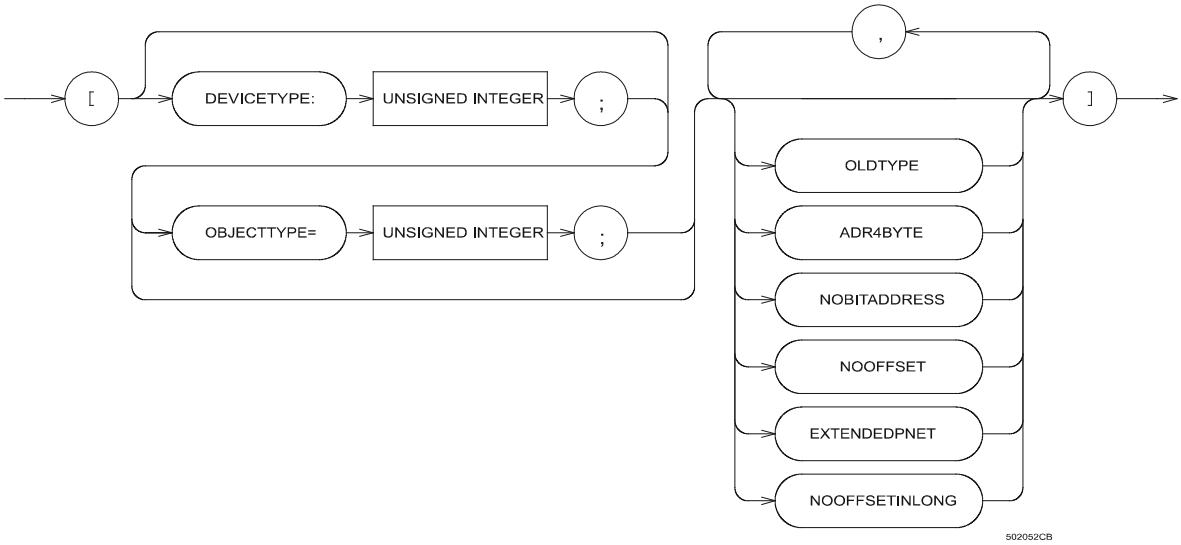
Global variable



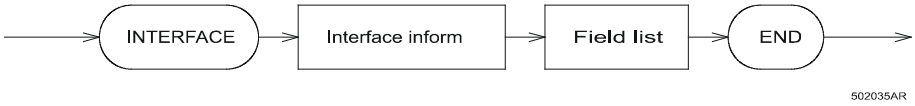
Identifier



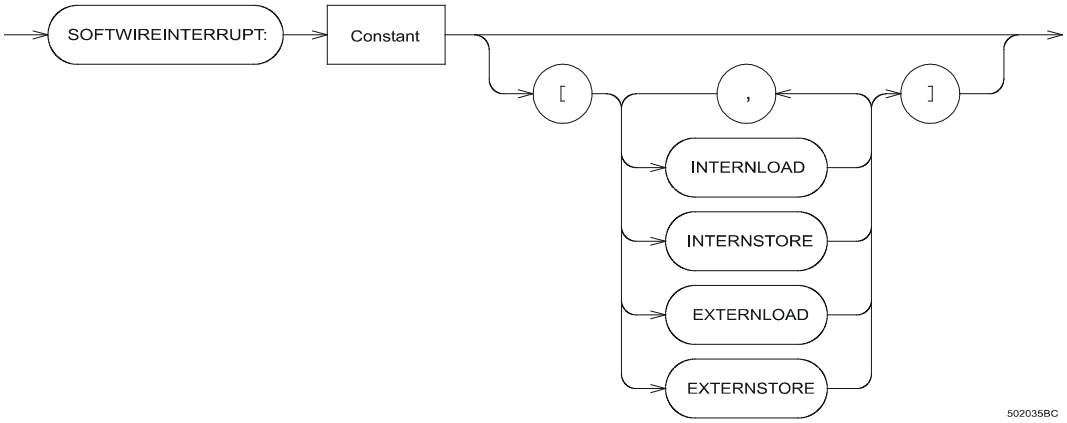
Interface inform



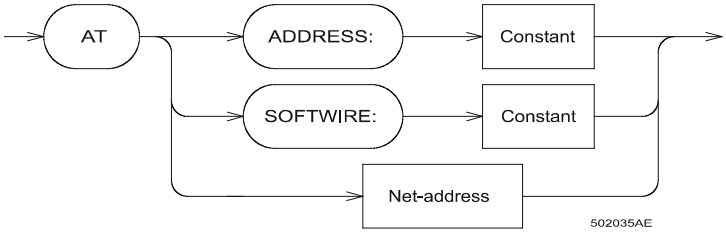
Interface type



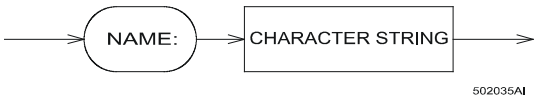
Intr. no



Location

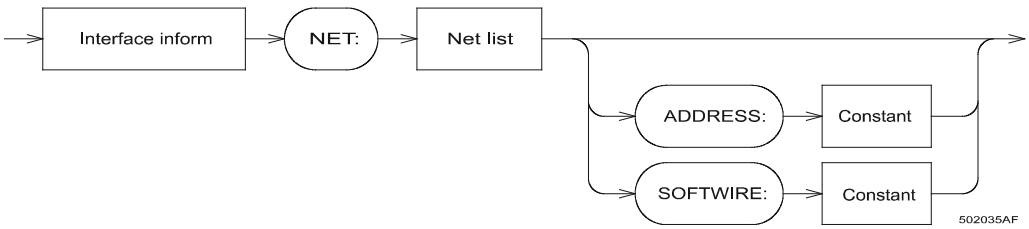


Name

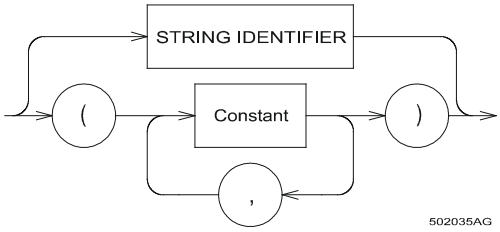


Name_conf_intr

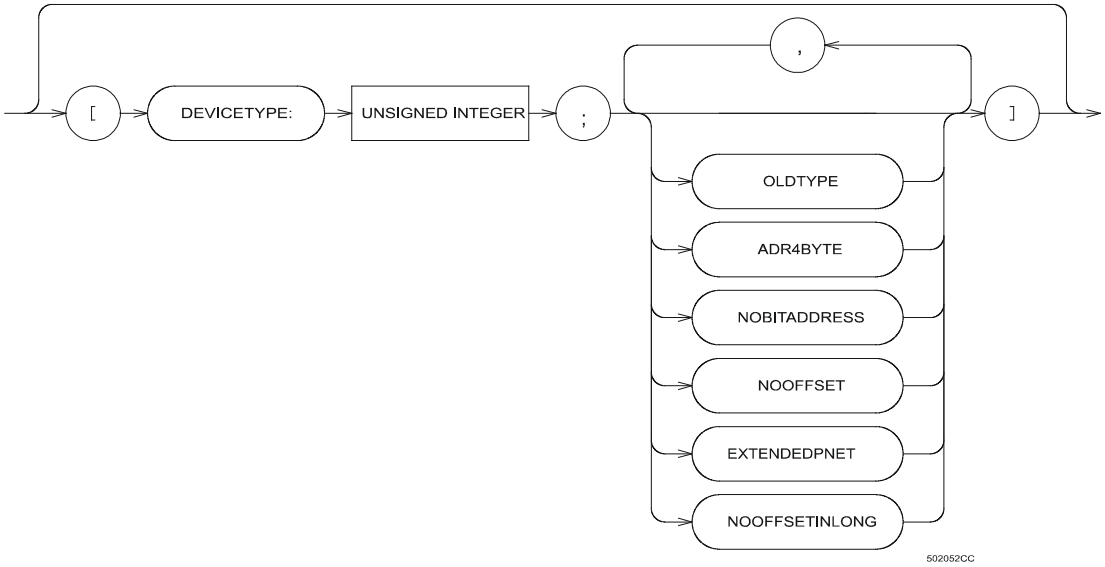
Net-address



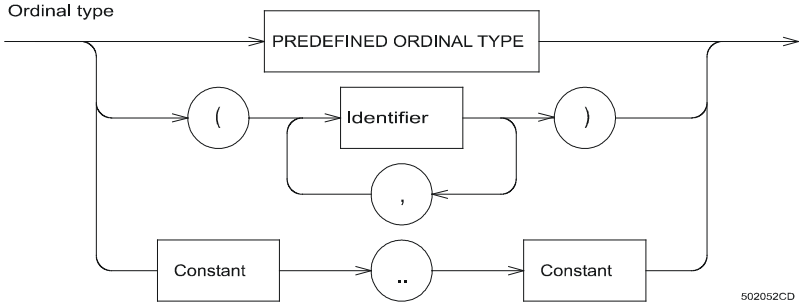
Net list



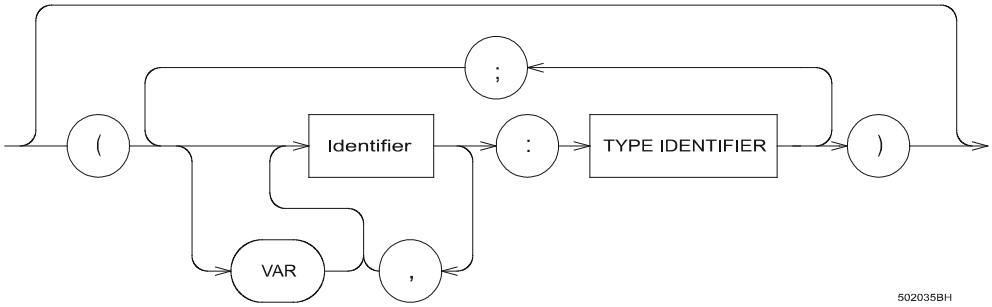
Netvariable inform



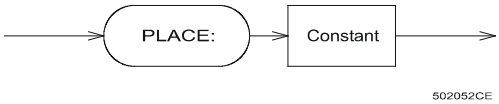
Ordinal type



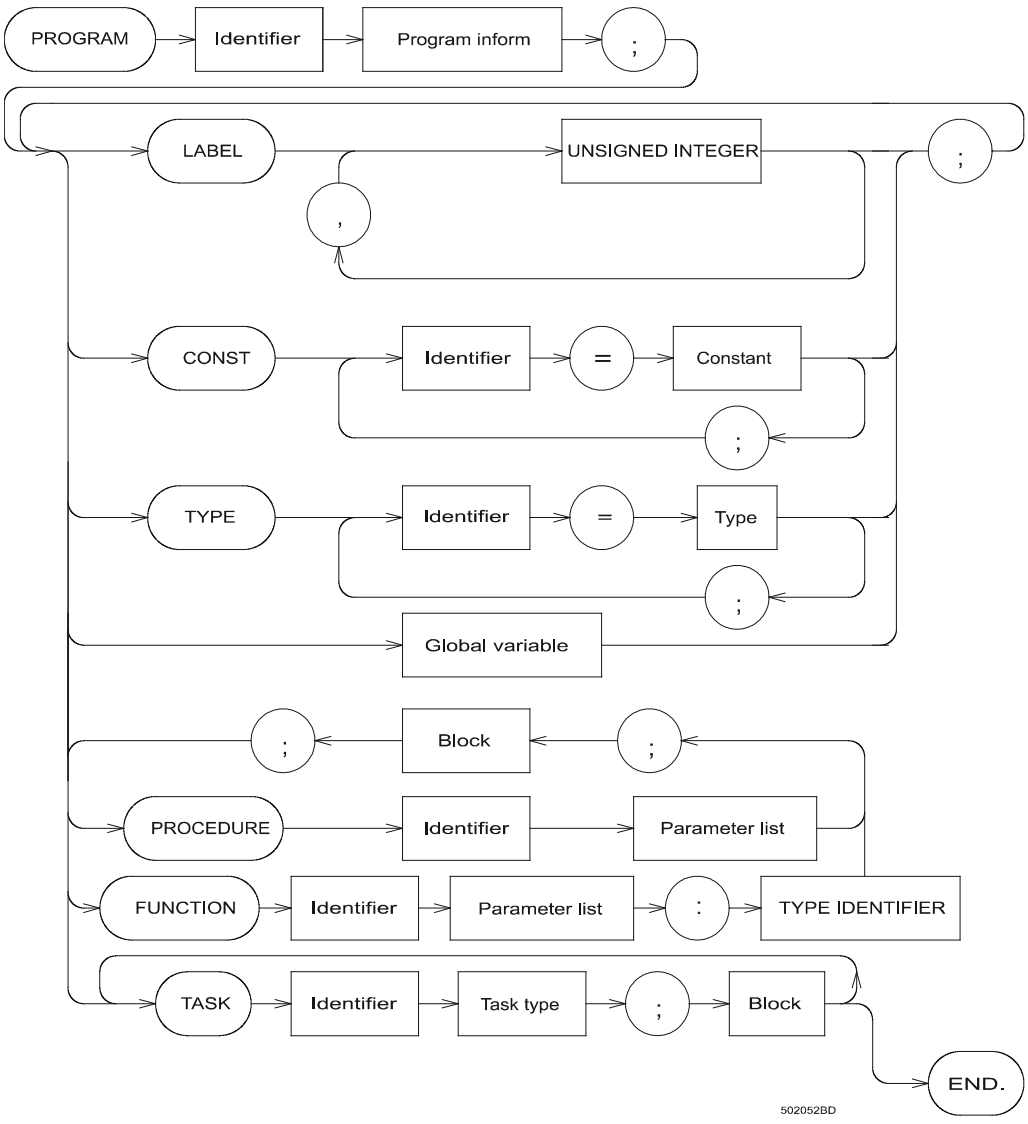
Parameter list



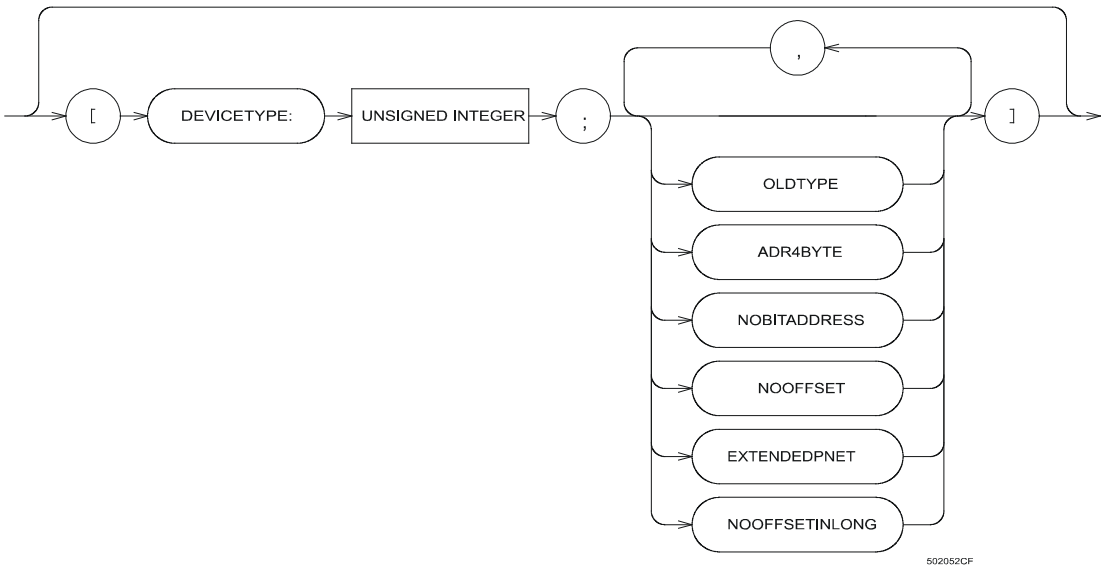
Place



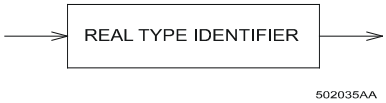
Program



Program inform

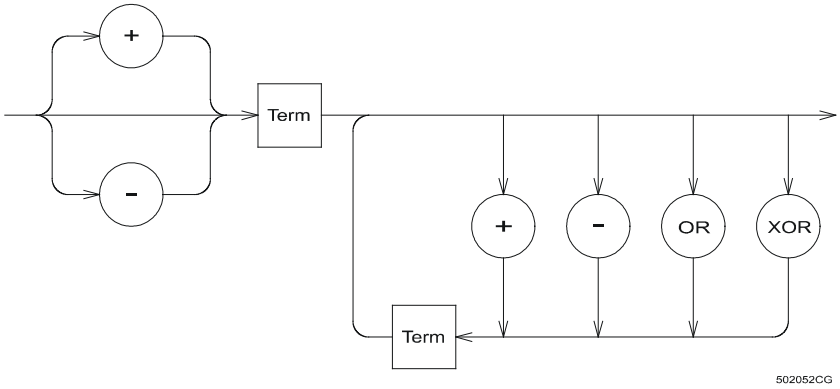


Real type

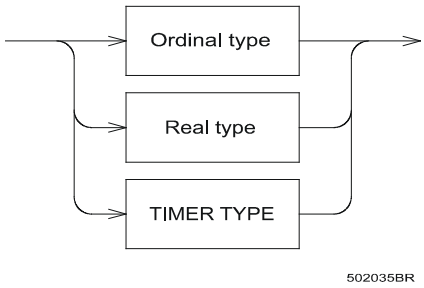


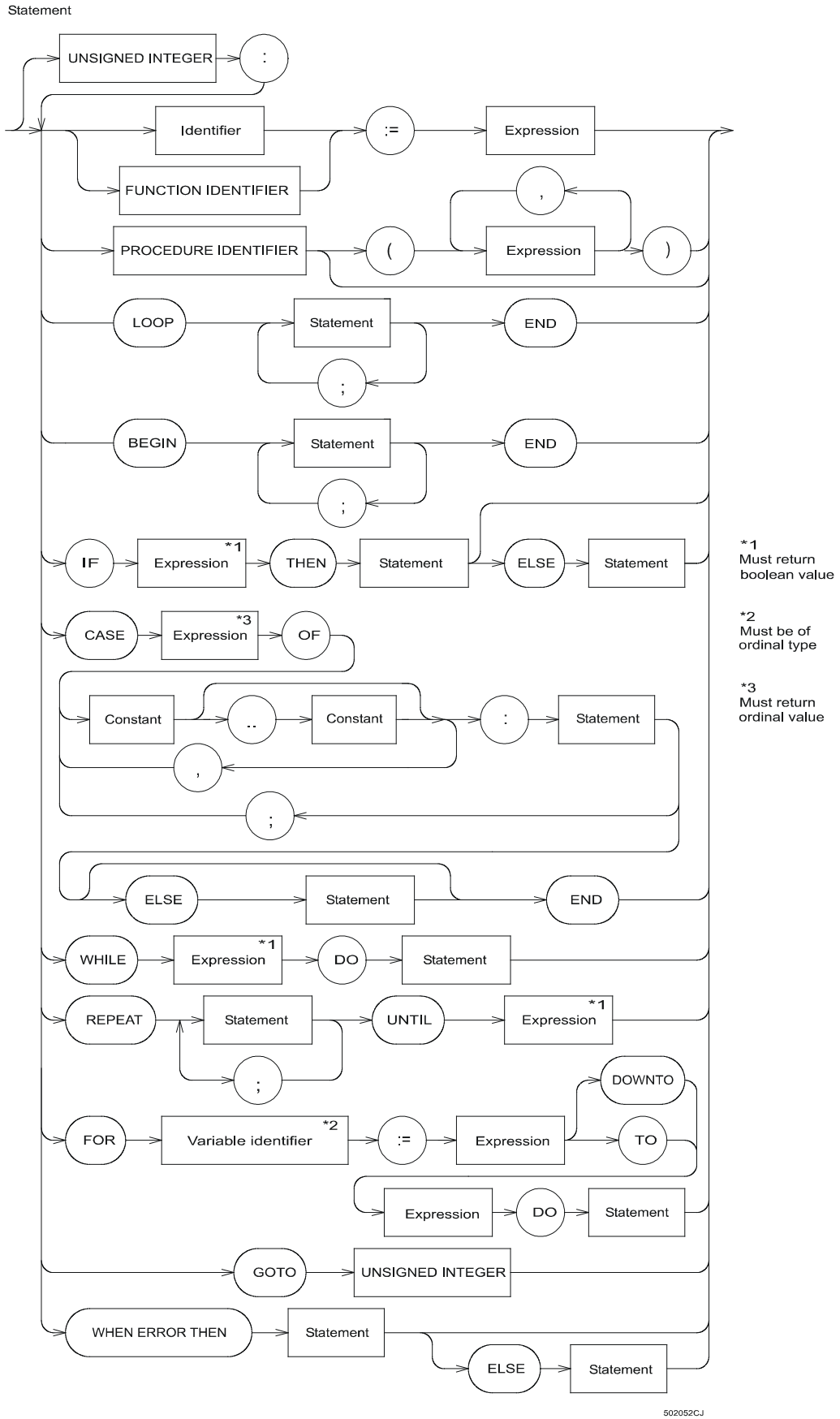
Record

Simple expression

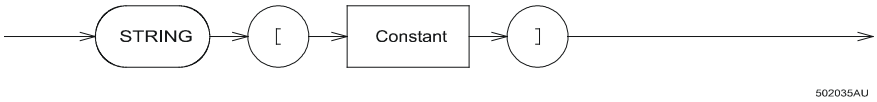


Simple type

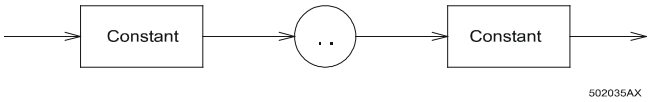



Statement
UNSIGNED INTEGER
:
Identifier
:=
Expression
FUNCTION IDENTIFIER
,
PROCEDURE IDENTIFIER
(
Expression
)
LOOP
Statement
END
;
BEGIN
Statement
END
;
IF
Expression *1
THEN
Statement
ELSE
Statement
CASE
Expression *3
OF
Constant
..
Constant
:
Statement
,
;
ELSE
Statement
END
WHILE
Expression *1
DO
Statement
REPEAT
Statement
UNTIL
Expression *1
;
FOR
Variable identifier *2
:=
Expression
DOWNTO
TO
Expression
DO
Statement
GOTO
UNSIGNED INTEGER
WHEN ERROR THEN
Statement
ELSE
Statement
502052CJ
*1
Must return
boolean value
*2
Must be of
ordinal type
*3
Must return
ordinal value

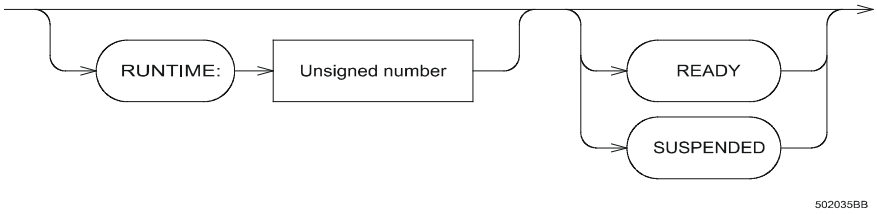
String type



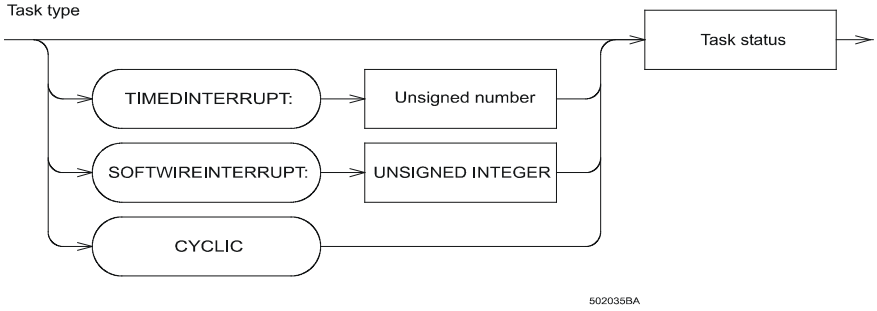
Subrange type



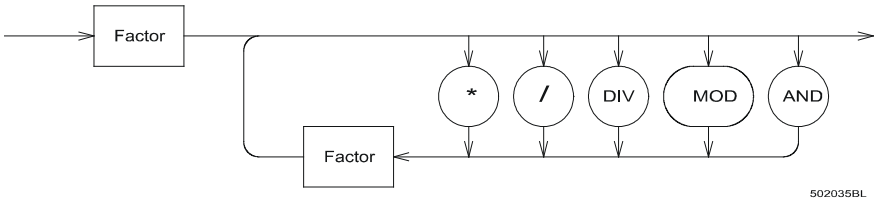
Task status

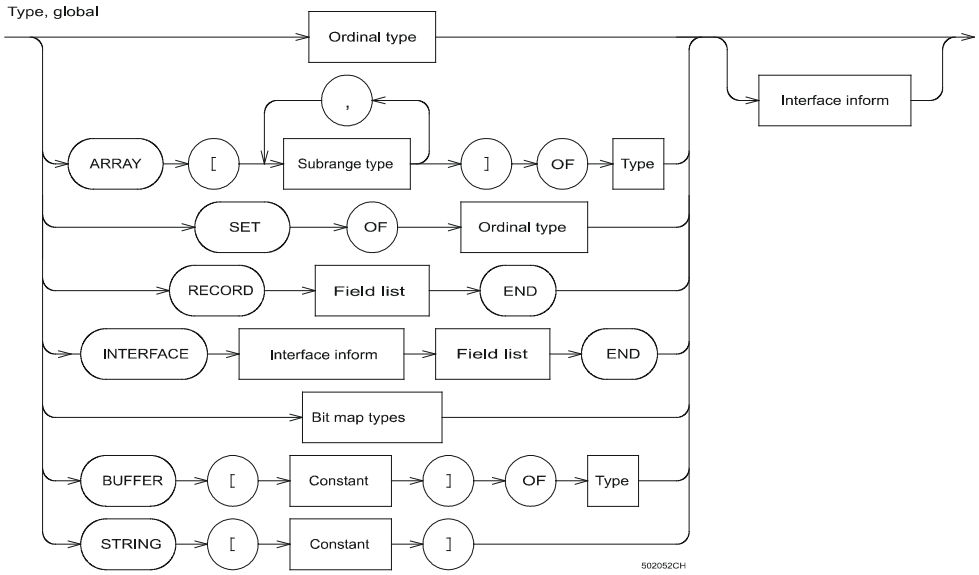


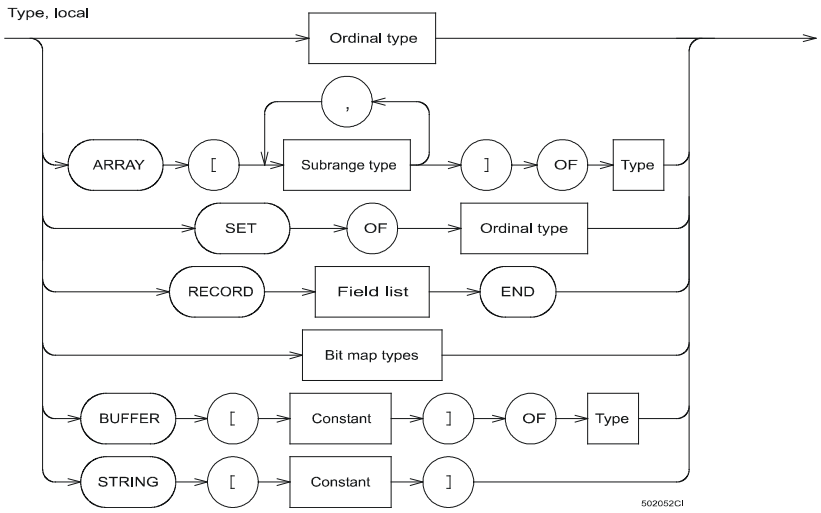
Task type

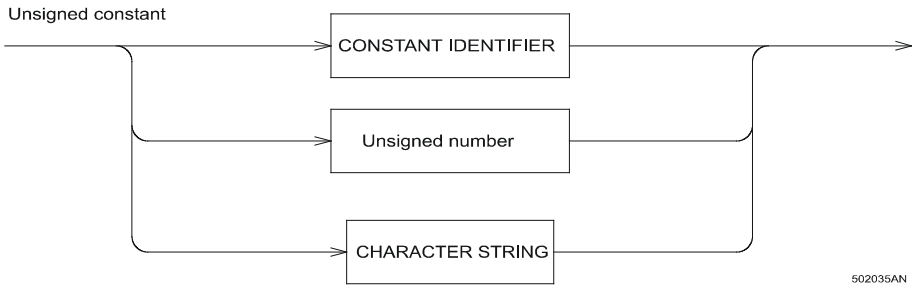


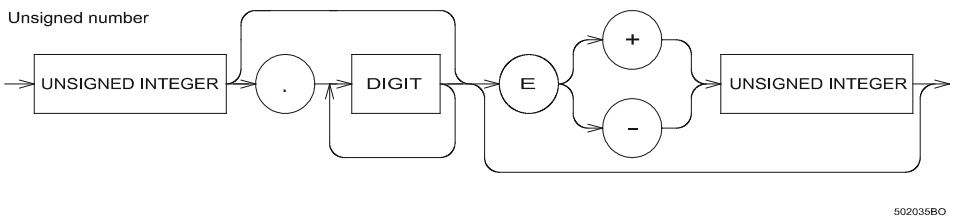
Term



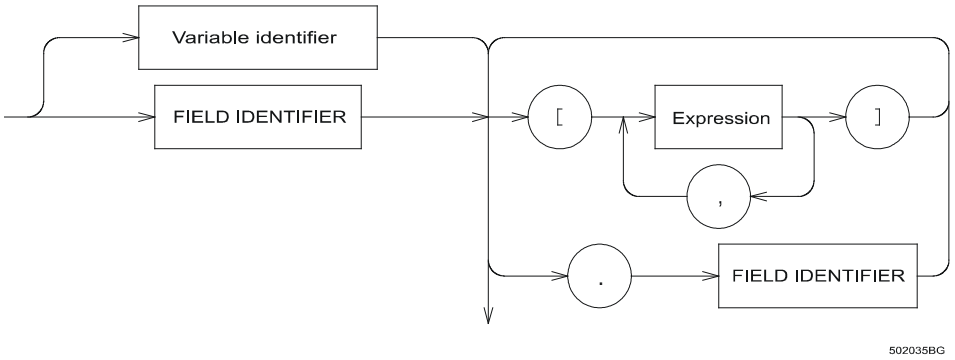

Type, global
Ordinal type
Interface inform
,
ARRAY
[
Subrange type
]
OF
Type
SET
OF
Ordinal type
RECORD
Field list
END
INTERFACE
Interface inform
Field list
END
Bit map types
BUFFER
[
Constant
]
OF
Type
STRING
[
Constant
]
502052CH


Type, local
Ordinal type
,
ARRAY
[
Subrange type
]
OF
Type
SET
OF
Ordinal type
RECORD
Field list
END
Bit map types
BUFFER
[
Constant
]
OF
Type
STRING
[
Constant
]
502052CI


Unsigned constant
CONSTANT IDENTIFIER
Unsigned number
CHARACTER STRING
502035AN


Unsigned number
UNSIGNED INTEGER
.
DIGIT
E
+
-
UNSIGNED INTEGER
502035BO

Variable



Var. identifier