502 086 07

VIGO

Fieldbus Management System
for Windows
95/98/ME/XP/NT/2000/XP/VISTA

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

502 086 07

September 2006

Contents

1 General information

VIGO is a Fieldbus Management System, installed on PC's running the Microsoft Windows^{TM} operating systems. VIGO is used in conjunction with process automation systems, where individual control units are distributed within a plant, and where one or more Fieldbuses are used for the data inter-communication. Microsoft Windows^{TM} is an operating system, which executes programmes, controls the keyboard and screen, manages the hard disc and contains tools for configuration and program execution. In a similar way, VIGO is an "operating system", used to handle the different tasks specific to a Fieldbus system.

Some of these tasks are:

- To provide a uniform and well-defined communication link between standard programs in PC's, and variables and constants in modules (nodes) on a Fieldbus. These variables and constants are identified by a unique name (identifier). A standard program could for example be an Excel spreadsheet, or it may be created using Visual Basic, Delphi, Visual C++ etc.

- To hold information about the location and type of each identifier. This information includes the node address for the interface module, a logical or symbolic address, an offset, the data structure, the data type etc.

- To execute simultaneous communication through different Fieldbus interfaces, and handle the queuing problems that occur in a Windows multi-tasking environment, when several applications wish to communicate at the same time.

- To keep track of which tools can be used with the various types of data and data structures, with consideration for the actual physical objects and interface modules used within the plant. These tools may be configuration tools, compilers, assemblers etc.

- To provide information to compilers and assemblers about variables that already exist in VIGO, so that they do not need to be declared again. It is therefore possible to create compilers where one does not need to declare global variables, because the compiler itself can load the necessary information directly from the description that VIGO holds about a plant.

- To provide an editor, in order to construct and maintain a description of the physical plant, where nodes, data types and the associated identifiers are defined. If one wants to insert, modify or delete single elements from the description, using a program other than the editor, this may be done using the editor's OLE automation interface. This might occur for example, if a plant description already exists in a file, and this is required to be used as a VIGO description of a plant.

- To associate the users program files, help files, connection diagrams, data specifications etc., with the physical objects and modules, which are contained within the VIGO description of the plant.

- To simulate plant data within the PC. This facility can be used in connection with an off-line configuration, backup / restore of plant information and when simulating plant functionality. This is useful for training purposes.

All exchange of data between inter-communicating PC application programmes and VIGO is done by means of OLE automation, (a Microsoft standard for data exchange). As an OLE automation Server, VIGO provides an open and well-defined interface to the user's application program. Any data requested from any point within the plant network, is treated and looks as if it were directly accessed from within the PC. The user does not need to consider variations in different communication protocols, data conversion or addressing methods.

From the users point of view, all these tasks are handled by VIGO, and the result is a simple, uniform and well-defined interface to all data on the networks. VIGO is an open system, where the program interface is written in such a way, that new tools and new Fieldbus systems can be developed and added by the user.

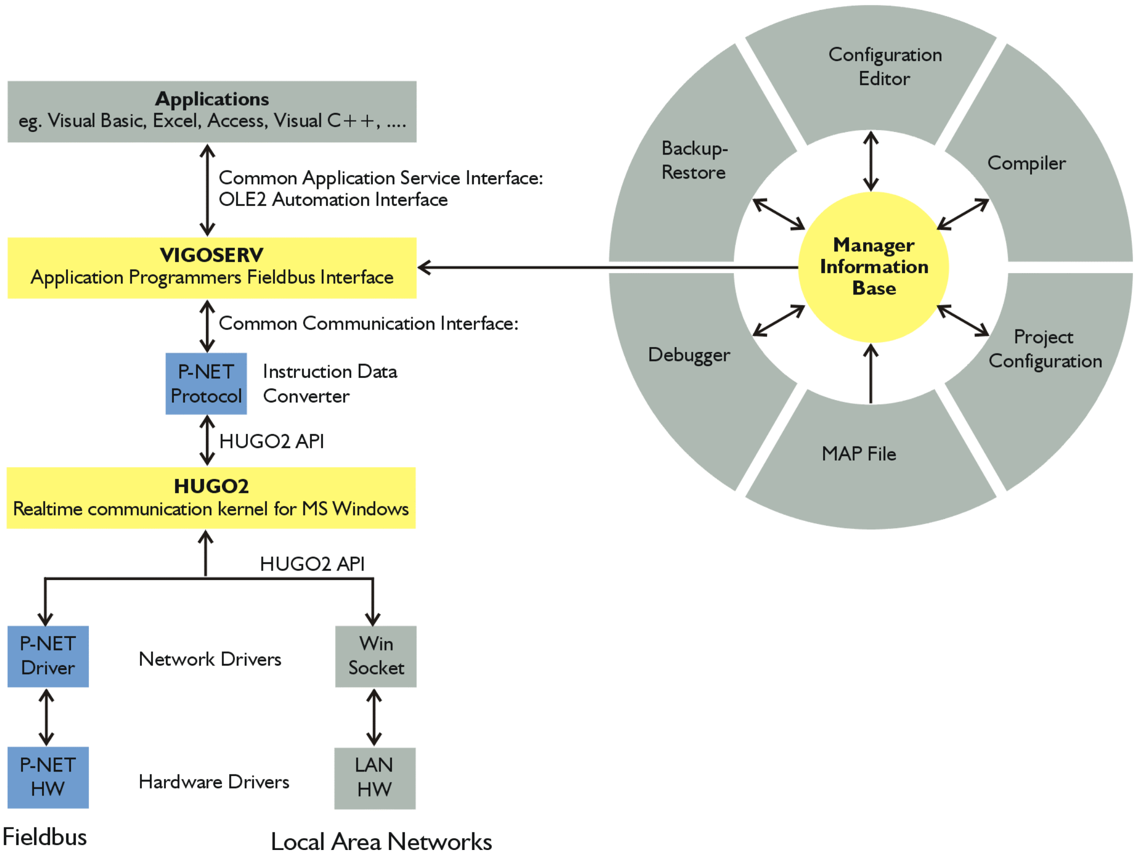
The impact of using VIGO is significant, in that there is now only a need to interface to one system, no matter what the Fieldbus type is. Tools, utilities and programmes developed for use with VIGO can therefore be regarded as general purpose. This means that an increasing number of companies can provide packages for common use, which will result in a shorter development phase. This will also lead to cost savings, since an integrator needs only to understand one system.

1.1 The VIGO elements

The Fieldbus Management System VIGO is a collection of several program elements. The basic elements within VIGO are *VIGOSERV,* the *MIB* and *HUGO2*. The flexible structure of VIGO allows additional elements to be easily added, and to grow with the users needs. These elements, which can be dynamically linked without requiring changes to the existing system, are *Instruction Data Converters*, *Network Drivers* and *Hardware Drivers*.

VIGO allows user applications to be designed without consideration for the underlying networks, by representing those networks as a collection of independent, installable components.
VIGO provides the opportunity for the user to dynamically add new tools, such as a Node Configuration Editor, a MAP file converter, a Backup/Restore utility, a Monitor, a Compiler, etc.

The elements of VIGO are shown below.



All this makes VIGO an open system, which can always be expanded for inclusion of new network connections to physical objects, and new tools for configuration. It is open, in the sense that anyone can provide a network or tool implementation, and anyone can develop an application that uses the communication functions offered by VIGO.

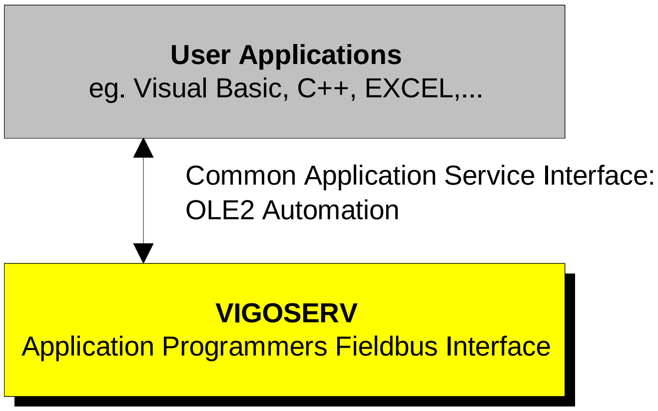
Within the following sections, the elements of VIGO will be examined in greater detail.

1.2 Application Programmers Fieldbus Interface

The Application Programmers Fieldbus Interface - *VIGOSERV*, provides a simple interface to standard program packages such as Visual Basic, Delphi and Visual C++, spreadsheets, databases, Human-Machine Interfaces and other visualisation programs such as SCADA.

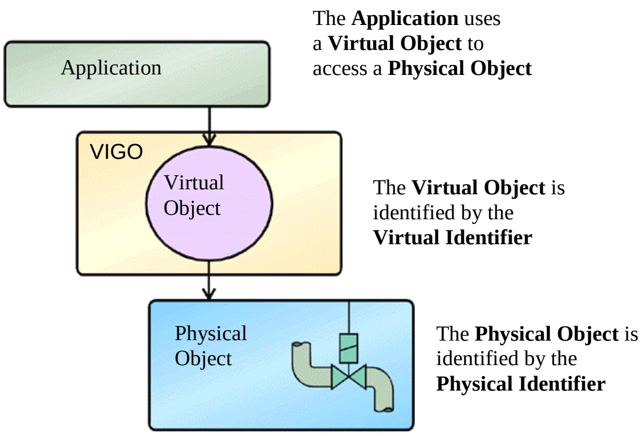
VIGOSERV is an OLE Automation Server, which creates a consistent and transparent interface between the user program (application), and the physical elements (objects) within the plant.

OLE Automation is a part of Object Linking and Embedding (OLE2), which is a facility within Microsoft Windows™, to enable real-time exchange of data between applications.



VIGOSERV supports functions, such as read and write to variables, upload and download of files, start, stop and reset of programs, etc., without being aware of network operations. These functions, together with all their parameters, define the Common Application Service Interface. The figure below illustrates the link between *VIGOSERV* and user applications.

Any manipulation of a particular physical object is achieved via its associated virtual object within *VIGOSERV*. Virtual objects are created by user applications, where a virtual identifier is also defined. The virtual object is made to point to the physical object by means of the physical identifier - a unique name. The physical identifier is defined in the *Manager Information Base*.



1.3 Manager Information Base

VIGO includes a *Manager Information Base - MIB*. VIGO uses the *MIB* to describe the **whole** Fieldbus control system of a plant, which in VIGO is called a *Project*.

In general terms, a Fieldbus system is constructed with a number of Fieldbus devices, called Nodes. The *MIB* contains a description of the different Nodes in the system, and holds information about these Nodes, such as Node Identifier, Nets, Node address, Node type and other relevant information. It also holds information about the Nets within the project. From all this information, the communication path to the Node can be computed.

Furthermore, a node consists of a number of variables. The *MIB* contains a description of all the variables within a Node that may be accessed via the Fieldbus. Each variable within a Node can be of simple (byte, integer, real etc.) or complex type (array, record, string).

In VIGO, the entire collection of variables within a Node can be regarded as one large variable of complex type, the Node type. Access to a variable within a Node is described using the same method as with an access to a Record in the Pascal or C languages, where the Node is the Record and the variable is a field within that Record.

In a similar way, the contents of all Nodes within a plant can be regarded as one huge variable, organised as a Record and represented by a *Project identifier*. Access to a Node within a Project is then described by means of the *Node identifier*, where the Project is the Record, and the Node is a field within the Project Record.

A *global identifier*, unique for a specific variable within the plant, may now be composed by combining the above-mentioned structured elements. A *global identifier* is the same for any device within the Project and starts with the *Project identifier* followed by a ':'. The rest of the *global identifier* is then constructed, by appending the *Node identifier* and the *sub-element identifiers*, to create the complete path to the variable. Each identifier is separated by '.', in exactly the same way as access to fields in a Record, e.g.:

Project_Identifier:Node_Identifier.Variable_Identifier

Thus, a Variable definition consists of a *Variable_Identifier*, information about the location of the Variable, and a Type description. Such a definition must be available for any type of Variable, be that a simple variable, a complex variable, a Node or a Project.

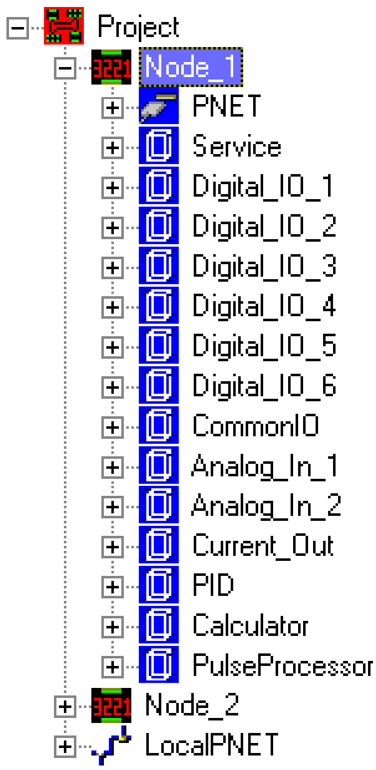
As an example, a simple Variable will be used. The Variable is identified by a Name, called the *Variable_Identifier*. The location of the Variable describes the internal address within a Node. The Type description for a simple Variable just defines one of the basic data types, e.g. real, byte, boolean etc.

Another example of a Variable is a Node. A Node is identified by a Name called the *Node_Identifier*. The location of the Variable describes the Fieldbus 'path' to the Node, including specific Fieldbus information. The Type description for a Node is given by the Node Type, which describes the internal variable structure.

If more than one Variable of identical Type is found within a Project, the Type only needs to be defined once. This includes *Node Types*.

Node Types are typically rather complex, but having a well-defined structure. Such types may be generated automatically from device descriptions or by compilers/assemblers.

The user interface used to monitor the contents of the *MIB* and to enable the structure of the system to be illustrated is handled via a *MIBOCX*. This is an OLE Control Extension (OCX) according to Microsoft Windows. The *MIBOCX* allows a browser function to be performed, and displays a tree-structure, in a similar way to standard file managers. In this case however, the elements are not drives, directories and files, but Project, Nodes, Variables, and Types. An example is shown in the figure to the right. This OCX control can be directly called and used by an object oriented programming language supporting this feature (Visual Basic, Visual C++, and Delphi). This OCX control is used within a number of different VIGO tools, including the *MIB Edit*.



In a similar way to standard Windows programmes, the right mouse button can be used within the *MIBOCX*, to show dedicated menus, depending on the selected object. This means that selecting a Node and using the right mouse button results in a menu list relevant for a Node. Selecting a Project provides another menu list relevant for a Project. This is described in more details later.

As described above, the *MIB* contains all the information required to access a physical object, such as a digital I/O, an analog I/O, a flow meter, etc. When *VIGOSERV* requests information from the *MIB*, using a *global identifier* for a physical object, the *MIB* collates all necessary information about the physical object, and returns this to *VIGOSERV*.

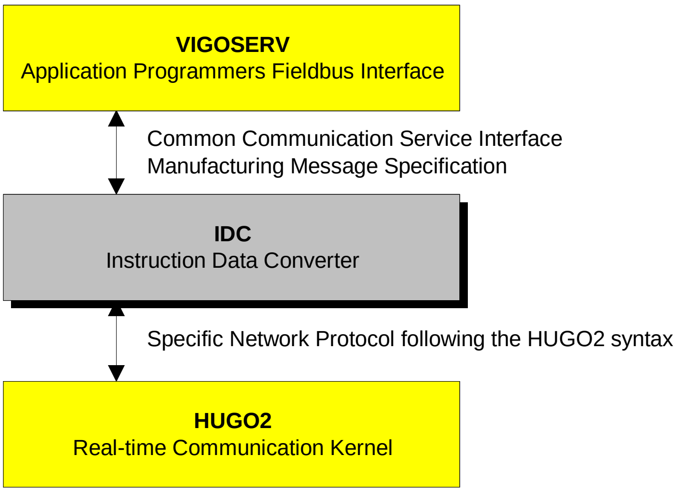
In other words, the *MIB* describes how data is structured, how different data elements are related, where data is stored, and who has access to that data. It therefore enables a physical plant to be completely described as a Project, in terms of data, related data structures and data location.
Once the data definition is completed, a system is capable of automatically acquiring data from, and distributing data to, control level devices, such as Windows applications, process computers, PCs, PLC's, I/O modules, etc.

1.4 Instruction Data Converter, IDC

Different Fieldbus systems may use dissimilar data formats, syntax's and services on a variety of networks. The purpose of VIGO is therefore to have a common application program interface to any Fieldbus interface. *VIGOSERV* defines a Common Communication Service Interface, which fulfils the demands for services and data formats for the different Fieldbus types. A plant can be built, which uses a variety of different Fieldbus systems simultaneously. For each Fieldbus within the plant, it is therefore necessary to be able to convert to/from various sets of services and data formats into the common format. This conversion is performed by a set of Instruction Data Converters, *IDC*'s, one for each Fieldbus system.

Information exchange between *VIGOSERV* and the *IDC*, is based on the RACKS - specification.
The Instruction Data Converter (IDC) is a Microsoft Windows™ Dynamic Link Library (DLL).

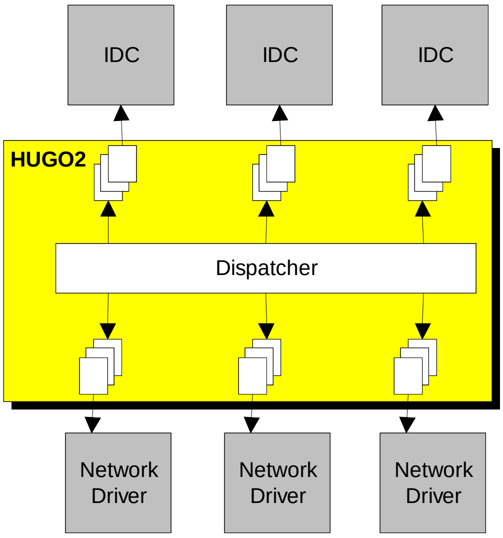
The *IDC* must convert the VIGO data and services into the related Fieldbus data and services that are understood by the relevant Fieldbus node.



This data must be packed in such a format in order that the related network driver is able to transform it for network communication. The *IDC* and network driver is always closely linked to each other, by means of an internal network specific packet format.

1.5 HUGO2, the Real-time Communication Kernel

The routing and handling of several simultaneous information packages for the same, or different networks, is also managed by VIGO, via the real-time communication kernel *HUGO2*. *HUGO2* ensures that communication packages and messages do not get mixed, in situations where several applications are trying to access the same bus system, in a multi-tasking environment. *HUGO2* takes care of defining and managing networks, queuing and routing messages, establishing error-handling procedures and handling interrupts at different levels. The queuing facility in *HUGO2* is shown in the figure to the right.



HUGO2 is designed for both time critical and non-time critical communication. Time critical communication is controlled by hardware interrupts, whilst non-time critical communication is performed by means of messages within the Windows environment.

HUGO2 is able to handle several communicating applications simultaneously, which may involve dealing with many requests and responses at the same time.

HUGO2 can dynamically load network drivers, which gives the user the opportunity to add new network drivers if required. Basically, *HUGO2* is a transport system, which means it does not need to know what is being sent. The interpretation of Fieldbus messages is carried out by the associated *IDC*.

HUGO2 is also a communication system that manages data security and integrity, for data inquiries made to the plant.

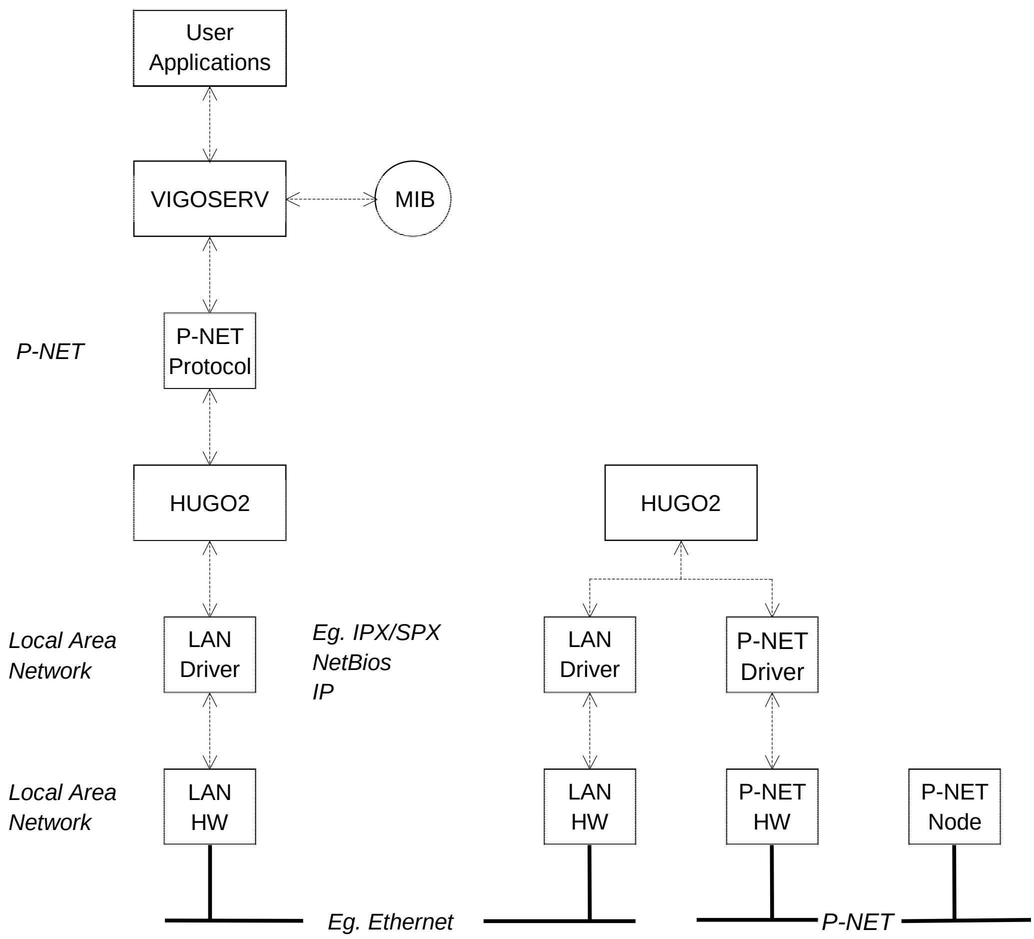
1.6 Network Driver

A *HUGO2* Network Driver interface provides the connection between *HUGO2* and a standard Fieldbus driver (for example P-NET), or a LAN driver (for example *VIGO-IP*).

A communication network can be realised in several different ways. Three network types can be connected to *HUGO2*. These are Fieldbuses, Local Area Networks (LAN) and Wide Area Networks (WAN). There are distinct differences in the usage of these network types. The LAN and the WAN types are only used for transporting messages, which means they have no knowledge of what is being sent on the network, whereas Fieldbuses have built in protocols, which interpret the contents of what is being sent and received.

The combination of network types provides the capability of installing a Windows application on a PC, which has access to a Local Area Network and/or Wide Area Networks, and then routing the information via another PC, which has access to a Fieldbus, to which the physical object is connected.

This is all illustrated in the figure below.



2 The VIGO programs

The VIGO Fieldbus Management System is a collection of associated programs, DLL's and tools. The principle window of the VIGO program is shown below. This must always be loaded (or minimised), when VIGO functionality is required.

The VIGO window consists of three tabs: [MIB View], [Workspace] and [MIB Edit].

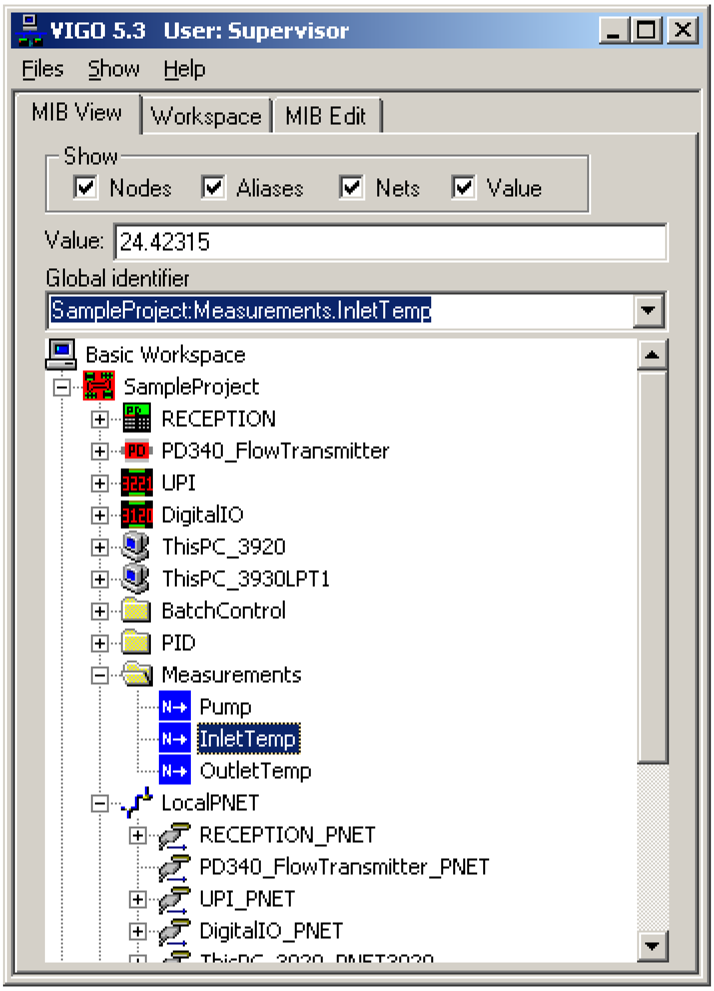
VIGO can be used for a variety of purposes, which depend on the requirements of the user.

Before VIGO can be used, it must be configured to match the required VIGO environment. This includes establishment of a Workspace having a selection of projects enabled, together with all the parameters of the appropriate drivers correctly set.

MIB Edit is used by the systems integrator who wants to set up a new, or modify an existing system, and needs to carry out the necessary configuration of the nodes. In this situation, VIGO can be started up from the Start Menu, from a shortcut or directly from the Windows Explorer.

2.1 MIB View

The [MIB View] tab shows the projects that can be accessed by the application programs that need to use VIGO. VIGO will be automatically started as soon as one application program creates a VIGO object. Under these circumstances, VIGO will be loaded in a minimised state, and will only appear in the task bar. VIGO will be automatically closed again, when VIGO objects are no longer required by the applications.



[MIB View] provides an illustration of the structure of a project, in terms of nodes, channels and other variables that are included in a particular system.

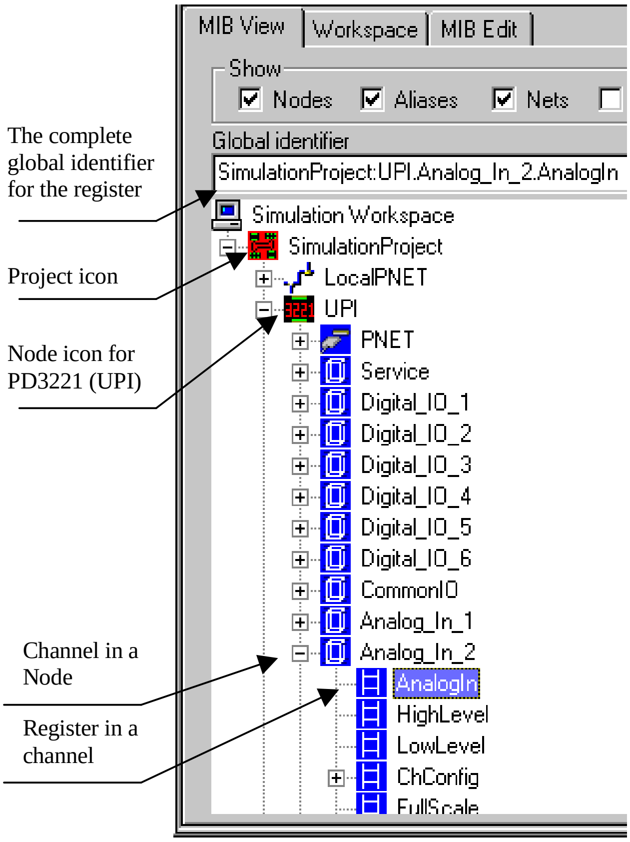
Furthermore, [MIB View] enables the user to find and select variables, in order to call upon other programs and tools that are relevant to the selected item. Such tools can be selected from a menu that appears when the right mouse button is clicked on a highlighted item.

The *MIB* supports a *Grouping* element that allows the user to group nodes, *aliases* etc. A group is shown with a group symbol, which is similar to a folder in Windows. The use of groups helps to provide you with an improved structural overview of your project, since you can create a group for a specific section of your plant or for a single machine.

[MIB View] utilises a custom control, called *MIBOCX*, which is an "OLE Control Extension", designed for VIGO. The *MIBOCX* is used to provide a visual representation of the structure and relationship of the variables within a project.

The project structure is shown in the form of a tree, in a similar way as does the Windows Explorer file manager. However, instead of showing folders and files, the *MIBOCX* in [MIB View], illustrates the nodes and variables relating to the project description of the system. The same *MIBOCX* control can be included in other programs, such as those developed using Visual C++, Visual Basic or Delphi, since all of these languages support the use of such controls.
If the right mouse button is pressed when an element in the structure is highlighted, a menu is shown. This menu provides a choice of functions and tools, but which are only relevant for the selected element.

A *Project* is identified by a name, and is represented by a *Project icon* in the *MIBOCX*.
A factory can be divided into different projects or projects can represent systems at different locations.
The elements that are used to describe a physical plant within a project, consist of *Nodes*, *Aliases* and *Virtual names*. The *Aliases* and *Virtual names* are used as shortcuts for constructing and combining identifiers from already defined variables, thereby giving access to actual variables in a more convenient way. A *Project* can be expanded into it's elements, by clicking on the + sign at the *Project* icon.
Networks are visible in the *MIB* structure. When *Nets* are created, they are shown in the *MIB* with a net symbol. Opening a *Net element* will show a list of nodes that are connected to the selected net.



A *Node* within the *Project* description (the *MIB*) is defined as a module or a unit within the physical plant (e.g. a PD3221- Universal Process Interface, UPI).
Each Node is represented by a *Node icon* in the *MIBOCX*.

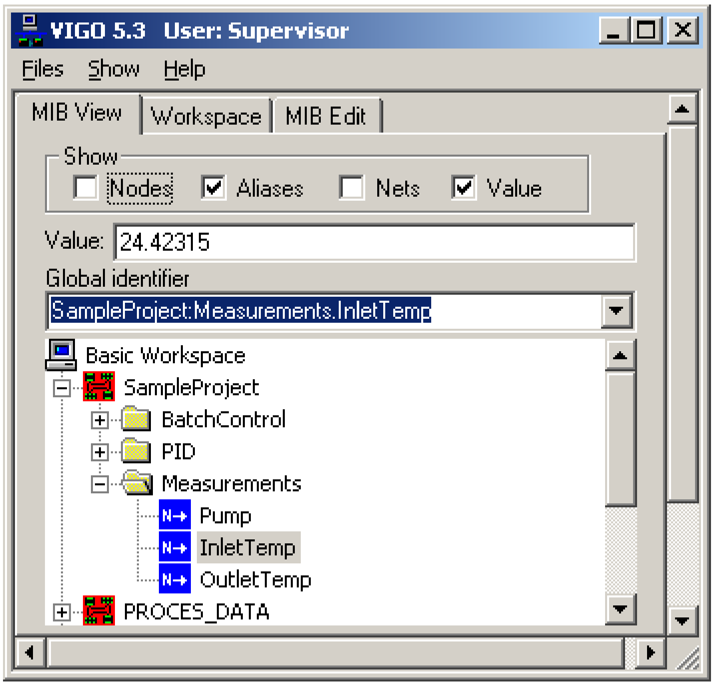
A *Node*, which is a variable, is based on a type, a *Node type*. The *Node type* describes the data structure within the node. The data structure of a particular *Node variable*, e.g. it's *channels* and *registers*, can be seen by clicking on the Node icon's + sign. A particular *Node type* can be used many times within a project description.

The *MIB* description is built using a number of inter-related elements. These elements may be of different *Kinds*. One *Kind* of element can represent a *Node*. Another *Kind* of element can represent an Array, and yet another can represent a *Channel*. Each *Kind* of element is represented with a particular icon, used to illustrate the variable in the *MIBOCX*.

It is also possible to select an element of an array, by changing the element index number. Click once on the selected index number, type in a new index number and then click the mouse pointer elsewhere.

In the example shown above, the Project is called *Simulation Project* and the Node is called *UPI*. Within the *UPI* node, the data structure in the form of channels can be seen. Within the *ANALOG_IN_1* channel, the register *ANALOGIN* is highlighted. The complete global identifier for a selected variable is shown in the *Global identifier* field.

The *ANALOGIN* register shown above has the complete global *Physical identifier:*
SimulationProject:UPI.ANALOG_IN_1.ANALOGIN



When a Project icon is opened in the [MIB View] tab, only *Nodes*, *Aliases* and *Virtual names* are shown. This is the default setting. *Nodes* and *Virtual elements* can be individually excluded from the view.

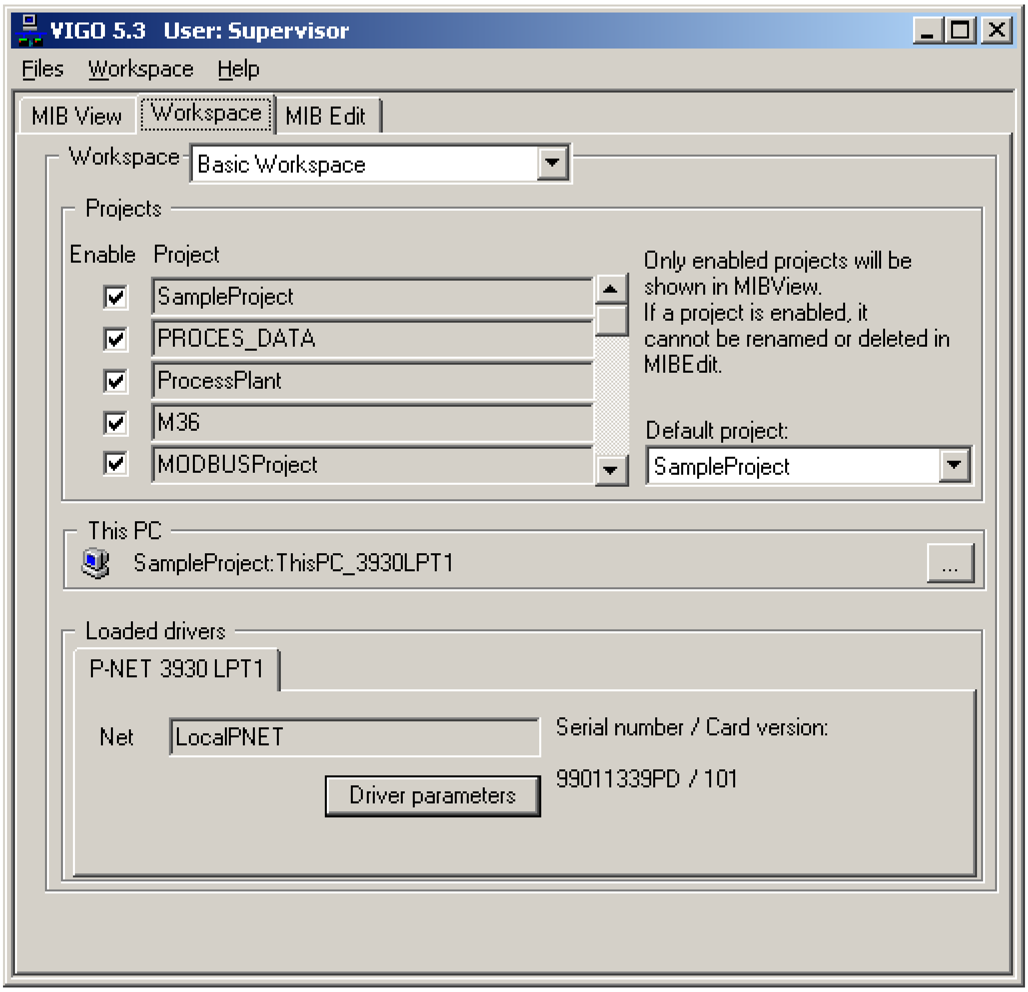
The [Show Nodes] check box and the [Show Aliases] (Virtual Elements) check box are used to limit the number of elements to be shown in [MIB View].*9*

The [Show Value] check box is used to add a *Value* field. This field shows the value of the selected variable. Entering a value, into this field followed by <Enter>, will write to the variable.

2.2 Workspace.

A [Workspace] in VIGO describes which projects will be loaded, defines the default project and identifies a Node in a project where the driver parameters and port settings are defined.

Each *workspace* holds its own set of parameters. The name for the *workspace* can be chosen to be the same as the name of the location. *Workspaces* can be added/deleted from the pull down menu. The [Workspace] list box shows the selected *workspace* used by VIGO.

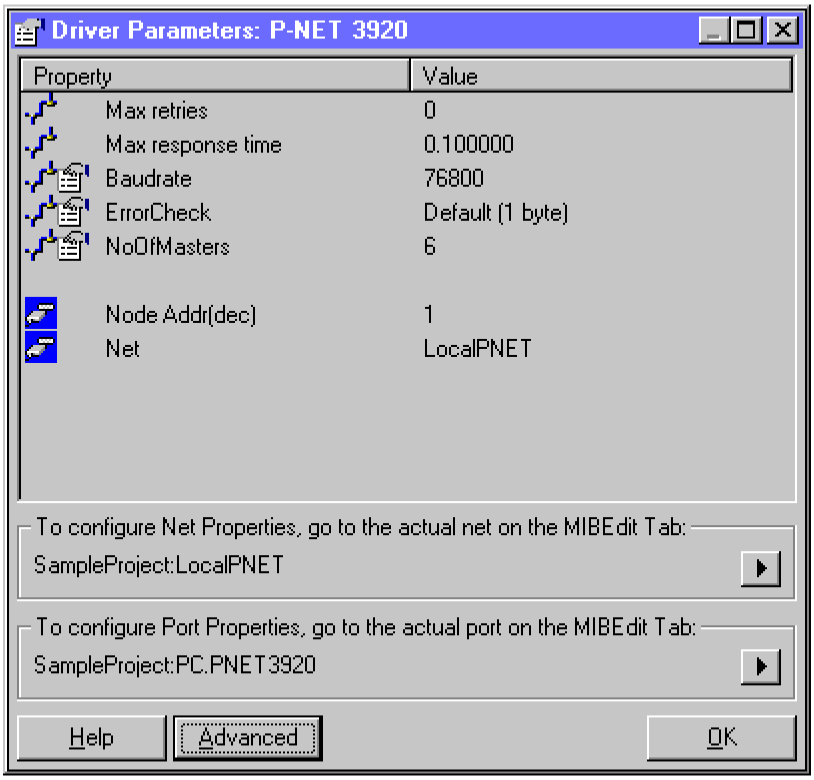


VIGO is designed to enable a PC to simultaneously handle multiple projects. Each *Project* is given a name, called the *Project Identifier*. A *Project* description is stored in a *MIB* file. The [Workspace] shows a list of existing *projects*. Only those *projects*, which are enabled in the actual *workspace*, will be shown in the [MIB View].

A [Default project] can be selected. When the *global identifier* is without project name, the *default project* is assumed. A *Project* may contain a number of networks, each with a number of Fieldbus nodes connected. These networks are specified as properties of the project.

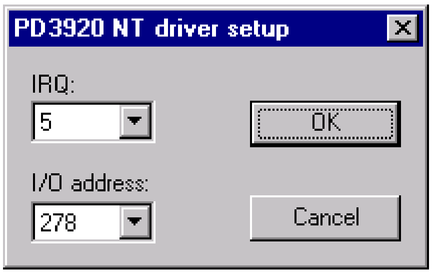
[Loaded Drivers] indicate which drivers are currently loaded.

The <Driver Parameters> button will display a new window, showing Port and Net properties for the driver in question. The properties cannot be edited from this window. See also *Guidance for selecting node address …* on page 65.



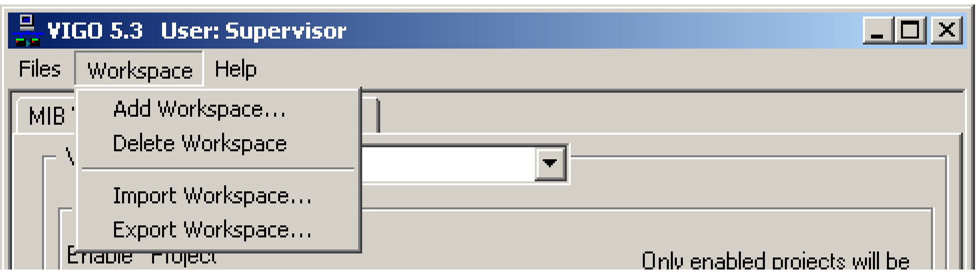
The values of the properties are stored in the *MIB* file, so the values can be edited from the [MIBEdit] tab.

Some drivers, e.g. the P-NET 3920 driver for Windows NT, requires parameters that are not suited to store in the *MIB* file. These parameters can be edited by clicking the <Advanced> button in the [Driver Parameters] window. The <Advanced> button will only be present if the driver in question needs such parameters.



2.2.1 Import/Export

From the [Workspace] menu it is possible to both import and export a *Workspace*.



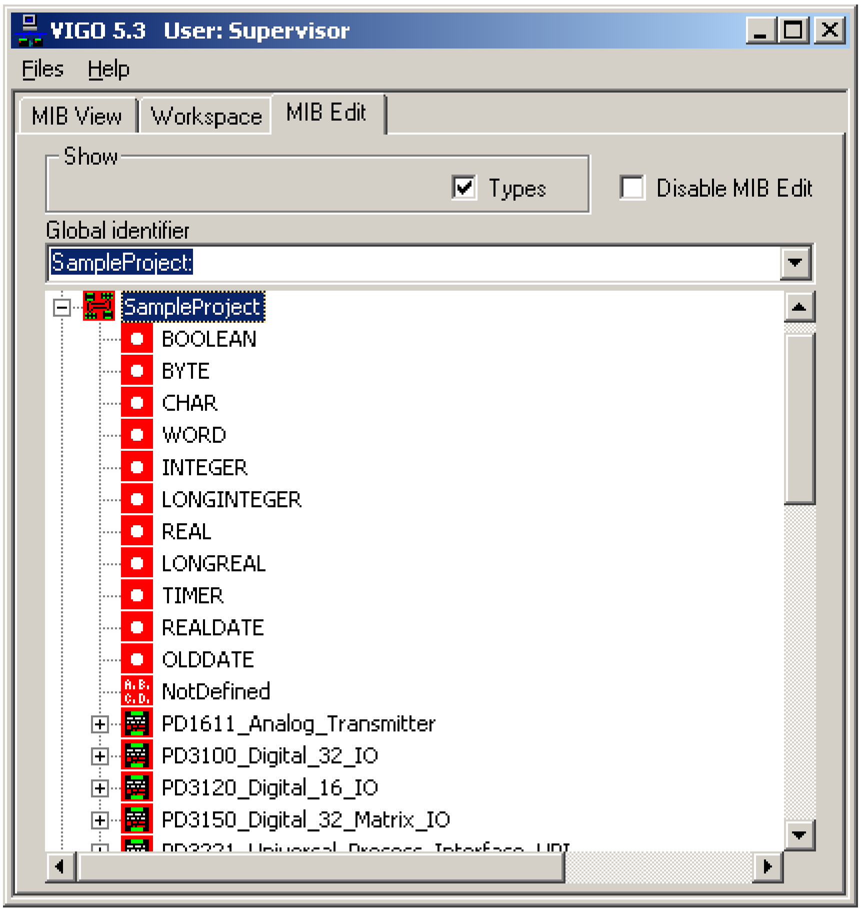
Exporting a *Workspace* will gather all relevant configuration data relating to the workspace (including *MIB* and *SIM* files for enabled projects) and put this data in a single file. This *V̲IGO C̲onfiguration F̲ile* will have the extension *vcf*, and can be imported from another PC.

The Import/Export feature is a fast and simple way to move a configuration from one PC to another. Companies who install VIGO along with their own products can also use it. They can setup their *workspace* as needed, export it and copy the file to a floppy disk. Inserting the disk in the PC during the VIGO install, will force VIGO to import this configuration file instead of the default configuration files that follows with the VIGO system.

2.3 MIB Edit

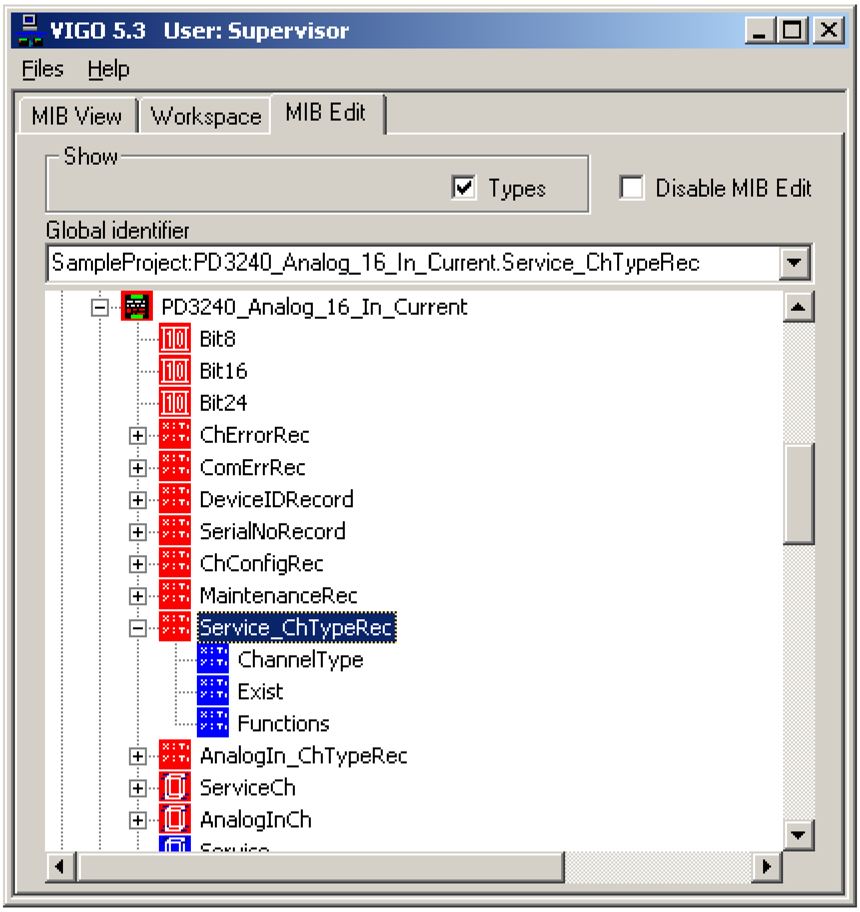
The [MIB Edit] tab shows all the projects that have been set up on the PC, and is used to modify the structure and properties of nodes and variables within a project.
A Variable occupies a memory location in a physical device. Variables are therefore located within Nodes, or as previously described, a Node itself can be regarded as one huge variable. Nodes are declared as being of a particular *Node type*. A *Node type* fully describes all the types of variables contained within that Node. When a *Node type* has been assigned to a Node, the declared variables become available for access through VIGO.

The various *Types* used within the *MIB* can be divided into two distinct groups:



Group 1, which includes Basic types (simple types), and Array types and Strings. These types are represented on the screen by a red icon, and do not include any sub elements (simple types have no "+" sign in front of the icon).
A Basic type is described in the *MIB* as a *Basic Type* Element, represented by a red Basic type Icon. The *Basic types* are: Boolean, Byte, Char, Word, Integer, LongInteger, Real, LongReal, Timer, RealDate and OldDate. The *Array Type Element* also belongs to this group, and is represented by a red *Array type* Icon.

The properties of an *Array type* element holds information about the type structure of the Array elements, as well as the minimum and maximum index for the Array. A string is a special form of an Array (consisting of an array of characters) and is represented by a red *String type* Icon.



Group 2:
This group of types (complex types) is described in the *MIB* as a *Type Element*, and is shown with a red icon, and having one or more *Sub Elements*, indicated with blue Icons. The *Type Element* properties hold the name of the type, and the structure of the type, (*Nodetype*, *Channeltype*, *Recordtype* etc.).

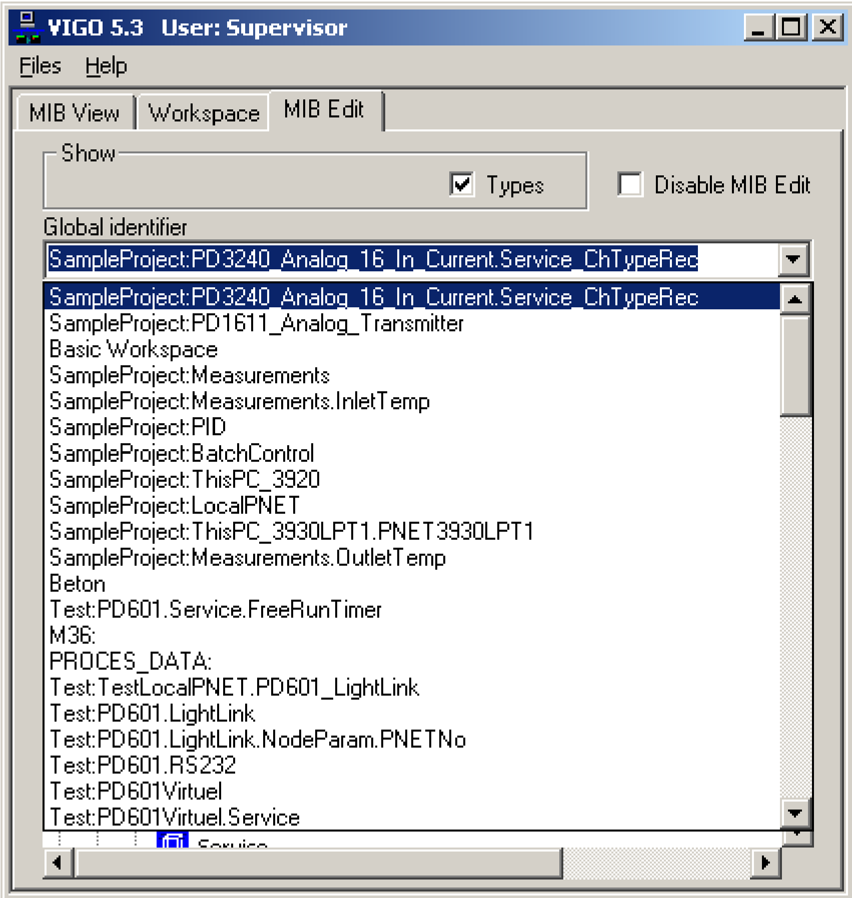
The *Sub Elements* with blue icons represent *Channels*, *Registers*, *Swno's*, Record fields etc. The properties of these Sub Elements describe the relative location of the sub type within the complex type, and the name of the sub type.

Blue icons are also used for physical Nodes, *Aliases* and *Virtual names*. These icons are not shown in Types view.

Note A special kind of Element can be found under a *NodeType*, a '*PortTemplate*'. A *PortTemplate* is not a variable itself, but when the *NodeType* is selected for a Node, a variable based on the *PortTemplate* will be inserted under the Node.

The *MIBOCX* offers the ability to show *Nodes or Virtual* elements or both, in Variables view. The icons shown in the *MIBOCX* represent variables, and they are normally blue, except for the Project icon and the array index, which are red. The purpose of providing a means of selecting which elements to show reduces the total number of icons displayed, and helps to make it easier to select a particular icon.
By clicking a "+" sign, the *MIBOCX* will automatically find and show the icons representing the *Sub elements*. New nodes, and *Virtual elements* of an already defined type, can be easily inserted.
Virtual names and *aliases* can be used to give alternative names to already defined Variables.

In the *Types View* mode, the *MIBOCX* only shows the tree structure of the types to two levels. A red icon with a “+” sign can be opened, and one or more blue icons will appear. This mode provides the opportunity to see all type definitions available within the Project, and to define new types.
When a new type is created, such as a new *Nodetype*, a *Recordtype* or an *Arraytype*, all the associated sub elements must already be declared, before the complex type is defined. This is similar to the procedure used when declaring types in Pascal or C++.



The *Global identifier* is not valid in the Types view.

However, this field can be used as the means to search for types and variables, having known names by keying these in.

Also the drop down listbox will list previously selected elements, and further selections can be made from this list.

The Properties of a new Element must be set up, before the new element can become operational. This is achieved by using the Properties Window, selected by using the right mouse button menu inside the *MIBOCX*.

The [MIB Edit] tab loads a copy of the *MIB* files into memory. These files are **not** used for communication, and during editing VIGO will continue to use the old version of the *MIB* files, until the new files are saved. When the *MIB* files are saved, all VIGO objects will automatically be updated to reflect the contents of these new files.

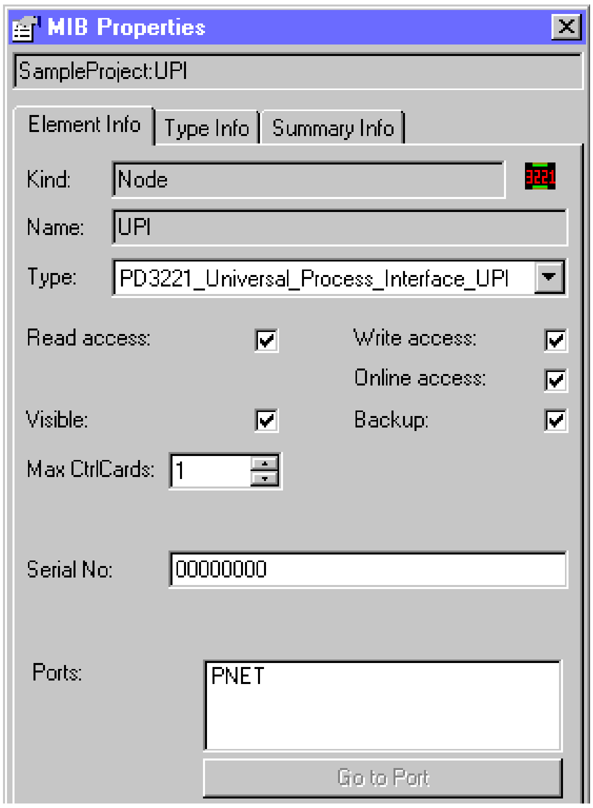
To save memory, *MIB Edit* can be disabled, using the [Disable MIB Edit] check box.

2.4 Properties Window

MIB Elements are used to describe nodes, nets and the different variable types, within the *MIB*. These elements can be of different *Kinds*. One *Kind* of element could describe a node, another *Kind* might describe an Array, and so on.

The *MIB* is a collection of such elements, which are referenced to each other as *Parent*, *Sub*, *Next* or *Previous* element.

Each element has a set of properties, describing its relationship with other elements and various related constants. These properties must be set up before the MIB can be used. The [Properties] window for a particular element is made available using the right mouse button menu, when that element is highlighted in the *MIBOCX*.



The [Properties] window is divided into two or three tabbed sheets, depending on the view mode. [Element Info], and [Type Info] are always shown. In *Variables View* (showing Nodes and variables), an additional tab sheet called [Summary info] is displayed, as shown in the figure.

If the *MIBOCX* is in *Variables View*, the Element info mainly applies to Nodes, for selection of the Node type, and also to select to which Network the Node is connected.

The [Type Info] tab provides information about the selected type, and cannot be modified from here. The [Summary Info] tab shows the access conditions that VIGO is using to access the selected variable. These access conditions are not necessarily the same access conditions that are defined for the element, as shown on the [Element Info] tab sheet. Furthermore, the [Summary Info] shows the *Internal address* as the sum of *SWNo* and *RegisterNo* for the variable.

If the *MIBOCX* is in *Types View*, [Element Info] shows the *Kind* of element and type Name of the selected item (red icons). For sub elements (blue icons), [Element Info] is used to select the type and relative location of the sub type. The [Type Info] shows the properties of the selected type and can be modified from here (modification can only be undertaken with red icons).

2.4.1 Properties in Element info

Kind

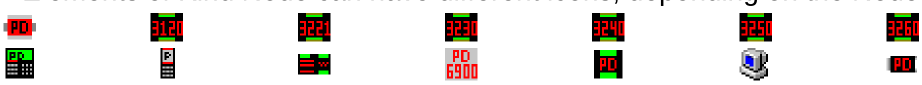
The *Kind* field shows the selected *MIB Element type*. Depending on the *Kind* actually selected, some of the following Properties can be applied, and are shown either in the [Element Info] or [Type Info] sheets.

The *Kind* is selected during the generation of new *MIB elements*. The *New* function is available in the *MIB Edit*'s right mouse menu. The *Kind* for a *MIB element* in the *MIBOCX* can only be changed, by firstly deleting it and then by adding a new *MIB element* of the required Kind.

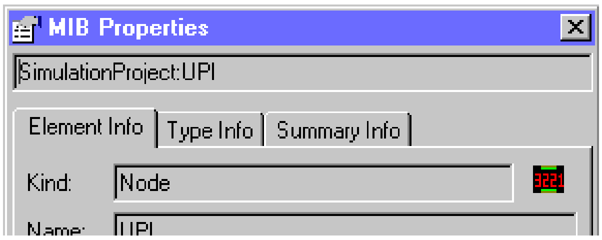
The different *Kind* icons are listed below. The first group being red icons, the next group being blue icons, and the last group being icons used for special elements.

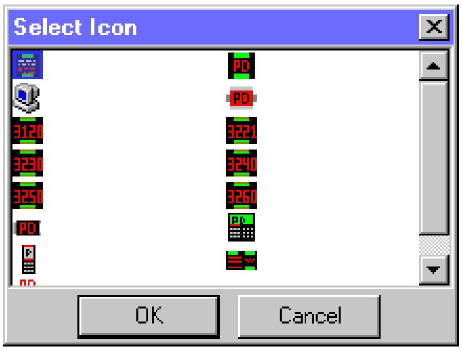
Project		
BasicType	NodeType	ChannelType
RecordType	Enumerated	ArrayType
BitArrayType	BufferType	SetType
String	BitMapType	PointerType
Procedure	Function	VirtualRecordType
VirtualArrayType	EnumeratedName	NetType
Node (*)		
Channel	Register	SoftwireNo
RecordField	Constant	Alias
VirtualName	Port	Net
PortTemplate	PortReference	Group
Property	StringValue	

* Elements of Kind Node can have different icons, depending on the NodeType:



When clicking on the Icon in the properties window, a new window for selecting an icon will appear, and the user can change the Icon for a specific Node.

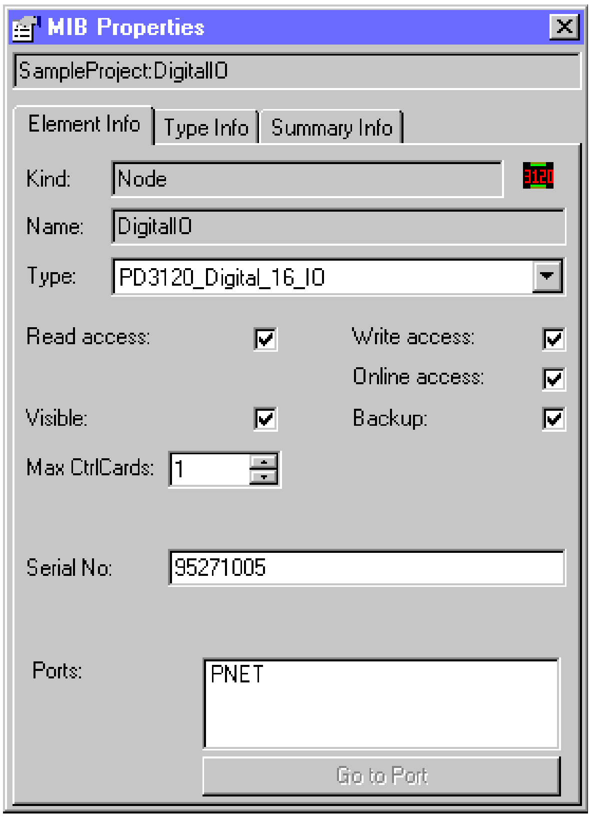




Name
The *Name* field holds the partial identifier for the variable or the type, which the *MIB Element* represents. The identifier can be modified, by first selecting the icon from the *MIBOCX*, then clicking the left-hand mouse button once on the name string, to enable the editing facility.

Type
The *Type* field indicates a type identifier for the variable or the type, which the *MIB Element* represents. In the Variables view, the Type can be selected or changed for *Nodes* and *Virtual Names*, by using the list box, and choosing one of the already defined types. In the *Types View*, the Type is used to select or change the sub types, which are to be included in already defined types.

Element access conditions
It is important to understand the principles of check box setting. When a check box is checked in the [Project Property] window, it will override the equivalent checkboxes in the *Nodes* properties of the project, and also those in *Channels* and *Registers*. Similarly, a ticked check box in a *Node* will override the equivalent check boxes in *Channels*, *Registers* and *SWNo*. The check boxes in *Channels* will override those that are equivalent in *Registers*.



The ability to Read from and Write to a variable can be defined in the [Read/Write access] check boxes.

The [Protected write access] check box should be checked when the variable is protected by the write enable bit in the module.

The [Online access] check box is used to indicate to VIGO whether the Project, Node, variable or a field in a record, is located in an external physical Node (checked), or in a simulation file (not checked). See also page 45, *Simulation mode*.

[Backup]. This check box indicates to the *Backup/restore* program to include this variable in the backup.

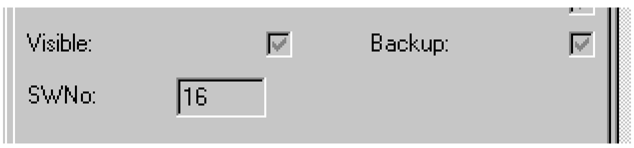
[Visible]. If this box is NOT checked, the variable (and the associated sub-variables) will not be shown in [MIB View].

[Max CtrlCards]**.** If the Operating system in the hardware device supports multiple communication blocks for *LongLoad* and *LongStore*, this value indicates how many communication blocks the PC should try to use.

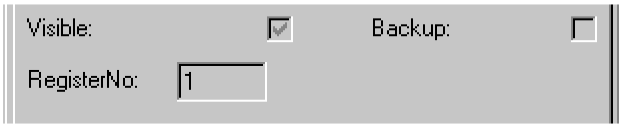
[Serial No:]. Identifies the Serial number for the Node.

[Ports] – Identifying the available *Communication ports* for this *Node Type*. You can jump to a *Port* by selecting it in the list box and click on the <Go to Port> button.

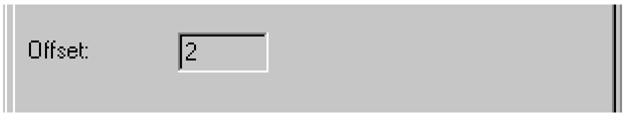
[SWNo:]. In P-NET, a *SoftWire* number (*SWNo*), defines a logical address to identify a variable. The [SWNo] must be set in accordance with the actual value within the node.



[RegisterNo] - holds the number, which references the location of a variable within a *Channel*. (*SoftWire* offset)



[Offset] - holds the offset, (in number of bytes) of a Record Field within a Record.

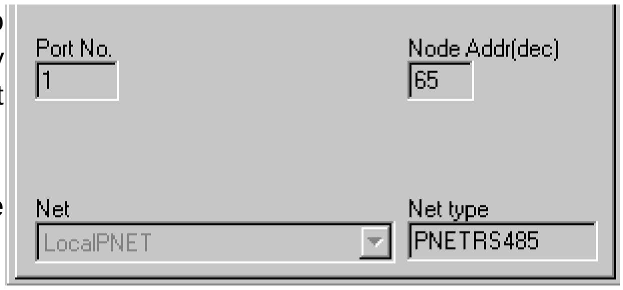


[PortNo]. Holds the *Port* number of the selected *Communication Port*.

[Node Addr(dec)]. This field holds the *node address*, which has been reserved for the node. This field is used together with the property of the *Net*, to form the complete route to the node. The *Node address* is a decimal value.

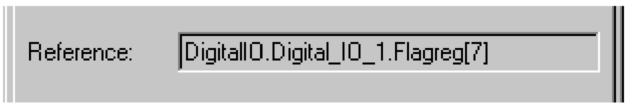
[Net] - is a list box, used to select to which *Net* the *Port* is connected. Only nets declared of a type that fit the port in question will be listed in the list box.

[Net type] – Identifies the type of the *Net*.

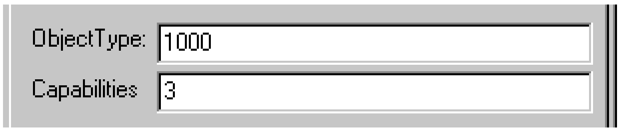


If the *Net*, to which the node is connected, is a Modem port, an additional field appears within which a telephone number must be set.
If the *Port* is a LAN port, the node address is either an IP or an IPX address (help on setting up an IP or IPX connection is found on the [Workspace] tab under [Driver parameters] for the IP or IPX driver). This field is also used by P-NET, when utilising the *Set P-NET Node Address* program.

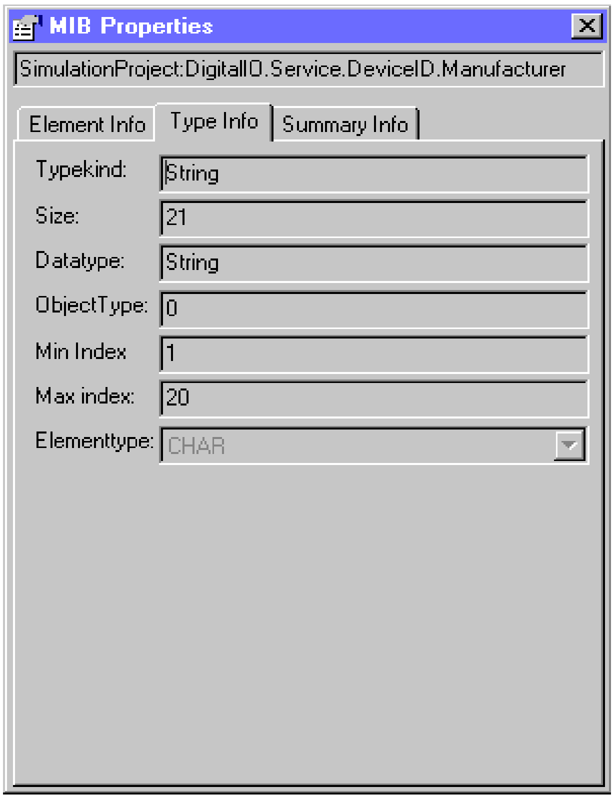
[Reference] is used to describe the full global identifier for an *Alias*. The reference can only refer to another identifier within the same project description, and therefore the Reference must be without the *Project Identifier*.



2.4.2 Properties in Type info



[Capabilities]. This property describes the capabilities of the destination Fieldbus Node. The coding of *Capabilities* is Node and Fieldbus dependent. Further information about the capabilities of a particular node, may be obtained from the vendor, or from the Node manual. Reference can also be made to the Appendix, for a list of the capabilities numbers for PROCES-DATA modules.



[Object Type] - is used for defining the availability of right mouse menu items. The *Object Type* adheres to the numbering identification as described in the Appendix.

[Size]. This property defines the size, in bytes, occupied by the variable within the node.

[Data type]. This field indicates the data type of the variable in the node.

[Min Index] and [Max Index] fields, indicate the minimum and maximum index of an Array.

[Elementtype] is used to describe the type of an Array element.

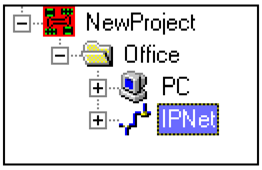
2.5 NET set up

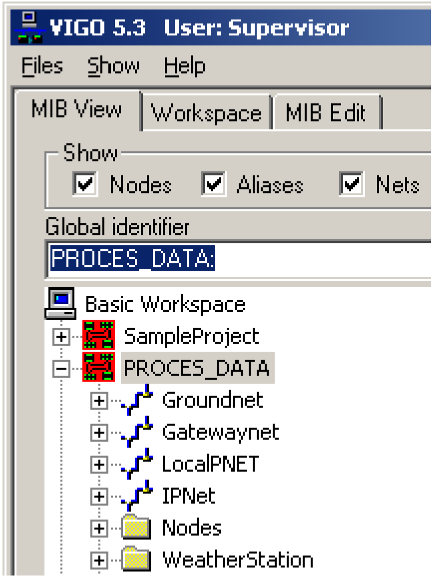
A *Project* may include a number of different communication networks, each having a variety of Fieldbus nodes connected. Each network (*Net*) is identified with a *Net Identifier*.

Adding *Nets* is done in the same way as adding other elements to the *MIB*.
When the *Net* is added, a type for the Net must be selected. This is done in the Properties window.

If a *Net* is placed in the outermost column, the net is said to be “Public”.

If a *Net* is placed within a *Group* object, the Net is “Private”.





Public Nets, which have been defined and named within a particular *Project*, are available to other projects within a given *Workspace*. *Public Nets* in different projects with identical names are therefore assumed to be connected. *Private Nets* having the same name as given in other *Projects* are assumed to be different *Nets*.

A pre-defined *Nettype* can be selected for use by a *Net*, such as a Local Area Network (LAN), Modem, Fieldbus etc. The type list box provides the means to select the required Net type for the net in question. Only *NetTypes* defined within the same project will be shown.

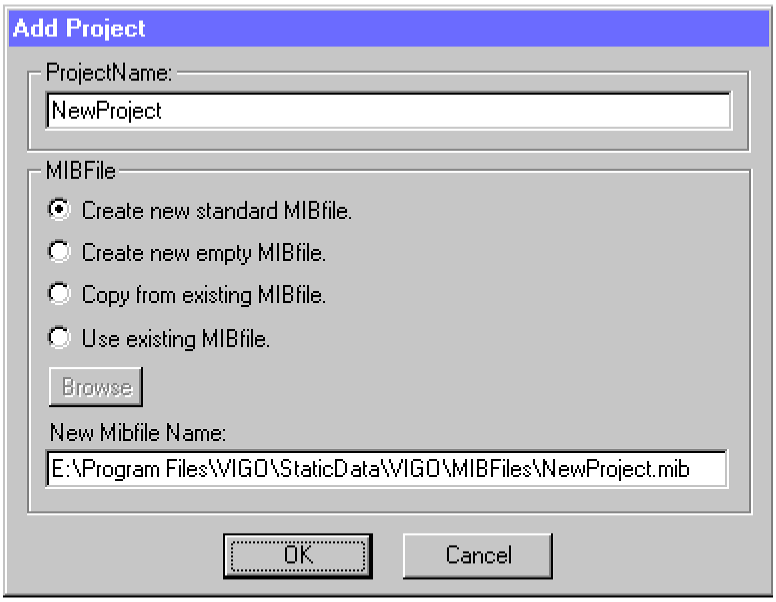
Any gateway or router must be specified in the *MIB*, including PC’s.

The information about all the *Nets* that are included in a *Project* is stored in the *MIB* file. This means that a *MIB* file can be located in a server, and shared with others.

2.6 Adding or Modifying Projects

Adding projects

A new project can be added to VIGO, by selecting the *Workspace icon* (a PC) or a *Project icon*, while in [MIB EDIT], and clicking the right mouse button. By selecting *Add Project* a window is presented as shown.



Choose a name for your project, and decide whether to do one of four things:

"Create new standard MIBfile" - which means that a description file called "PDTYPES.MIB" is copied and associated with the new project. This file contains several predefined Types, which can be used inside other Types, or for setting up new variables or a complete Project. New Types can also be added to this Project description file.

"Create new empty MIBfile" - which means that a description file called "EMPTY.MIB" is copied and associated with the new project. This file contains only simple predefined Types. New Types can also be added to this Project description file.

"Copy From Existing MIBFile" – means that a new MIB is created, which is based on a *MIB* that already exists. Amendments to this file will not affect the MIB from which this file originated.

"Use existing MIBfile" – which means that the *MIB* file associated with this project is not copied but uses one that has already been generated. An existing MIB file can be selected by using the Browse facility. It should be noted that any amendments made to this file would also be seen by other projects, which have been specified to use this *MIB*.

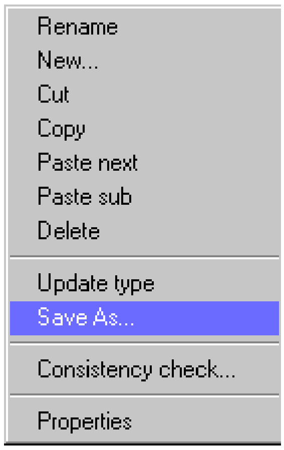
The actual name and location for a *MIB* file for a specific project can be found by selecting Properties on the right mouse menu for the Project in question. A new *MIB* file can also be selected by clicking the Browse button.

Modifying Projects
Selecting an element in a Project description is performed using the left-hand mouse button. Sub elements can be displayed if a plus [+] sign, associated with an element is clicked. Sub elements become hidden when a minus [-] sign is clicked. It is also possible to select an element of an array, by changing the element index number. Click once on the selected index number, type in a new index number and then click the mouse pointer elsewhere.

When modifying a Project description, the right mouse button menu is used to add, copy, paste and delete Elements within the *MIB*. This can be performed from within one Project description, or between different Project descriptions.

The right mouse menu is object oriented, which means that the functions in the menu vary, depending on the selected object/element. The principle is, to present only the functions and tools, applicable to the selected object.

If a *NodeType* icon is selected, or the *Project icon* is highlighted, the specific *NodeType* or Project can be saved, as a *sub-MIB* (*SMB*) file or as a *MIB* file. This is achieved by using the *Save as* menu item from the right mouse menu.



The saved Type can then be included in another Project description, by creating an Empty Type and using the right mouse menu item *Update Type*.

A *SMB* file generated by the *Process-Pascal* compiler can also be arranged to be included in a *Node Type* in the project description.

To ensure that all types within a Project description are declared before they are used, a *Consistency check* can be performed (selected by using the right mouse button).

The result of the *Consistency check* is automatically displayed in the form of a dialogue box indicating OK, or by providing a list of errors.

The Properties for any element are set from the [Properties] window, available from the right mouse button menu, when the element is selected.

2.7 VIGO access control

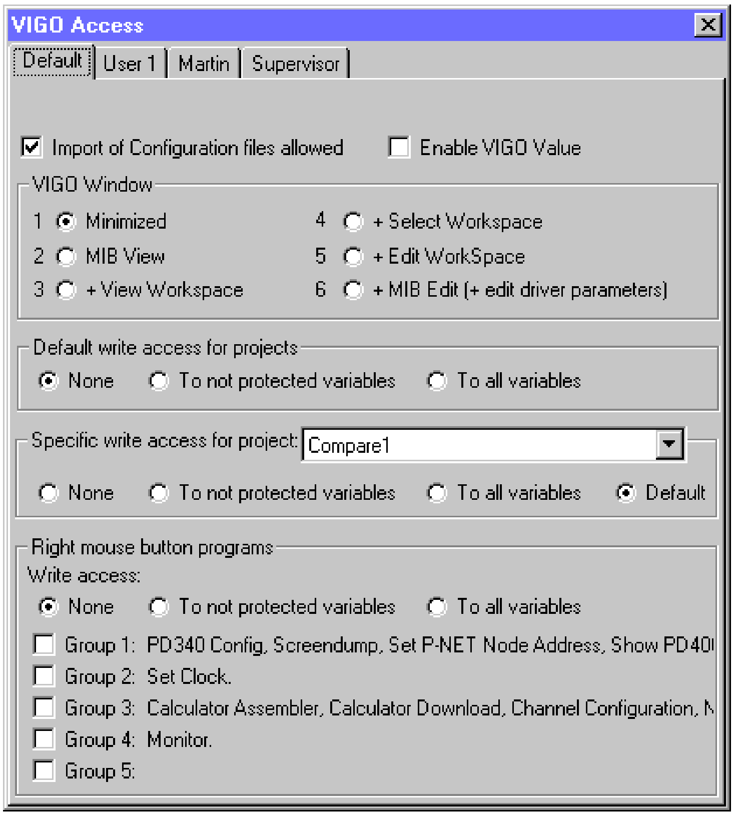
The access to the different functionality in VIGO can be limited according to the operators' responsibility. As an example, the daily operator is limited to only read/write non-protected variables. The local electrician is trained in how to replace and configure nodes. Therefore, he is allowed more functionally. The person who has installed the system can be the supervisor and is allowed unlimited access.

VIGO allows four types of users, each with their own level of access:

- Default
- User 1
- User 2
- Supervisor

Default user has no password protection.

The names "User1" and "User 2" can be changed. The access level relates to the VIGO window ([MIB View], [Workspace] and [MIB Edit]), write access to variables, and availability of programmes in the toolbox (right mouse button) menu.



[Import of Configuration files]
Determines whether or not the user will be able to import configuration files from the <Workspace> menu.

The [VIGO Window] function can be selected to be one of the following:

- "Minimized": The [VIGO window] cannot be opened; only shown minimized on the task bar.
- "MIB View": Only the [MIB View] tab is visible.
- "+View Workspace": Means that both the [MIB View] tab and the [Workspace] tab are active.
- "+Select Workspace": allows in addition, that different *workspaces* can be selected.
- "+ Edit Workspace": Allows in addition, that the *workspace* and *driver parameters* can be edited.
- "+ MIB Edit (+ edit drivers parameters)": Allows the user to edit the *MIB*, together with all other functions.

[Default write access for projects]
The write access for the Application programs that use VIGO, can be set individually for each Project, or a default setting can be used.

If [None] is selected, no writing to any variable is permitted, but they can all be read.

When [To not protected variables] is selected, normal unprotected variables can be written to.

When [To all variables] is selected, there is no limitation on reading or writing.

Generally, if the property of a variable in the *MIB* indicates no write access, the above selection will have no effect, and the result will be no write access to that variable.

Besides the [Default write access for projects], a [Specific write access for project] can be defined. In this way, the write access can be defined individually for each or a specific project. The write access definition for a specific project will overwrite the default definition.

The availability of certain [Right mouse button programs] can be limited for the different types of users. The right mouse button programs are divided into five groups, and each of the groups can be selected as available or not. The selection will be directly reflected on the right mouse button menu when selecting the elements in the *MIB* using the *MIBOCX*.

3 The Common Communication Interface

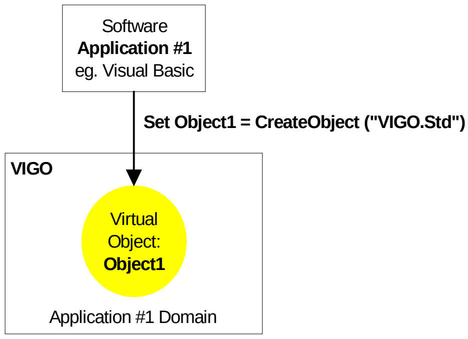
3.1 Single virtual objects

The following section describes the use of VIGO. The description uses as a basis, the programming language Visual Basic, but this can be translated into Visual C++, Delphi or Access forms without difficulty.
From a development point of view, it is a simple procedure to create an application, which has access to the physical objects.

There are only three steps to follow.
Step 1:
The first step is to create a virtual object recognised by a *virtual object identifier*. In this example the *virtual object identifier* is set to "Object1".

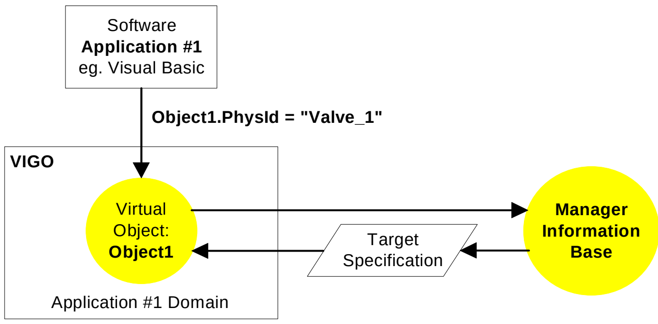
```
set Object1 = CreateObject("VIGO.Std")
```

From now on, the application will point to the Virtual Object by using the *Virtual object Identifier*.



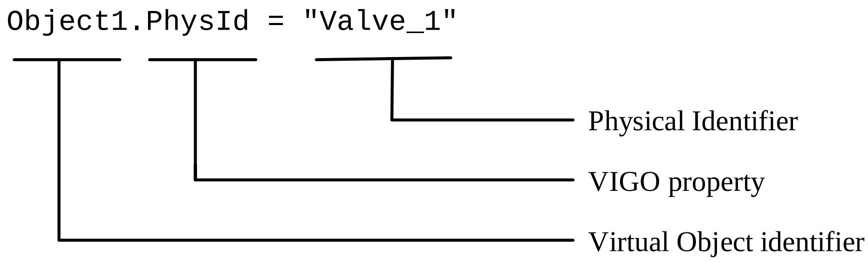
Step 2:
The next step is to associate the *virtual object* with the *physical object*.

The *virtual object* within VIGO, has a property called *PhysId*, which contains the *Physical Identifier*.



All information that is necessary to access the physical object will be obtained from the previously configured *Manager Information Base (MIB)*, by setting the *PhysId* property. See the figure above.

For example, a valve is the *Physical Object*, and labelled as 'Valve_1'. In the *Manager Information Base*, the 'Valve_1' is used as the *Physical Identifier* for this valve and points to the physical Valve.

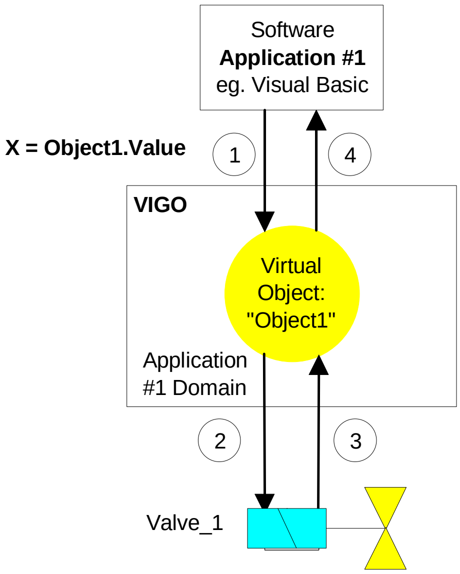


Step 3:
Once the *virtual object* points to the *physical object*, it is possible to operate upon the *physical object*. For example, it is possible to read or write to a variable.

In the case of the valve identified as Valve_1, it is now possible to get (read) or set (write) the state of Valve_1.

The valve state is read using the following code:

```
X = Object1.Value
```



By assigning the Value property of the *virtual object* "Object1" to the local application variable X, X will contain a Boolean value indicating the valve state of the *physical object*.

The valve is closed (set to OFF) using the following code:

```
Object1.Value = OFF
```

If the application needs to manipulate another physical object, step 2 and step 3 have to be carried out again.

For example, to read the value within a physical object uniquely identified as "FlowRate" and then to read a temperature in a different object, the following code would be used when using the same virtual object.

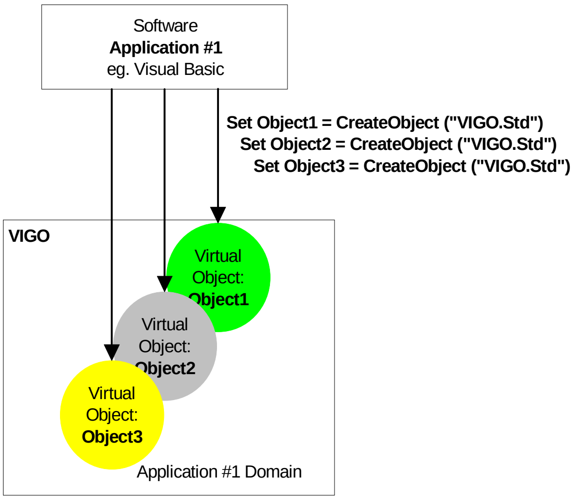
```
Object1.PhysId = "FlowRate"
Y = Object1.Value
```

The temperature is known to be found in the Project called “SampleProject”, in a Node identified as “UPI” having an analogue input channel, with the temperature value contained in the measurement register.

```
Object1.PhysId = "SampleProject:UPI.ANALOG_IN_1.ANALOGIN"
Z = Object1.Value
```

3.2 Multiple virtual objects

For an application that needs to communicate with many physical objects repeatedly, several objects can be created. For example, a device number, a valve position and a temperature are to be monitored continuously.



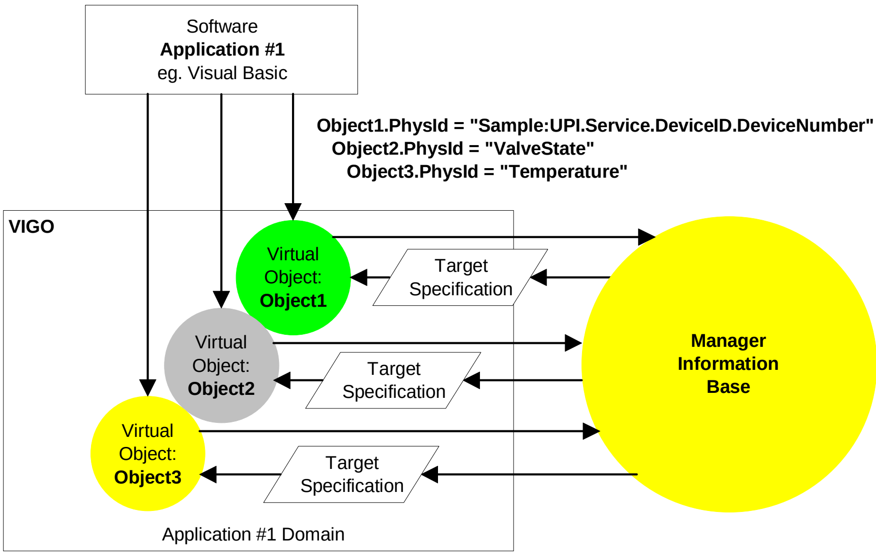
Step 1:
To create multiple virtual objects within VIGOSERV, The OLE function *Create-Object* must be called, for each instance of a new virtual object.

For example:
```
Set Object1 = CreateObject("VIGO.Std")
Set Object2 = CreateObject("VIGO.Std")
Set Object3 = CreateObject("VIGO.Std")
```

Step 2:
The second step is to get the virtual objects to point to the Physical Objects.

The association of virtual objects with physical objects only needs to be carried out once. This means that the procedure of calling the *MIB* with the global identifier, in order to retrieve the related information and apply it to the *Target Specification*, is only done once. Following this, the physical objects can be directly manipulated via the virtual objects, leading to a faster access, because all the network and address information is available within the *Target Specification*.

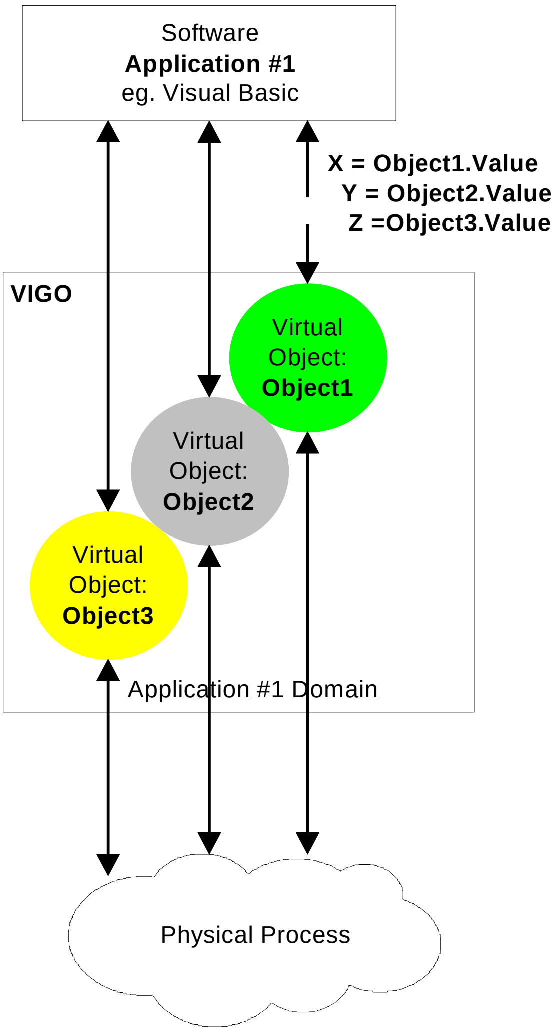
For example:
```
Object1.PhysId = "Sample:UPI.Service.DeviceId.DeviceNumber"
Object2.PhysId = "ValveState"
Object3.PhysId = "Temperature"
```



Step 3:

Now that the *virtual objects* are pointing to their associated *physical objects*, each of the three *physical objects* can be operated upon via the *virtual objects* (Object1, Object2 and Object3).

The current values for the valve position, device number and the temperature can now be monitored as shown below:



(Read device number)

```
X = Object1.Value
```

(Read valve state)

```
Y = Object2.Value
```

(Read temperature)

```
Z = Object3.Value
```

The first assignment will enable the device number to be ascertained from the application's local variable X. The second assignment will enable the valve position to be determined by the local variable Y. The third assignment will ensure that the variable Z contains the temperature value.

When it is required to update the values several times, only step 3 needs to be carried out again. No further calls to the *Manager Information Base* will be carried out, because all the required data are already contained within the virtual objects.

3.3 Application domains and shared physical objects

Each application accessing *VIGOSERV* has its own application domain within VIGO. An application can only access the virtual objects it has created.

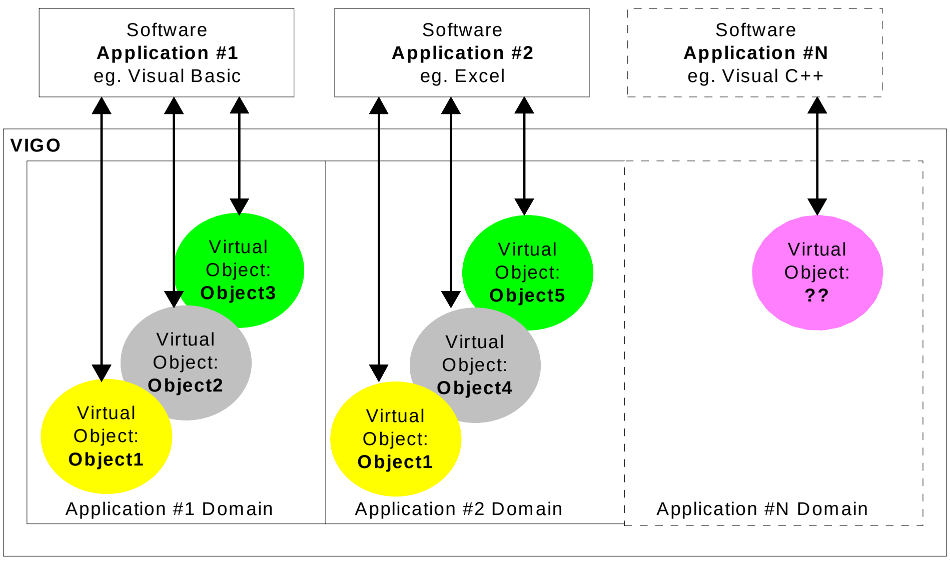
If a number of separate applications need to operate on the same physical object, each application has to create its own private virtual object, and point to the same physical object as the other applications. For example, two applications want to access the same measurement value, identified as "Temperature":

```
Application #1     (Eg: Excel)
Appl1 = CreateObject("VIGO.Std")
Appl1.PhysId = "Temperature"
```

```
Application #2     (Eg:Visual Basic)
Appl2 = CreateObject("VIGO.Std")
Appl2.PhysId= "Temperature"
```

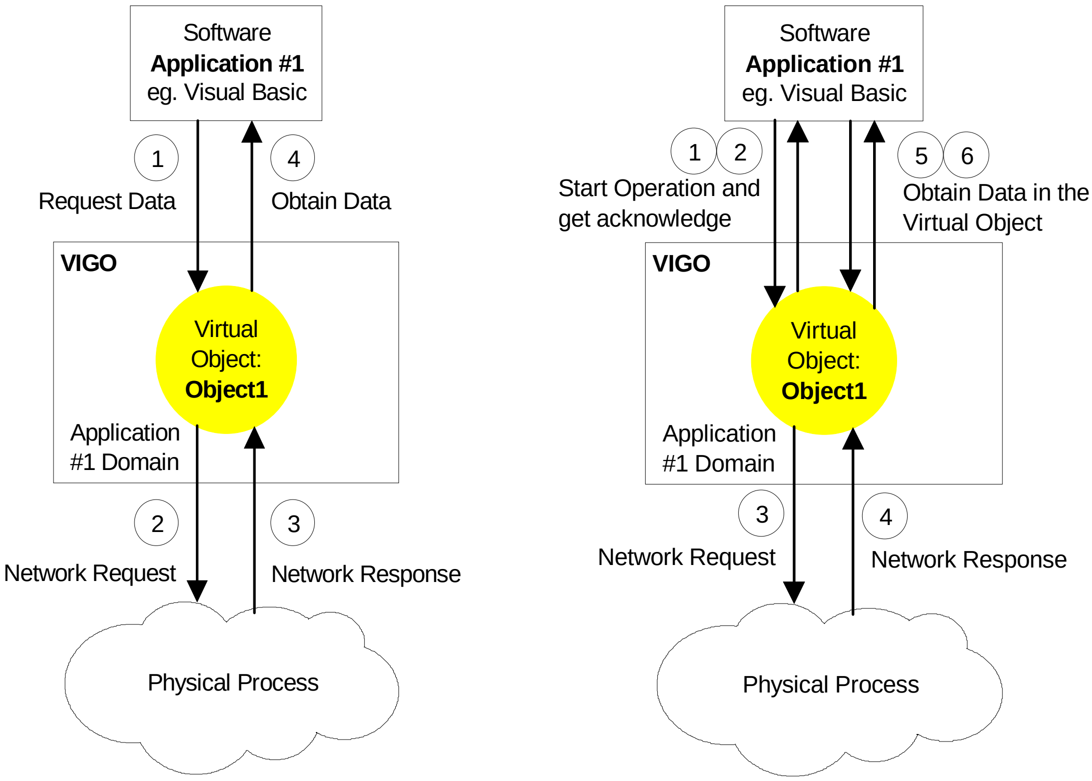
In this situation, both the *virtual objects* "Appl1" and "Appl2", are pointing to the same physical object.

Different applications may use the same identifier for the virtual object, but *VIGOSERV* will still contain a virtual object for each application, as illustrated in the figure below.



3.4 Two ways of accessing variables over the fieldbus

Two different modes can be used to access a physical object. The first mode is called *Direct-access***,** which sends a command via the network, i.e. a read command, waits for the result and then returns to the application when the command has finished and data has been obtained from the physical process system. The *Direct-access* approach is shown in the figure below, in the figure to the left.



The second mode is called *Buffered-access***,** which also sends a command via the network, i.e. a read command, but here VIGOSERV will return immediately to the requesting application before the command has finished and the data has been obtained from the physical object. After a while, *VIGOSERV* will return the result to the specific *VIGOSERV* property associated with that particular physical object, and it is now up to the application to read the result. This is shown in the figure above, to the right. To initiate a request using *Buffered-access*, two Methods called *DoRead* and *DoWrite* are used. The VIGO object must be created as a *VIGO.PRO* object to get access to these methods.

The idea behind *Buffered-access* is to make parallel execution possible. For example, by initiating the reading of ten different values from the process system, the latest sampled results held within *VIGOSERV* can be read later by the application, when required.

Example using *Direct- access*:
Read the Valve State

```
X = Object1.Value
```

Set the Valve State to OFF

```
Object1.Value = OFF
```

Example using *Buffered- access*:
Start obtaining the Valve State

```
Object1.DoRead
```

Later in the user application program, read the Valve State

```
X = Object1.InValue
```

Set the Valve State to OFF

```
Object1.InValue = OFF
Object1.DoWrite
```

If a new result has not yet been obtained by the virtual object property following a *DoRead*, by the time the application requires the use of it, the property will act like *Direct-access*. In this case, the return to the application will only occur when the result has been obtained.

3.5 Operating on Complex Variables

It is possible for the user application to operate on complex variables contained within a node. In order to do this, it is necessary to understand how the complex data are handled by *VIGOSERV*.

VIGOSERV is able to transfer a complex variable from a node into a virtual object, using a single request from the user application. To do this, the property *PhysId* already knows the variable is of a complex type, and by using the *DoRead* Method, the data is obtained in the virtual object.

For example, the complex variable "Coordinate" is composed of “X_Value”, “Y_Value” and “Z_Value”.

Coordinate
- X_Value
- Y_Value
- Z_Value

The *physical identifer* for the complex variable would therefore be:

```
Object1.PhysId = “Coordinate”
```

The next step is to load the complex variable from the node, for the virtual object Object1:

```
Object1.DoRead
```

The above-mentioned complex data is now available within the virtual object

To select one of the sub-fields within the complex structure, another property called *Sub-PhysId* must be used. The internal access property *InValue* is used to operate on these data elements in the sub-fields.

For example:
```
Object1.SubPhysId = "X_Value"
X = Object1.InValue
Object1.SubPhysId = "Y_Value"
Y = Object1.InValue
Object1.SubPhysId = "Z_Value"
Z = Object1.InValue
```

In a similar way to reading the entire complex variable with a single request, it is also possible to write the complete complex variable, using a single request from the user application. For example:

```
Object1.DoWrite
```

This way of handling complex data structures can reduce the total number of data transmissions on a network. This feature also gives the ability to obtain information, which is closely related, and time synchronised.

3.6 Error handling

VIGO provides extensive information about any errors that may occur during each of the communication tasks and during the use of VIGO.

When using properties or methods for an object, the VIGO system will set an *ErrorCode* Property to a value that corresponds to the result of the performed action. The *ErrorCode* may relate to errors in communication, conversion errors or errors from searching in the *MIB*.

The *ErrorCode* can be monitored by the application, following any access to a property or method for the object. If an error occurs, the *ErrorCode* is set to a unique number. If no error occurs, the ErrorCode will be SUCCESS (0x0000).

The application can also read the *ErrorCode* as a text string. The error string is contained in the *ErrorString* Property. When reading the *ErrorString* property, a translation of the *ErrorCode* into a text string is automatically performed. If an error occurs, the application can be programmed to take specific action, as shown in the examples below.

Error handling in VIGO follows the OLE Automation Exception rules contained within the Microsoft Windows™ OLE2 specification. When virtual objects are created, Exceptions are disabled.

It is possible to disable and enable exceptions using the following property for the object:

```
Object1.EnableExceptions    = True    (* or False *)
```

Below is a Visual Basic program example using Exceptions:

```
Sub Timer1_Timer ()
     On Error GoTo ErrorHandler ‘Exception
     TempText.Text = Object1.Value
Finish:  Exit Sub

ErrorHandler:
TempText.Text = Object1.ErrorString
Resume Finish

End Sub
```

The example above shows that the error information given by the object property *Error-String*, will be shown instead of the temperature "Object1.Value", in cases where an error occurs.

Below is a Visual Basic example, where the *ErrorCode* is monitored following an access to a read Property:

```
Object1.PhyId ="Temperature"
Temp = Object1.Value
If Object1.ErrorCode Is SUCCESS Then
   TempText.Text = Temp
     Else
   TempText.Text = Object1.ErrorString
End If
```

The error information given by the object property *ErrorString* will be shown in the text field identified by “TempText”, in cases where an error occurs.

3.7 Error messages and Error Files

The object property *ErrorString* contains a text string that describes the current error in plain text. The error may have occurred from within *VIGOSERV*, an *IDC*, a network (eg. P-NET Fieldbus, LAN etc.), the *MIB* that holds the project description, or the communication kernel *HUGO2*.

The error message is converted from an error code into an error message, which is automatically translated into the same language as that selected on the machine in which VIGO is running. The error messages are found in files, having the file-extension corresponding to the language definition specified by Microsoft. A VIGO standard installation provides texts for Danish and English errors. The text files are found in the VIGO program folder with the extension “DAN” and “ENG” respectively.

Copying an existing error text file into another file with the same file name can create an error text file for a new language, but with a file extension that matches the new language. The error messages can then be translated into the new language within the new file.

3.8 Simulation mode

One of the property settings of a VIGO object determines whether specific variables are located externally within an actual physical node on a network, or are held internally within the PC for simulation purposes. This property, called *OnLineaccess* can only be set when using *VIGO.PRO*.
The state of this property can be assigned from within an OLE compliant application programme written, for example, in Visual Basic or Delphi.

The corresponding property in the MIB, called *Online access* can also be set to specify whether variables are located externally or internally.

If the *OnLineAccess* property for a VIGO object is set to false (not-checked in the properties window), it means that the value of a particular variable can be read or modified from an internal location (on the PC), rather than relying on the fact that the physical node would normally have to be connected. This facility can be extremely useful during the commissioning and testing phases of a new project. Once one of the properties has been set, VIGO ensures that any reading or writing to a declared variable will be directed to the internal simulated variable. Any additional operations on this variable, such as, for example, to simulate the incrementing of a counter, would be arranged using a separate simulation-test program, which will run in parallel with the application being tested.

3.9 OLE Automation Interface

VIGOSERV has been designed in accordance with OLE Automation rules, defined for the Microsoft Windows™ environment. *VIGOSERV* gives access to object properties and methods, which can be used by any application supporting OLE Automation. For a virtual object, a property represents a variable and a method represents a procedure.

Each property offers a pair of functions, one to get (read) the property value and one to set (write) the property value. Therefore, when object properties are used in application programmes through *VIGOSERV*, one of two things are performed:

Set the value of a property (Write)
Get the value of a property (Read)

With most properties, their values can either be read or set according to the needs of the application. Properties that can be read or set are called read-write properties. Some properties only allow an application to get their value. These are called read-only properties.

A method performs an action on an object, and may or may not return a value. Methods may take a number of arguments. Arguments can be passed by value or by reference.

3.10 Performance

The performance of VIGO is not dependent on the user application, because VIGO is built for real-time communication, which is performed using interrupts. The communication task will not stop, even if the loading of a large file is taking place or a word processing program is being started.

VIGO is able to handle several hundred external data requests per second. However, performance depends on the efficiency of the underlying network driver and network performance.

4 Advanced VIGO Programming.

In VIGO there are two 32-bit OLE automation interfaces, structured as InProc OLE servers. These interfaces are called *VIGO standard* and *VIGO professional*.

VIGO standard is a reduced interface for ease of use, where the number of properties available is limited to the following: *PhysId*, *Value*, *ErrorCode*, *ErrorString*. All commonly used read and write operations can be achieved.
To establish contact with VIGO standard, an OLE automation object must be created, where the OLE name for the VIGO standard object is *VIGO.STD*.
Invoking a particular command, which depends on the programming language being used, performs this.

In Visual Basic, the command is:

```
Set Obj = CreateObject ("VIGO.STD")
```

In Delphi, the command is:

```
Obj:= CreateOleObject ('VIGO.STD');
```

In Visual C++, the command is:

```
Obj -> CreateDispatch("VIGO.STD");
```

VIGO professional is used for advanced programming, with an extended set of properties and methods.
To establish contact with VIGO professional, an OLE automation object must be created, where the OLE name for the VIGO professional object is *VIGO.PRO*.
Invoking a particular command, which depends on the programming language being used, performs this.

In Visual Basic, the command is:

```
Set Obj = CreateObject ("VIGO.PRO")
```

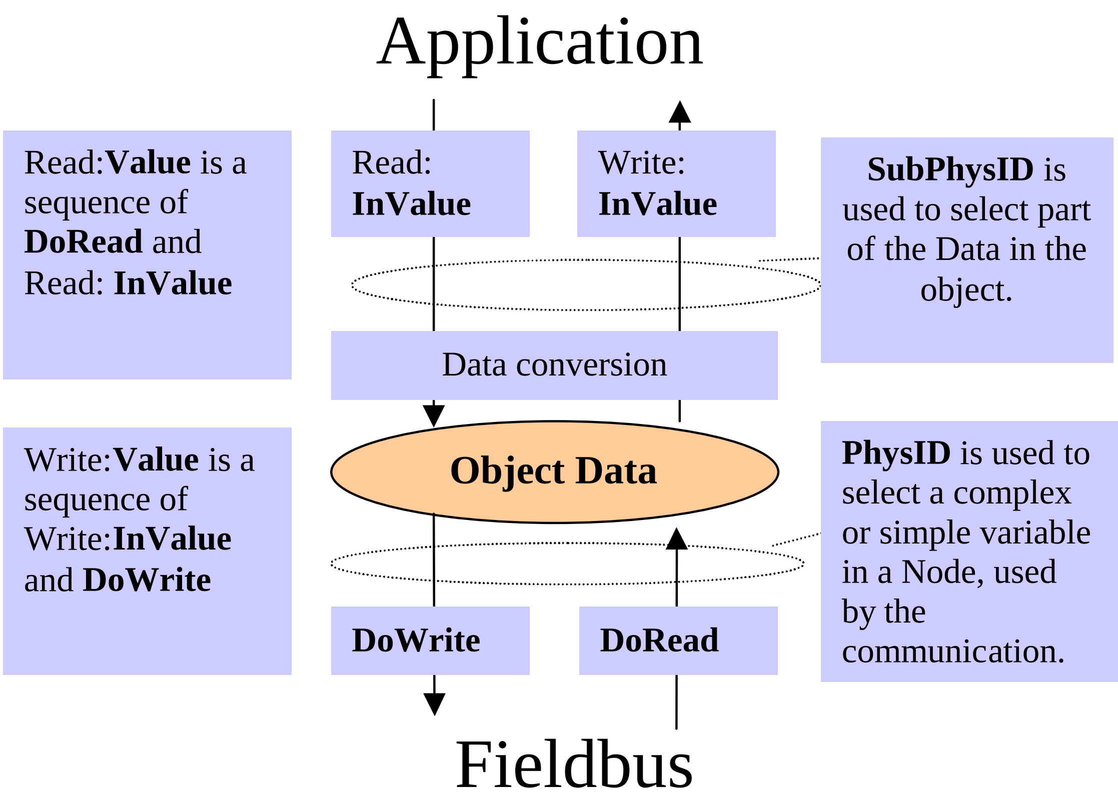
In Delphi, the command is:

```
Obj:= CreateOleObject ('VIGO.PRO');
```

In Visual C++, the command is:

```
Obj -> CreateDispatch("VIGO.PRO");
```

4.1 Properties and methods in VIGO professional



A Read of a variable within a node into an application is divided into two steps. First, the content of the variable is loaded into the Object Data. The next step is to convert the received data, and then transfer the converted data to the application. A similar situation occurs to Write, except that the first step is to convert data sent from the application, and then to store the converted data in Object Data.

4.1.1 PhysId

PhysId is used to relate the VIGO object to a variable within a Node. Assigning the *Global Identifier* to the *PhysId* will achieve this**.** The normal format for a *Global Identifier* is:

ProjectIdentifier:NodeIdentifier.ChannelIdentifier.Register…

If the *global Identifier* does not contain a *ProjectIdentifier*, the default project selected in the [Workspace] tab in VIGO will be assumed.

Writing to this property will start a search in the *MIB***.** This search will return all the necessary information about the variable, such as the Addresses and Offsets needed to access the variable within a particular Node.

Writing to the *PhysId* property will resize and clear the Object Data to zero.

The following Properties of an object are set according to the contents of the MIB:

NodeAddress, InternalAddress, Offset, BitNo, ICDNo, Size, ObjectType, DataType, NodeCapabilities.

DoRead/ DoWrite methods, or a read/write of the *Value* property, will use these properties when accessing Variables in Nodes via the Fieldbus.
The following assignments are also performed:

```
SubOffset= 0
SubDataType= DataType
SubBitNo= BitNo
SubSize= Size
```

These properties are used when reading/writing to the *Value* and *InValue* properties.

The result of the search in the *MIB*, will also set the following properties**:**
ReadAccess, WriteAccess, OnLineAccess

The *ErrorCode* will be set, depending on the result of the search in the *MIB*.

4.1.2 SubPhysId

SubPhysId is used to specify a sub-part of a complex variable already specified in *PhysId***.**

The *SubPhysId* must be assigned with the additional ".Identifier" , e.g. a record field identifier. The *PhysId* can be set to point to a complex array-variable. The *SubPhysId* property can then, for example, be used to specify a specific array element.

The *SubPhysId* property cannot hold: *Project, Node, Channel, Register or SwNo*, only Array [index] 's and Record field Identifiers. The *PhysId* must be pointing to at least a *SwNo* or a *Register* in a *Channel*, before *SubPhysId* can be used to select part of the *SwNo* or the *Register*.

Writing to the *SubPhysId* property will start a search in the *MIB***,** to get the value of the following properties: *SubOffset, SubDataType, SubBitNo, SubSize***.**
No other properties are affected, including Object Data.

If the *SubPhysId* is set as an empty string, the following assignments are performed:

```
SubOffset= 0
SubDataType= DataType
SubBitNo= BitNo
SubSize=Size
```

The *ErrorCode* will be set according to the result of the search in the *MIB*. The properties set by *SubPhysId* have an influence on the access, when using the *InValue* property and *Value* property.

Read/Write to *InValue*:

If *SubPhysId* is empty, the entire variable specified by the *PhysId* will be transferred between the application and the VIGO object.
If *SubPhysId* is not empty, only the part of the variable specified by the *SubPhysId***,** will be transferred between the application and the VIGO object.

4.1.3 InValue

InValue represents a sub-part of the variable in Object Data, as specified by the *SubPhysId*. The *InValue* property is declared as a Variant type, and can handle all kinds of data (Integer, Real, String, Arrays, etc.). A Read of this property will return the specified part of the internal variable in the VIGO object. Similarly, a Write will write to the Object Data. There is no Fieldbus communication.

When reading *InValue***,** VIGO converts the part of Object Data specified by the *SubDataType***,** and returns this data as a Variant**.** A part of this conversion task, is also to swap the bytes, according to the little/big endian principle (Intel/Motorola). The data-conversion is **ONLY** performed on simple data-types (Boolean, byte, integer, real...). It is therefore **not** recommended to use *InValue* on complex variables.
In the same way, a Write to *InValue*, will convert the Variant from the application to the correct data type appropriate for the variable in the field device, and will then store the data in the internal Object Data.

If the *SubPhysId* holds a part of an identifier, only the part specified by the *SubPhysId* will be transferred between the application and the object.

4.1.4 DoRead

The *DoRead* method is used to load a variable from a node into Object Data. The variable is specified by the properties set by *PhysId (NodeAddress, InternalAddress, Offset, BitNo, Size, ObjectType, DataType, IDCNo, NodeCapabilities).* The data is NOT converted.

DoRead starts the communication, and then immediately returns to the calling application. The Data will arrive later in Object data. If a new *DoRead* is started, within the same VIGO object, before the previous *DoRead* or *DoWrite* has been completed, the process will not return to the application, before the former *DoRead/DoWrite* is completed and the new *DoRead* has been started. *DoRead* cannot raise an exception due to communication errors. The application program must read *ErrorCode* before accessing the data.

The *DataReady* property will be “FALSE”, until a completed response from the Node is received.

4.1.5 DoWrite

The *DoWrite* method is used to transfer the data from Object Data into a variable in a node. The variable is specified by the properties set by *PhysId (NodeAddress InternalAddress, Offset, BitNo, Size, ObjectType, DataType, IDCNo, NodeCapabilities)*. The data is NOT converted.
DoWrite starts the communication, and returns immediately to the calling application. The response will arrive later. If a new DoWrite is started, within the same VIGO object, before a previous *DoRead* or *DoWrite* has been completed, the process will not return to the application before the former *DoRead/DoWrite* is completed and the new *DoWrite* has been started. *DoWrite* cannot raise an exception due to communication errors. The application program must read *ErrorCode* when *DoWrite* has completed.

The *DataReady* property will be “FALSE”, until the completed acknowledge from the Node is received.

4.1.6 Value

Value is a property, representing the variable specified by *PhysId* and the *SubPhysId*. The *Value* property is declared as a Variant type, and can handle all kinds of data (Integer, Real, String, Arrays, etc.).

A Read or Write to this property is equivalent to a read or write to the variable. VIGO returns to the calling application when the data is available, or when no response has been received after a timeout of a maximum of two seconds.

During a Write to *Value*, VIGO takes care of converting the Variant received from the application into the correct data type appropriate for the variable in the field device. It also swaps the bytes according to the little/big endian principle (Intel/Motorola). When reading *Value*, VIGO converts the loaded data into a Variant, using the properties set by *SubDataType***.** The data conversion is **ONLY** performed on simple data types (Boolean, byte, integer, real etc.). It is therefore **not** recommended to use *Value* on complex variables.

If SubPhysID is empty, a read from *Value* is the same as first calling *DoRead***,** followed by Reading *InValue*, and a write to *Value* is the same as Writing to *InValue***,** followed by calling *DoWrite***.**

If the *SubPhysId* is not empty, only the part specified by the *SubPhysId* will be transferred between the application and Object Data, but the **whole** variable will be transmitted on the network to/from the Node.

Note: Be careful if *SubPhysId* is not empty when using *Value* ! Some of the data in the object may be zero.

4.1.7 ExAnd (And)

The *ExAnd* property is a P-NET specific property, to AND a value to a variable in a P-NET module. The *ExAnd* property is declared as a Variant type. Writing to this property will perform a logical AND function between Data written to the property and the Data already in the variable. The result is stored in the Variable.

4.1.8 ExOr (Or)

The *ExOr* property is a P-NET specific property, to OR a value to a variable in a P-NET module. The *ExOr* property is declared as a Variant type. Writing to this property will perform a logical OR function between Data written to the property and the Data already in the variable. The result is stored in the Variable.

4.1.9 TestAndSet

The *TestAndSet* property is a P-NET specific Read Only property, to Test-And-Set a Boolean in a P-NET Node. Reading this property will start a special communication service that reads and sets the Boolean true in the Node. The result of the reading is returned to the application as a Variant.

4.1.10 ErrorCode

ErrorCode is a Read Only property of the type Integer (2 bytes). This property indicates whether an error has occurred, after accessing certain properties and methods.

ErrorCode = 0 indicates that there is no Error. If the *ErrorCode* <> 0, this indicates, that some aspect of a transfer has been found to be incorrect.

The following properties and methods will generate an *ErrorCode*:

> *PhysId, SubPhysId, Value, Invalue, DoRead, DoWrite, Download, Upload, ProgramState, ModelName, Revision, Programname, NodeAddress, Vendor, Start, Stop, Reset, Resume, Kill, SelectProgram, UnSelectProgram, DeleteDomain, TerminateDownload, ExAnd, ExOr* and *TestAndSet*.

The *ErrorCode* is not changed until one of the mentioned VIGO properties or methods is used again. The *ErrorCode* can be read as error text in the *ErrorString* property.

4.1.11 InformationInErrorCode

This Boolean property controls if Historical Errors are visible in the *ErrorCode*. If *InformationInErrorCode* is set to “TRUE”, then Historical Errors are also visible in *ErrorCode*. The default value for *InformationInErrorCode* is “FALSE”.

4.1.12 ErrorString

The *ErrorString* is defined as a Read Only string (max 150 characters). In reading this property, a text string will be returned, containing an explanation of the *ErrorCode*. The language of the error string depends on the language that has been selected on the machine. If the selected language is not supported in VIGO, English will be chosen.

If the *ErrorCode* is zero, the *ErrorString* will be empty.

4.1.13 DataReady

DataReady indicates whether a *DoRead*, *DoWrite*, *Upload* or *Download* cycle has finished. *DataReady* should be tested before a new *DoRead* or *DoWrite* is used on the same object, *DataReady* returns “False” when a *DoRead* or *DoWrite* method is in progress. *DataReady* returns to the calling application immediately.

4.1.14 SetVIGOMessage

This method is used to set up a message that will be sent when a *DoRead* or *DoWrite* on the particular object has finished.

The method is called using four parameters:

```
SetVIGOMessage(parameter1, parameter2, parameter3, parameter4)
```

Parameter1: Type "Long".
Handle of Window that the message will be sent to.

Parameter2: Type "Long".
MessageNumber that will be posted after *DoRead/DoWrite* has finished.

Parameter3: Type "Long".
Optional user data. This value will be posted along with the message itself as the *wParam* of the Windows message.

Parmeter4: Type "Long".
Optional user data. This value will be posted along with the message itself as the *lParam* of the Windows message.

An application can call this method before starting a *DoRead* or *DoWrite*. This way it is not necessary for the application to continuously call *DataReady* to check if the *DoRead/DoWrite* has finished. The *SetVIGOMessage* method must be called before starting the *DoRead* or *DoWrite*.

4.1.15 EnableExceptions

If *EnableExceptions* is set “TRUE”, all errors from VIGO will perform an Error Exception in the client program. The client program must then handle the exception.

The default value for *EnableException* is “FALSE”.

The Exception handler can then read the *ErrorCode* and *ErrorString*.

4.2 Properties set by PhysId

PhysId normally sets the following properties. These properties can be read, and by doing so, the *MIB* can be checked. In very special situations, the application program can, with care, write to these properties.

4.2.1 InternalAddress

The *InternalAddress* property is designed to hold the “internal address” of a variable in a Fieldbus module.
For P-NET, the “internal address” is a *Softwire* number, but it can also be a physical address in a module. If *PhysAddress* is true, a physical address is assumed.
The *InternalAddress* is used by *DoRead*, *DoWrite*, *Download* and *Upload*, and indirectly by *Value*.
InternalAddress is automatically set when writing to a *PhysId***.**

4.2.2 BitNo

The *BitNo* property is used to select a single bit in a BitArray. This property is used when accessing a field device. The *BitNo* is used by *DoRead*, *DoWrite* and indirectly by *Value***.** It is automatically set when writing to a *PhysId***.**
Writing to the *BitNo* property will copy *BitNo* to *SubBitNo*.

4.2.3 Offset

The *Offset* property is used to specify a byte offset within a complex variable in a Node. When a record field within a larger complex variable is to be selected, the offset specifies the position of the first byte of this field within the record. *Offset* is used by *DoRead*, *DoWrite*, *Download* and *Upload* and indirectly by *Value***.**

When the *Offset* property is changed, *SubOffset* is automatically set to 0.

The *Offset* is automatically set when writing to a *PhysId***.**

4.2.4 Size

The *Size* property indicates the size of the Variable (in bytes), to be accessed via the Fieldbus. It indicates to the communication stack the number of bytes to be transferred. It is also used to allocate memory for the VIGO object. *Size* is used by *DoRead*, *DoWrite* and indirectly by *Value***.**

Size is automatically set when writing to a *PhysId***.**

Writing to the *Size* property will copy *Size* to *SubSize*.

4.2.5 ObjectType

The *ObjectType* property holds an integer value associated with a particular type of object. The object type is used to identify whether the object (specified by *PhysId*), is a particular type of *Node*, *Channel*, *SwNo*, or *Register*. As an example, all the different *Channel types*

have different object type numbers. The *ObjectType* should reflect the actual data type in the field device.
An application program can use this property for testing the object type. For example, the download program can only work with a *Program Channel* as the target. A list of *Object types* for channels and modules can be found in the Appendix.

ObjectType is automatically set when writing to *PhysId***.**

4.2.6 DataType

The *DataType* property holds an integer value, which defines a particular data type. This object type should reflect the actual data type in the field device. It **is *not*** used for data conversion. A list of data types can be found in the Appendix.

DataType is automatically set when writing to *PhysId***.**

4.2.7 WriteAccess.

The property *WriteAccess* holds the status of a variable, selected by *PhysID***.** When writing to *PhysId***,** this property is set to “True”, if all elements in the *Global Identifier* have [Write Access] checked (Project, Node, Channel, Register or SwNo) else it will be set to “False”.

4.2.8 ReadAccess.

The property *ReadAccess* holds the status of the variable, selected by *PhysID*. When writing to *PhysId***,** this property is set to “True”, if all elements in the *Global Identifier* have [Read Access] checked (Project, Node, Channel, Register or SwNo), otherwise it will be set to “False”.

4.2.9 OnlineAccess

OnlineAccess is a Boolean property, used to indicate whether a *DoRead*, *DoWrite* or indirectly by *Value***,** shall access an external Node or an internal simulation file**.** If *OnlineAccess* is “True”, there will be communication on the Fieldbus network. If *OnlineAccess* is “False”, the data will be read from or stored in a simulation file. When writing to *PhysId***,** this property is set to “True”, if all elements in the *Global Identifier* have “*OnlineAccess*” checked (Project, Node, Channel, Register or SwNo), otherwise it will be set to “False”.

4.2.10 ProtectedWriteAccess.

This property reflects the state of the [Protected] checkbox of the variable selected by *PhysID***.** *ProtectedWriteAccesss* = “True”, means that the variable is protected by *Write Enable*. It is also used to prevent the user accessing variables with *ProtectedWriteAccess***,** when [VIGO Access] is not granting write access to write protected variables.

4.2.11 NodeCapabilities

The *NodeCapabilities* property informs the *IDC* which protocol limitations that shall be used for read or write to a specific Node.

The value in *NodeCapabilities* depends on the format that can be used on specific Fieldbuses. A list of the capabilities numbers for P-NET can be found in the Appendix.
NodeCapabilities is used by *DoRead*, *DoWrite* and indirectly by *Value***.** It is automatically set when writing to *PhysId*.

Please note, that the value of *NodeCapabilities* is returned as a string, converted to hexadecimal format. For example, a value of 32 is returned as a string holding the characters '20'.

4.2.12 NodeAddress

The *NodeAddress* property holds the full address of a Node on the Fieldbus. The format for the Node address must follow the *HUGO2* standard for building a Node address.
NodeAddress is automatically set when writing to *PhysId*.

4.2.13 MaxRetry

Reserved for future use.

4.2.14 PhysAddress

The *PhysAddress* property is of type Boolean. When the property is set "TRUE", VIGO will inform the PNET *IDC* to use physical addressing, instead of logical addressing.
The physical address used must be written in the *InternalAddress* property.
This property is set "False" when writing to *PhysId*.

4.2.15 IDCNo.

The *IDCNo* property must hold the number for the *IDC* that is appropriate for the target Node. *PhysId* normally sets this property**.** This property should only be accessed in very special circumstances.

4.3 Properties set by SubPhysId

SubPhysId normally sets the following properties. These properties can be read, and by this means, the *MIB* can be checked. In very special situations, the application program can, with care, write to these properties.

4.3.1 SubBitNo

The *SubBitNo* property is used to select a single bit in a BitArray. This property is used when the application exchanges data with the VIGO object, and the data type is a Bit array. The *SubBitNo* is used by *InValue* and indirectly by *Value***.**

SubBitNo is automatically set when writing to *PhysId* or *SubPhysId*.

4.3.2 SubOffset

The *Suboffset* property is used to specify the byte offset within a complex variable, located in a VIGO object. When a record field within a larger complex variable is to be selected, the offset specifies the location of the first byte of this field within the record. The *SubOffset* is used by *InValue* and indirectly by *Value***.**

SubOffset is automatically set when writing to a *PhysId* or *SubPhysID*.

4.3.3 SubSize

The *SubSize* property indicates the size (in bytes), of the selected part of the Variable in the VIGO object. This property is used when the application exchanges data with the VIGO object. The *SubSize* is used by *InValue* and indirect by *Value***.**

SubSize is automatically set when writing to *PhysId* or *SubPhysId***.**

4.3.4 SubDataType

The *SubDataType* property holds an integer value that identifies a particular data type. The number should reflect the actual data type of the selected part (*SubPhysId*) of the variable in the VIGO object. This *SubDataType* is used by the conversion function when the application exchanges data with the VIGO object. A list of data types can be found in the Appendix. When writing to *PhysId* or *DataType*, the new value of *DataType* is copied to *SubDataType*.

The *SubDataType* is used by *InValue* and indirectly by *Value***.**

4.4 RACKS (MMS) related properties and methods

4.4.1 ProgramState

The *ProgramState* property follows the MMS standard, and is used in *Program Channels*. These channels must be of *ObjectType* 11 otherwise *ProgramState* cannot be used.

Further information about *ProgramState* can be found in the manual for any of the nodes that have a *Program Channel*.

4.4.2 ProgramName

The *ProgramName* property follows the MMS standard, and is used in *Program Channels*. These channels must have the *ObjectType* property set to 11 otherwise *ProgramName* cannot be used.

Further information about *ProgramName* can be found in the manual for any of the nodes that have a *Program Channel*.

4.4.3 FileName

This property is of type string. It is used to hold a path to a file when *Download* or *Upload* is called.

4.4.4 Progress

Progress is a Read Only property of Integer type that holds the number of bytes transmitted in percentage of the total number of bytes, to be transmitted. The *Progress* property is valid with *Download* or *Upload*.

The *Progress* property is very useful, for indicating to the user that the data transmission is still running.

4.4.5 StopSequence

The *StopSequence* method can stop a data transmission, which has been started with *DoRead*, *DoWrite*, *Download* or *Upload*.

4.4.6 Download

This method is used to *Download* a program to a standard *Program Channel*. The paths must be set prior to its use, in *Filename*.

4.4.7 Upload

This method is used to upload a program from a standard *Program Channel*. The paths must first be set in *FileName*. **This method is not implemented in VIGO 5.0.**

4.4.8 DeleteDomain

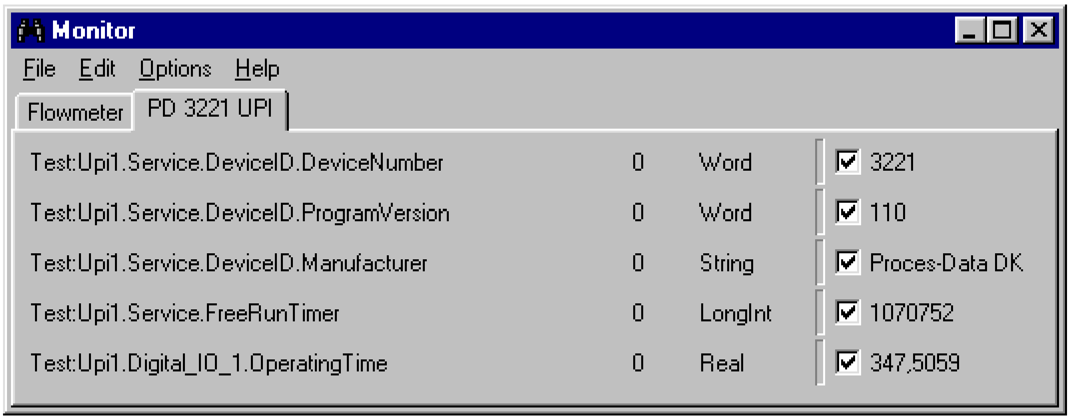
This method is used to delete the selected domain in a standard *Program Channel*. The domain must be selected first.

4.4.9 Start, Stop, Resume, Reset, Kill

These methods can be used with a standard P-NET Program Channel. They provide equivalent name functions, as described in the standard for the *Program Channel*.

5 Monitor

The *Monitor* program is a service tool for monitoring fieldbus variables, and to enable parameters to be configured within Fieldbus based control systems. The *Monitor* can be used to both display and modify the value of variables. Variables are usually identified using a globally recognized name, called the Physical Identifier (*PhysId*).



The *Monitor* can display the value of many variables at the same time, each one allocated to a separate line, these lines can then be grouped on different tab sheets.

Specified variables can be located within different projects, nets and nodes. Normally, each line is divided into three fields: [Physical Identifier], [Type] and [Data]. An optional [Offset] field can also be shown. The width of the [Physical Identifier], [Offset] and [Data] fields, can be adjusted by dragging the vertical line shown between the [Type] and [Data] fields, and by re-sizing the window.

Starting the *Monitor* from within VIGO will automatically create a monitorline holding the selected *PhysID*.

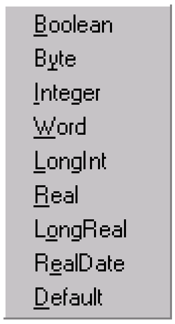
The [Physical Identifier] field is used to define which variable to display. Double clicking a line in this field will cause it to change into an editing field, which also includes a <MIB> button. The contents of the Physical Identifier field can now be keyed in manually, or alternatively, by pressing the <MIB> button. The required variable can now be selected from the project structure. The format of a *Physical Identifier* entry would normally consist of a "project name" followed by a colon, then the rest of the identifier, which includes the node name and the variable name. For example: "Test:UPI1.SERVICE.WDTIMER". The project identifier and the colon can in fact be omitted. It is then assumed that the *Default Project*, as previously specified, will act as the project identifier.

Manual input of a *Softwire* number is performed by formatting the *Physical Identifier* as: ‘Project identifier:Node identifier.Softwire number’. For example, ‘SampleProject:UPI1.18’ (or UPI1.$12 in Hex).

The *MIB* is used to convert the *Physical Identifier* into the actual "address", required to access the variable.

5.1 The Type Field

Under normal circumstances, the *MIB* also returns the data type of the variable in question, which is then automatically inserted into the [Type] field. In case the *Physical Identifier* is inserted manually as a *Softwire* number, the data type must also be set manually. Clicking the right mouse button from within the [Type] field will produce a list of available data types for display. Clicking on one of these will insert the data type name into the [Type] field, and the displayed variable will be formatted as such.



If the *Physical Identifier* specifies an Array or Record, the Type field shows "-----", because it is not possible to present the complete value of a complex variable.
When accessing a variable using a *Softwire* number, or part of a complex variable using an offset, the data type must be selected manually. As previously described, this is done by clicking the right mouse button within the [Type] field, and then selecting the appropriate data type. If the selected data type differs from the data type specified in the *MIB*, it is shown enclosed in brackets, e.g. [LongInt], and the readings seen may be unpredictable. By selecting “Default”, the type specified by the MIB is used.

5.2 The Offset Field

The value within this optional field is always assumed to be zero, for variables of simple data type. For variables of complex type (array or records), the [Offset] field can be used to manually define an offset, in bytes, to a sub element of the variable. Double clicking within the [Offset] field, enables the offset value to be changed from zero. However, this means that the *Physical Identifier* no longer fully represents the data value displayed.

5.3 The Data Field

This field displays the value of the variable, pointed to by the [Physical Identifier] field. A check box in the field is used to enable automatic updating of data values. The refresh rate can be specified in the <Options> menu. The default rate is two updates per second.

The [Data] fields Edit mode is activated by double clicking on the field. The [edit] field will in edit mode change it’s appearance. In edit mode the user can manually change the value of the selected data.

The readout format can be selected to be in Decimal (default), Hexadecimal or Binary, for the following data types: Byte, Integer, Long Integer and Word. The number of digits displayed after the decimal point can be selected for variables of the Real data type. The selection can be made by using the right mouse button, when the cursor is pointing to a particular value in the [Data] field.

If an error message or any other information relating to a variable is received, these are appended in the [Data] field. As default, only error messages are shown. Display of additional information can be enabled from the <Options> menu.

5.4 Main menu

5.4.1 File

The <File> menu contains the following items, which all relate to Monitor screen layouts.

<File | New Page>

The New Page function creates a new tab sheet with five empty monitor lines. The <Edit | Insert> menu item is used to add extra lines to the active tab sheet.

<File | Delete Page>
The Delete Page function removes the currently selected tab sheet.

<File | Clear Monitor>
The Clear Monitor function will remove all pages and create a new page containing 5 empty lines.

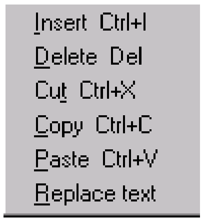
<File | Open>
The Open function reloads a monitor layout from a previously saved file. The included identifiers will automatically be converted into the 'address' needed to access the variable. This is performed using the contents of the *MIB*, and will therefore reflect any changes that have been made to the *MIB* since the last screen save.

<File | Save>
The Save function will store the screen layout parameters together with the list of included Identifiers in a file with the file extension ‘mon’.

<File | Save as>
The Save as function provides an opportunity to store the screen layout in a file with user-defined path and name. The file is saved with the file extension ‘mon’.

<File | Exit>
Exit closes the program. When the *Monitor* is closed, the current *Monitor* layout is automatically saved, as the default screen layout. Next time the *Monitor* is started, the default *Monitor* layout is automatically loaded.

5.4.2 Edit



The edit functions are used to customize a screen layout. They apply to the currently selected line. A monitor line can be selected by clicking on the line with the left-hand mouse button. Selected lines are displayed with all text as bold. It is possible to select multiple lines by following one of the tree procedures described below.

- Pressing the <Ctrl> button and clicking on the lines in question enables the selection of multiple lines. The lines will on selection be displayed as lowered.
- The selection of continuous lines is possible when a line is selected and the <Shift> button is pressed followed by a selection of another line. This will cause all of the lines between the two selected lines to be selected. The lines will on selection be displayed as lowered.
- Pressing <Ctrl>+<a> selects all *Monitor* lines on the currently selected tab sheet.

Notice that changing the active tab sheet will cause all selected *Monitor* lines to be unselected.

<Edit | Insert>
The Insert function will insert an empty *Monitor* line over the selected line placed nearest to the top of the active tab sheet. If no line is selected the new line will be placed at the top of the active tab sheet.

<Edit | Delete>
The Delete function will delete the selected *Monitor* lines.

<Edit | Cut>
The Cut function will delete the selected *Monitor* lines, and save them on the clipboard.

<Edit | Copy>
The Copy function will save a copy of the selected *Monitor* lines on the clipboard.

<Edit | Paste>
The Paste function will insert monitor lines, previously saved on the clipboard, above the selected line.

<Edit | Replace text>
This menu starts a search and editing form, that enables the searching and replacement of text in the [PhysID] fields.

5.4.3 Options

The default fields in a *Monitor* line are: Physical Identifier, Type and Data. However, it is possible to customise the lines using the Options menu.

The option menu can also be made available, by clicking with the right mouse button on the *Monitor* lines and selecting the desired menu or submenu item from the popup menu. The choices in the popup menu will only influence the currently selected *Monitor* line.

<Options | Rename Page>
Rename Page is used to change the caption on the tabsheet.

<Options | Show Offset>
Show offset is used to enable the Offset field. The Offset field is disabled as default.

<Options | Show types>
Show types is used to enable the Type field. The Type field is enabled as default.

<Options | Enable Info>
When reading or writing to a variable, an error message and/or other information relating to the variable, may be received from the node.
An error message is always appended to the Data field. Activating the <Enable info> function, will also append any additional information to the Data field.

<Options | Refresh rate>
The refresh rate is defined as the number of full screen (all *Monitor* lines) updates per second. The refresh rate has a default value of 2 Hz, meaning that all values in the monitor lines will be updated twice each second. The refresh rate can be set to one of the following values: No Update, 1, 2, 5 and 10 Hz. Selecting a low frequency, will reduce data traffic on the bus.

<Options | Field Colors>
The menu item contains a sub menu that allows the user to change the colors of each field type.

<Options | Font>
The menu item allows the user to change the font selection in the *Monitor* lines.

<Options | Default Settings>
The menu item returns the *Monitor* lines to the default settings.

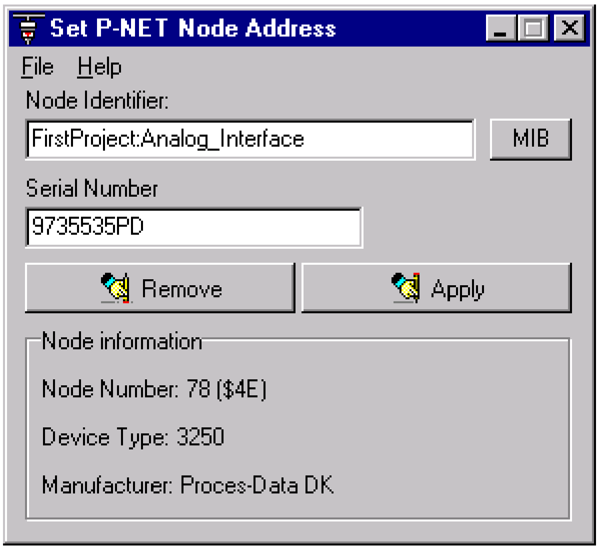
5.4.4 Help

Guidance in using *Monitor* is available from the Help menu. It is also possible to obtain guidance on a specific subject by pressing <F1> after selecting an item in the *Monitor*.

6 P-NET Tools

A number of tools that are specifically used in conjunction with the P-NET Fieldbus, are available in the VIGO program package. Some of the more general-purpose tools that can be used with all P-NET standard modules are described in this chapter.

6.1 Set P-NET Node Address



Each P-NET node that is located within a single bus segment must be configured with a unique node address. P-NET nodes are normally shipped from a manufacturer with the node address set to zero. Since node address zero is not permitted to be used for normal communication, the connection of such a node will not interfere with any of the other nodes already running on the network. When a new node is connected to the network, the desired node address can be set, by using a special feature of P-NET. Sending a broadcast message to all nodes, consisting of the new node address, together with the serial number of the node in question performs this.

The purpose of this program, is to enable the setting of the node address within a physical node, by means of using it's serial number.

This program is launched from the MIB browser *MIBOCX*, by selecting it from the right mouse button menu, when a Node is highlighted.

Selecting a different Node in the MIBOCX (by activating the [MIB] button), will automatically update the [Node Identifier] in the *Set P-NET Node Address* program window, and will display data about that node.

If the node is recognised as a P-NET master module, a [No. Of Masters] field is also shown. This indicates the maximum number of masters currently allowed to be connected to the network segment.

The node in question must be included in the project description in the *MIB*, and it's properties must also be set correctly, including the desired node address.

When the program is opened and a node is selected, the following four situations can occur:

1: If the node specified in the [Node-Identifier] field cannot be found at the node address as specified in the MIB, the serial number of the should be keyed in, and then the <Apply> button should be pressed. The function of the <Apply> button is to send a broadcast message to all nodes, commanding the node with serial number xxxxxxx, to set its node address to the attached value. If a node with the specified serial number is found, the [Node info] for the module in question will be shown.
If contact with the node cannot be established, the [Node info] field will display "No contact with Node". If this is the case, it should be checked that the serial number of the module is correct, and that the module has been correctly connected to the network.

2: A node is found on the network, and the serial number and the [Node info] is automatically displayed for that node. If this information corresponds to what is required, as specified in the *MIB*, the communication parameters for the node are correctly set up, and no further action needs to be taken.

3: A node is found, and the [Node info] is shown, but the node is not the expected node as specified in the *MIB*. This indicates that the node has been configured with the wrong node address, and it must be removed to ensure future communication integrity. By pressing the <Remove> button, the node is removed from the network as far as communication is concerned. This is done by setting the node address to zero. The [Node info] will now display "No contact with Node" and situation 1 will now apply.

4: A node is found, and the [Node info] is shown, but the node is not the expected node as specified in the *MIB*, or random communication errors occur. This could mean that two or more nodes are configured for the same node address, and these nodes should be removed and re-configured to maintain communication integrity.

Guidance for selecting Node Address and No of Masters for a Project
The P-NET node address can be in the range from 1 to 125. The No. of Masters can be in the range from 1 to 32. The lower numbers are reserved for Master modules. Node addresses for Slaves must always be higher than the No. of Masters.
If for example a project consists of 5 Master modules and 15 Slave modules, the No. of Masters and Node Addresses could be selected in the following way:

> No of Masters = **6** (one master number is reserved for future extension)
> Node Address for the Master modules are then in the range from **1** to **6**
> Node Address for the Slave modules are then in the range from **7** to **125**

6.1.1 Help

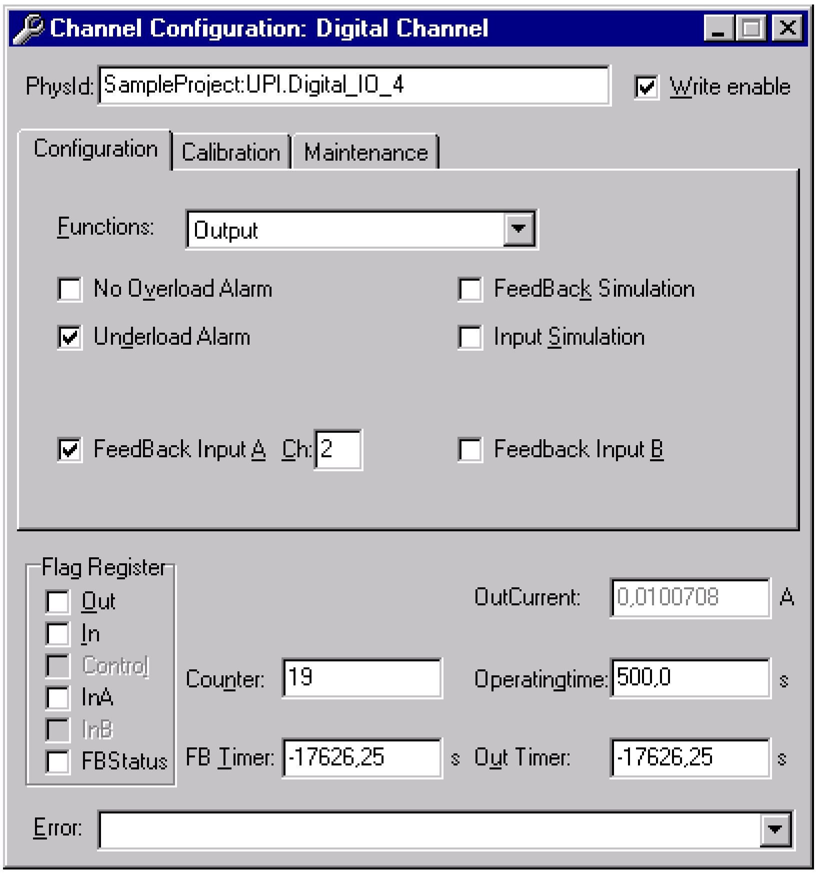
Help on the use of the *Set P-NET Node Address*, is provided from the <Help> menu. The help file consists mainly of parts of this manual.

6.2 Channel Configuration

The purpose of this utility program is to enable a node channel to be configured, maintained and monitored. It is launched from the *MIBOCX* using the right mouse button menu, when a channel is selected.

The *Channel Configuration* program is designed to recognise a number of standardised channels.

This provides the user with a convenient way of configuring the various channels, which make up a module.
The Channel configuration window is divided into three sections. The upper section contains the [PhysId] field, which displays the identity of the Channel. The [Write enable] check box is common for the entire node and must be checked, to allow the contents of configuration registers to be changed.



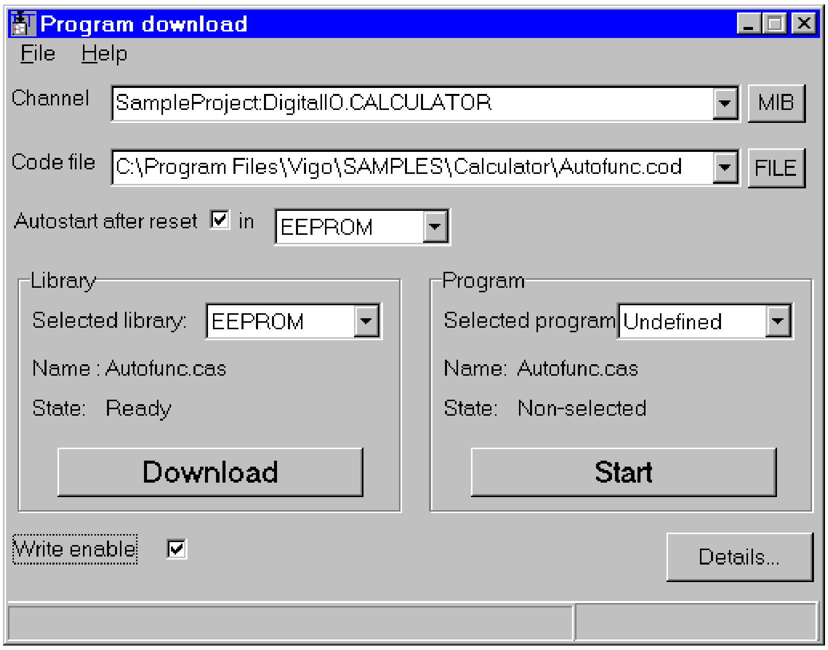
The middle section consists of a number of tab sheets, each containing a formatted view of the various configuration registers, enabling ease of amendment or monitoring.

The lower section provides a display of real-time values, which are specific to the selected channel type. Although the values shown depend on serviceable communication and the state of the current process associated with the channel, many can be amended locally, using the PC keyboard or mouse, e.g. resetting a counter to zero, or changing the state of a digital output.

The ability to display a particular channel configuration screen, depends on that channel type being included and selected within a node already defined in the *MIB* project file.
Screens are available for the following Channel types: Service, Digital I/O, AnalogIn, PID, AnalogOut, Weight, Communication and Program Channel.

6.3 Program Download

The purpose of this program is to provide the means to download program code, i.e. Process-Pascal code or Calculator Assembler code. The code can be downloaded to all modules supporting a P-NET standard Program Channel, such as the PD600 series and the PD5000 series of controllers from PROCES-DATA A/S. *Program Download* is called from the *MIBOCX* using the right mouse button menu, when a Program Channel is selected.1



The *Program Download* utility also supports the downloading of Calculator programs, to other modules supporting the P-NET standard Program Channel, such as the PD3120 module.
When this program has been launched via the *MIBOCX*, the *Channel identifier* is automatically inserted. Pressing the [MIB] button and selecting a new Program Channel can change the identity of a Channel.
A File browser can be opened for selecting a *Code file* by pressing the [FILE] button The node to which a program is to be downloaded, must first be defined in the *MIB*, before a download can proceed.

Clicking the [Details] button opens a window, from where the selected program can be stopped, started, killed etc. In addition, the Actual size, Max size, Code type, and Version of the program in the selected library can be seen.

6.3.1 Channel

The [Channel] combo box is used to insert the name of the channel, to which a program is to be downloaded. The selected channel must be a standard *Program Channel*, of object type 11. The object type is defined in the *MIB*. If the selected channel is not of object type 11, a message box will display "Error in PhysId name".

A channel identifier can be selected using four alternative methods. When this program has been launched via the *MIBOCX*, the channel identifier is automatically inserted. The channel Identifier can also be inserted from the *MIB* by clicking the [MIB] button, and then double clicking on a channel name within the *MIBOCX*. It can be included as a start up parameter for *Program Download*, or it can be directly keyed in into the combo box.

When a new channel identifier is inserted, it will always become the highlighted item in the [Channel] combo box list. The selected item will be inserted at the top of the list. If the selected channel was not previously included in the list and the list is full, the oldest channel identifier will be deleted. Once a channel identifier is included in the combo box, it can be easily selected from the list, which can hold up to 6 channel identifiers.

6.3.2 Code file

The [Code file] combo box is used to specify the file to be downloaded. The extension is normally ".COD" for e.g. *Process-Pascal* programs and *Calculator* programs, or " CXE " for calculator programs developed under Windows 3.11. The selected code file must contain the kind of program code expected by the selected channel. The code type is checked prior to the program being downloaded.

A code file name can be inserted in the [Code file] combo box using three methods. The code file name can be included as a start up parameter for Program Download. Alternatively, it can be selected from the Open file dialog, by clicking the [FILE] button, or it can be directly keyed in into the combo box.

When a new file name is inserted, it will always become the highlighted item in the [Code file] combo box list. The selected item will be inserted at the top of the list. If the selected file was not previously included in the list and the list is full, the oldest file name will be deleted. Once a file name is included in the combo box, it can be easily selected from the list, which can hold up to 6 file names.

6.3.3 Autostart after reset

The [Autostart after reset] check box, defines how the selected program will behave, following a reset being applied to the node holding the program. If [Autostart] is checked, the selected program will perform an auto start after a reset. If it is not, the selected program will be put in the Idle state after a reset. The check box reflects the state of *ChConfig.EnableBit[0]* in the *Program Channel*.

Some node types can hold several programs within a library. With PD controllers, these programs can be stored in different memory types. The list box adjacent to [Autostart after reset], defines which program will be started, if [Autostart after reset] is checked.

The list box reflects the value of *ChConfig.Ref_A* in the *Program Channel*. The state of the [Autostart] check box and list box can only be changed, if [Write enable] is checked.

6.3.4 Selected library

The selection in the [Library] list box defines, to which library domain the program is to be downloaded. The list shows the possible choices for the selected channel. The possible values are read from the selected channel, in the variable called *MemoryInfo*.

The [Name] and [State] fields in the [Library] panel show the name and state of the selected program in the library. The library list can also be used to monitor the names and states of other programs in the library. *Library State* can take the following values:

0: Non-existent
1: Loading
2: Ready
3: In-use
4: Complete
5: Incomplete
14: Deleting

The [Selected library] list box reflects the value of *LibraryStatus.LibraryIndex* in the *Program Channel*. When a new value is selected, it is stored in *LibraryControl.LibraryIndex* in the *Program Channel*.

6.3.5 Selected program

The [Selected program] list box is used to select a program. The list shows the possible values for the selected channel. The possible values are read in the selected channel, in the variable called *MemoryInfo*.

If a program is already running when selecting a new program, the running program will be stopped and killed, and the new program will be selected, which will be put into the Idle state.

After selecting a program, the program can be started by pressing the <Start> button.

The [Name] and [State] fields in the Program panel show the name and state of the selected program. *Program state* can take the following values:

0: Non-selected
1: Unrunable
2: Idle
3: Running
4: Stopped
5: Starting
6: Stopping
7: Resuming
8: Resetting

The [Selected program] list box reflects the value of *ProgramStatus.SelectedProgram* in the *Program Channel*. When a new value is selected, it is stored in *ProgramControl.ProgramToSelect*.

6.3.6 Download button

Before downloading, a channel must be specified, a code file and a library must be selected, and [Write enable] must be checked. Clicking the <Download> button starts the downloading procedure, for the code file selected in the [Code file] combo box, to the channel and library defined in the [Channel] combo box and the [Selected library] list box.

If the download parameters specify a memory area that is already in use, by being in a state of e.g. running or selected, a message box showing “Selected library in use ! Continue ?” will appear. If <Yes> is selected, the program that is currently running or selected, will be stopped and killed, and the new program will be downloaded.

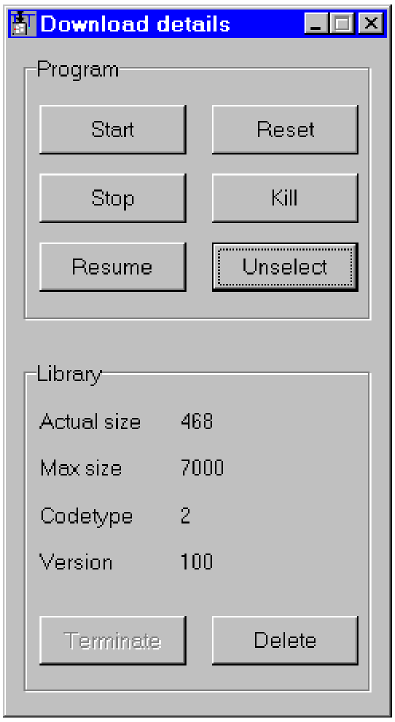
6.3.7 Start button

Clicking the <Start> button will start the *Selected program*. Clicking the [Start] button only has an effect, if a program is selected, and the program is in the *Idle* state.

6.3.8 Write enable

The [Write enable] check box enables the values in *ChConfig* of the selected channel to be changed, using the values available in the [Autostart after reset in] list box. [Write enable] must also be checked to download programs.
If [Write enable] is not checked, [Autostart after reset] and [Download] are disabled, and greyed out.

6.3.9 Details



Pressing the [Details] button opens a window with more detailed information about the selected channel. The [Program] field in the *Download details* window provides buttons to [Start], [Stop], [Resume], [Reset], [Kill] and [Unselect] the program defined in the [Selected program] list box.

The [Library] field in the *Download details* window, shows [Actual size], [Max size], [Code type] and [Version] of the program defined in the [Selected library] list box. The [Library] field also includes a <Terminate> button, which will terminate downloading to the selected library, and a <Delete> button, which will delete the program in the selected library.

6.3.10 Starting the Download Utility from a shortcut

Program Download is normally started from within the *MIBOCX*, via the right mouse button menu, when a *Program Channel* of object type 11 is selected. If this method is used, the identity of the selected *Program Channel* will be automatically inserted into the <Channel> combo box.
Program Download may also be started up using a previously prepared shortcut. Using this method, it is also possible to include two parameters. The first parameter is the *PhysId* of the channel to which the program code is to be downloaded. The value of this parameter will then be automatically inserted in the [Channel] combo box. The second parameter is the name of the code file to be downloaded. The value of this parameter will be automatically inserted in the [Code file] combo box. The parameters can be included by selecting *Properties* of the shortcut icon and appending them to the command line.

When the *Program Download* tool is closed, the contents of the [Channel] and [Code file] combo boxes are saved in a file called {PD}PROGRAMDOWNLOAD.CFG. This file is placed in the current folder, which would typically be the VIGO folder. When *Program Download* is started again, the contents of the 2 combo boxes will be restored, if no start up parameters has been given.

6.3.11 PD 5000 Controller

A *PD 5000* controller has two *Program Channels*, *OPSYSCH* and *PPPROGCH*.

OPSYSCH holds the controllers' operating system, and *PPPROGCH* holds a *Process-Pascal* program.

These two programs are inter-dependent. If the *Process-Pascal* program is running, a new operating system cannot be downloaded.

If a *Process-Pascal* program is present in the Flash library, a new operating system cannot be downloaded to Flash, because the operating system is located at the beginning of the Flash memory area, and the *Process-Pascal* program is loaded immediately after it.

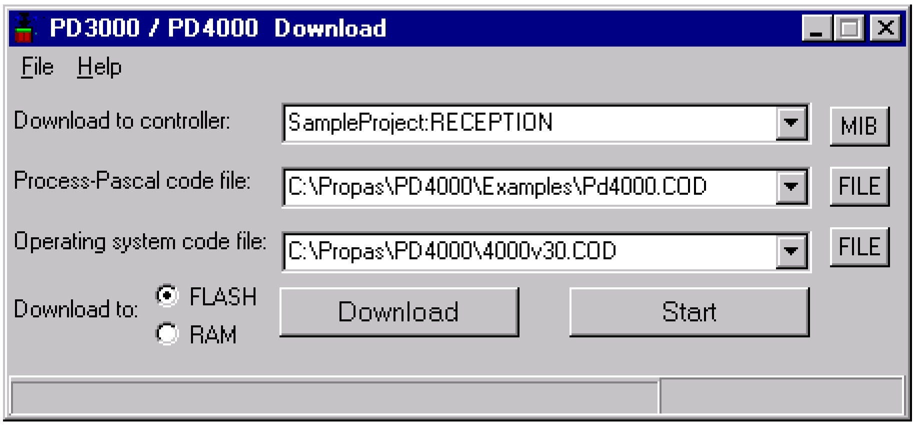
The situations described above are just two of the aspects, which the *Program Download* utility automatically monitors, to ensure download integrity. Should any other prohibited situations occur, the utility would provide any necessary warnings, before a program is automatically stopped or deleted.

7 Tools for PROCES-DATA modules

The tools described in this chapter relate only to modules manufactured by PROCES-DATA A/S.

7.1 PD 3000 / PD 4000 Download

The purpose of this program is to download *Process-Pascal* code to modules not originally designed to support the P-NET standard *Program Channel*. This applies to the *PD 3000* and *PD 4000* series of controllers with older operating systems. The utility can be called from the *MIBOCX*, using the right mouse button menu, when the appropriate node is selected.



The program is downloaded to the controller specified by the identifier inserted in the **[**Download to controller] edit field. After a program has been downloaded, it can be started by clicking the <Start> button.

If the selected node is a *PD 4000* controller, and it is required to store the code in flash memory, the *Process-Pascal* program must be downloaded together with an operating system. The name of the file holding the operating system is inserted in the [Operating system code file] edit field.

When a controller, a *Process-Pascal* code file, and possibly an operating system code file, have been specified, the program can be downloaded, by clicking the <Download> button.

7.1.1 Download to controller

The [Download to controller] edit field is used to insert the name of the controller to which a program is to be downloaded. The selected controller must be of type *PD 3000* or *PD 4000*.

A controller identifier can be inserted using four alternative methods. When this program has been launched via the *MIBOCX*, the controller identifier is automatically inserted. The controller name can also be included as a start up parameter. It can be inserted from the *MIB*, by clicking the <MIB> button and then double clicking on a controller name within the *MIBOCX*, or it can be directly keyed in into the edit field.

The integrity of the controller identifier is NOT checked, until the <Download> button or the <Start> button is clicked. When either of these actions occurs, a test will establish whether the controller is of the correct type and version. The version must be 2.00 or later.

7.1.2 Process-Pascal code file

The [Process-Pascal code file] edit field is used to identify the file to be downloaded, which contains the *Process-Pascal* program code. The selected file must be a *Process-Pascal* ".COD" file, generated by the *Process-Pascal* compiler version 2.00 or later.

A code file name can be inserted using three alternative methods. The code file name can be included as a start up parameter when launching the *PD 3000 / PD 4000 Download* program. It can be selected from an Open file dialog, by clicking the <FILE> button and then opening the file, or it can be directly keyed in into the edit field. The integrity of the name, type and version of the selected file is not checked, until the <Download> button is clicked.

7.1.3 Operating system code file

The [Operating system code file] edit field is used to specify the file to be downloaded, which contains the Operating system. This file is only required when a *Process-Pascal* program is to be downloaded to FLASH memory, in a *PD 4000* controller.

The selected file must contain the *PD 4000* operating system, as a ".COD" file, version 2.00 or later, (NOT a ".HEX " or ".EP0" file), as provided by PROCES-DATA A/S. For example, the file might be called "4000v30.COD".

A operating system code file name can be inserted using three alternative methods. The code file name can be included as a start up parameter when launching the *PD 3000 / PD 4000 Download* program. It can be selected from an Open file dialog, by clicking the <FILE> button and then opening the file, or it can be directly keyed in into the edit field. The integrity of the name, type and version of the selected file is not checked, until the <Download> button is clicked.

7.1.4 Download to

The <Download to> radio buttons define to which memory type the program is to be downloaded. If the controller is a *PD 3000*, the program can only be downloaded to “RAM”. If the controller is a *PD 4000*, the program can be downloaded to either “RAM” or “FLASH”.

If <FLASH> is selected, the name of a file containing the *PD 4000* operating system must be inserted in the [Operating system code file] edit field, before clicking the <Download> button. The correct selection of <RAM> or <FLASH> options is not checked, until the <Download> button is clicked.

Clicking the <Download> button commences the process of downloading the code file(s) specified in the code file edit field(s), to the selected controller.
Before a program is actually downloaded, the controller is stopped and reset.

If <FLASH> is selected, the “FLASH” memory is first cleared, which takes about 20 seconds. The operating system is then downloaded. Finally, the *Process-Pascal* program is downloaded.
Once a program has been downloaded, it can be started by clicking the <Start> button.

The state of the <Download to> radio buttons should not be changed during the period between download and start.

7.1.5 Starting PD 3000 / PD 4000 Download

When the *PD 3000 / PD 4000 Download* utility has been started from the *MIBOCX*, the contents of the [Download to controller] edit field are automatically inserted.

PD 3000 / PD 4000 Download may also be started by setting up a shortcut. In this situation, it is also possible to transfer 4 command line parameters, by amending the shortcuts properties. The first parameter is the controller identity, to which the program is to be downloaded. The value of this parameter will be inserted in the [Download to controller] edit field. The second parameter is the name of the *Process-Pascal* code file, and this will be inserted in the [Process-Pascal code file] edit field. The third parameter is the name of the Operating system code file, which will be inserted in the [Operating system code file] edit field. The fourth parameter defines the state of the <Download to> radio buttons. If the value of this parameter is “RAM”, the <RAM> button will be checked, otherwise the <FLASH> button will be checked.

When the *PD 3000 / PD 4000 Download* program is closed down, the contents of the [Download to Controller], [Process-Pascal code file] and **[**Operating system code file] edit fields, and the state of the <Download to> radio buttons, are all saved in a file called {PD}PDDOWNLOAD.CFG. This file would normally be located in the current folder, e.g. VIGO. When the *PD 3000 / PD 4000 Download* program is started again, these values are restored, if no start up parameters have been specified.

7.2 Calculator Assembler

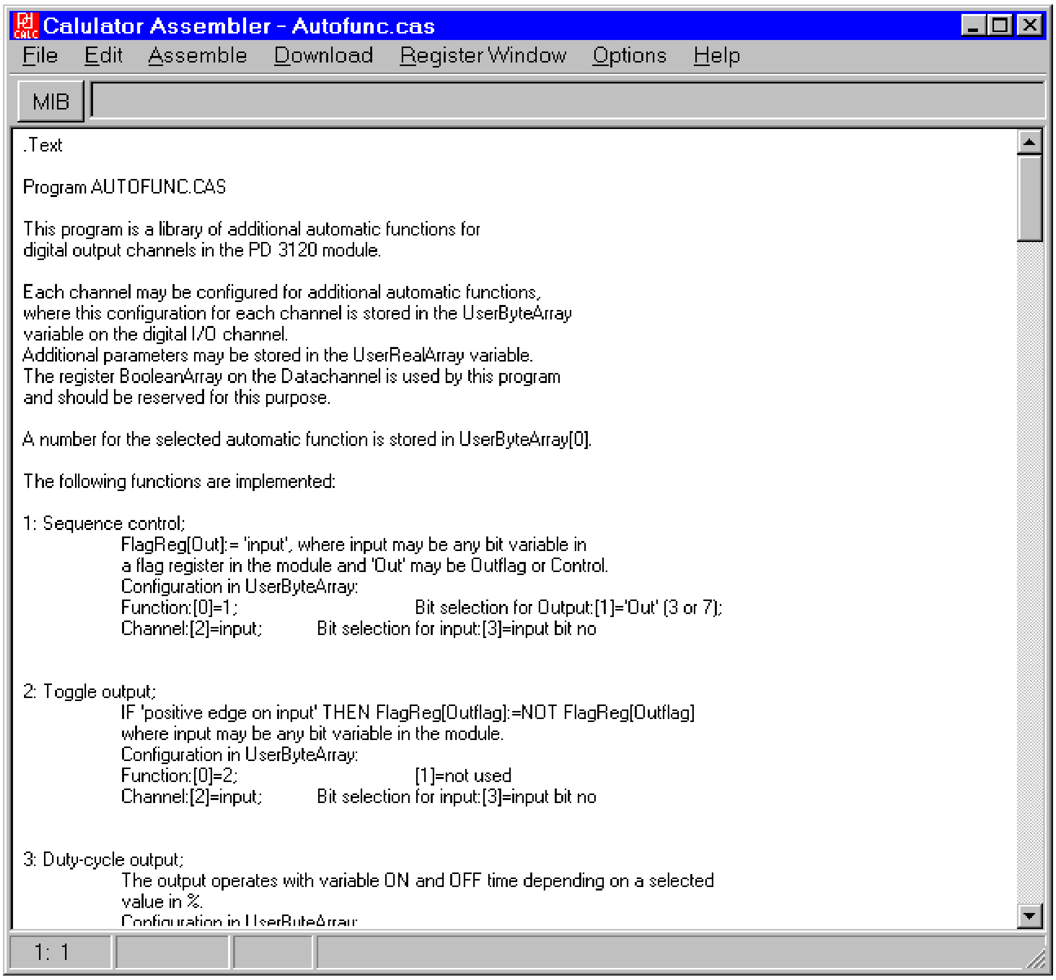
The *Calculator Assembler* provides all the necessary services for a programmer to design, edit, assemble and download programs to calculator channels in P-NET modules, from a PC. The *Calculator Assembler* is an integrated program with an editor, an assembler, a debugger and a loader for P-NET. Calculator programs are written as assembler instructions in text files. By using the editor, the source text is edited and saved. By using the assembler, the source text is assembled to generate the calculator instructions. These instructions are downloaded to the *Calculator channel*, and the program can then be started. It is possible to debug the downloaded program by single steps or a break point.

An example of a *Calculator Program* with some typical instructions.

```
         Move #D,CR1              ; Let CR1 point out the pulse
                                  ; processor channel in a PD 3221
Start:   Move #0,IR1              ; First element
Loop:    Move CR1:#A[IR1], Acc    ;Load pulse processor
                                  ;registers[elementNo]
         Add 100
         Move Acc, CR1:#A[IR1]    ; Store back incremented value
         Inc IR1                  ; Next element
         Move IR1, Acc            ; Load IR1 into Acc
         Comp Acc > 15            ; Is last element treated?
         Jump.False Loop          ; No: then repeat loop
Finish:  .                        ; Yes: then ....
         .
         .
         Jump Start               ; Last instruction must be an
                                  ;unconditional jump to a label
End                               ; End of program
```

7.2.1 User Interface

The program's main window contains a menu bar at the top and a status line at the bottom. A file can be opened for editing and assembling. Furthermore, the program has a <MIB> button used to select the destination module for download and debugging.



In the following paragraphs, the different parts of the program are covered in detail separately.

7.2.2 Editing a file

The editor is used to edit a calculator program. Files can be opened by the 'File | Open...' command in the menu. Files are also saved and printed from the File menu. Calculator assembler source files have “CAS” as default extension.

A new (blank) file window corresponding to a new assembler file can be created with the 'File | New' command.

The standard edit commands (listed in the Edit menu) for copy, cut and paste of a selected text are available in the editor. Text can be exchanged to and from the Windows clipboard.

The editor can undo the last cut command. Marked text will be replaced when new text is entered. The search and replace operations are listed in the Search Menu.

In the status line the cursor's position is shown. It is also indicated if the file has been modified since the last time it was saved.

Note that when the register window (used when debugging) is shown, the content of the edit window is locked. If changes have to be made, close the register window, make the changes, assemble the program and download it again. Then reopen the register window to see the effect of the changes.

7.2.3 Assembling a program

The assembling of a source file is started from the Assemble Menu. A status dialogue displays the line number and size of the generated code during the assembling.

When an error is found, a dialogue will pop up to inform the user about the error. After pressing the OK button in the status dialogue, the assembling process continues. Pressing the cancel button in the status dialogue can interrupt the assembling process.

During the assembling process a debug (*.deb) file is generated. It contains a list of line numbers, instruction addresses and label names. The list is used for debugging.
After a successful assembling (No errors found), it is possible to download the generated code to a calculator channel. The download command in the main menu is the means to do that.

Using the Assemble menu, generated code can be saved in a file. The 'Write to INC file' command in the 'Assemble' menu, will create an include file for *Process-Pascal*. The 'Save to COD file' command will save to a file on disk, ready to be downloaded to a calculator channel.

*.inc	The generated calculator instructions as *Process-Pascal* source text. May be included in a Process Pascal program (ASCII).
*.cod	The generated calculator instructions in a binary format.

7.2.4 Downloading a program

For downloading of calculator programs, a *Calculator Channel* must be selected as destination. A *Calculator Channel* is a channel with a physical ID ending with '"CALCULATOR".
To select a calculator channel, activate the <MIB> button at the top left of the screen. This changes the edit window into a *MIB Viewer*. Use the mouse to expand the *MIB* structure to find the desired *Calculator Channel*. When the calculator channel is highlighted, click on the <MIB> button again or double click the channel. This will close the *MIB Viewer* and the edit window reappears. The selected channel is now shown to the right of the <MIB> button.

When the desired destination channel is shown, select the <Download> menu. This will start the standard P-NET downloader application. For further information on the download program, refer to the documentation for the downloader.

7.2.5 Debugging a program

The *Calculator Assembler* supports interactive debugging of a *Calculator Program* downloaded to a *Calculator Channel*. The debugger makes it possible to single step through the calculator instructions or to set a break point.

After an assembling of a source text, the generated codes must be downloaded to a *Calculator Channel*, and the module must be reset to reinitialise the calculator. The debugger is started by selecting <Register Window> in the main menu. When debugging starts, the register window will show the calculator's registers. It is not possible to edit the program source when the register window is shown.

The user can single step through the program, by pressing a key <F7>. To set a break point, the user places the cursor on the line containing the instruction at which a stop is required, and then press the <F4> key. The *Calculator* then starts, and runs until the break point is reached. When the execution stops the *Calculator's* internal registers can be inspected in the register window. The register values can also be changed. To restart the *Calculator Program* from the first instruction, the <F2> key can be pressed.

The register window also contains three buttons, which acts as shortcuts to the [Debug Step], [Debug Goto cursor] and [Debug Reset] commands. The [Debug Reset] command stops the calculator if it is running, and resets the calculator's instruction pointer to the first program instruction. When the debugger is reset, has been single stepped or has reached a breakpoint, the line containing the next instruction to be executed, is highlighted as marked text.

7.2.6 Calculator programming

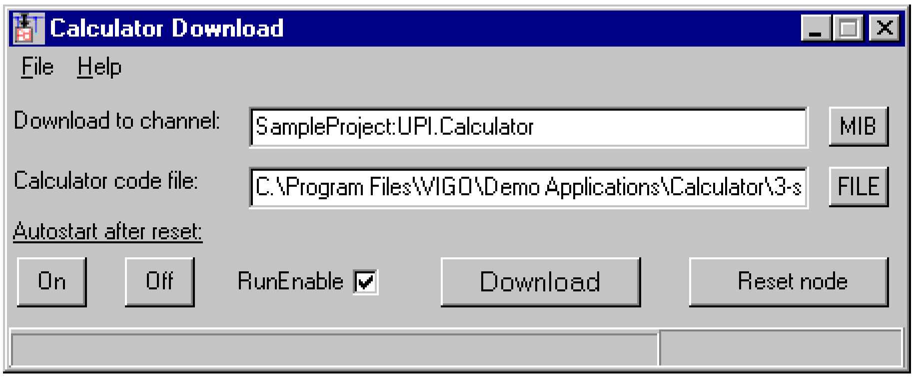
Details about the *Calculator Programming*, the calculator registers and the instruction set are found in the Calculator Programming Manual "*PD Calculator Assembler, ref. no. 502 061*".

7.2.7 Help

Online information about this program, is available by using the Help menu.

7.3 Calculator Download

The purpose of this program is to download *Calculator code* to modules **not** supporting the standard P-NET *Program Channel*. It is used to download calculator program code to the *Calculator Channels* included in the *PD 3221 UPI* and *PD 3230 Weight* modules produced by PROCES-DATA A/S. It may be started from within the *Calculator Assembler*, or it can be launched from the *MIBOCX* using the right mouse button menu, when a *Calculator Channel* has been selected.
When a program has been downloaded, it can be started, by checking the <RunEnable> option.



If this utility program is called using the right mouse button menu, the identity of the Channel to which the download is to be made, is automatically provided. The Channel identity can be changed, by pressing the <MIB> button, and then selecting a new *Calculator Channel*. A File browser can be opened for selecting a Code file, by pressing the <FILE> button. The node containing the *Calculator Channel* must be defined in the *MIB*, before any Download procedure can be started.

The program is downloaded to the *Calculator Channel*, specified by the identifier inserted in the [Download to channel] edit field. The name of the file holding the calculator code, is specified in the [Calculator code file] edit field.

Before downloading, a channel must be specified, and a code file selected. The *Calculator Download* program automatically controls the state of the *WriteEnable* flag in the Node. Following a download, the *WriteEnable* state is set back to the value it had, before the start of the download.

7.3.1 Download to channel

The [Download to channel] edit field is used to insert the identifier of the channel, to which a *Calculator Program* is to be downloaded. The selected channel must be a *Calculator Channel* in a *PD 3221 UPI* or *PD 3230 Weight* node. These channels have an object type of 7.

A channel name can be inserted using four alternative methods. If this program is launched using the right mouse button menu, the identity of the Channel is automatically inserted. The channel name can also be included as a start up parameter. It can be selected from the *MIB*, by clicking the <MIB> button and then double clicking on a channel name within the *MIBOCX*, or it can be directly keyed into the edit field.

7.3.2 Calculator code file

The [Calculator code file] edit field is used to select the file containing the *Calculator Program* to be downloaded. The selected file must be a calculator file, having an extension of ".COD", which has been generated by the *Calculator Assembler* produced by PROCES-DATA A/S. However, *Calculator Programs* developed under Windows 3.11 may use the extension ".CXE".

A code file name can be inserted using three alternative methods. The code file name can be included as a start up parameter. It can be selected from an <Open file> dialog, by clicking the <FILE> button and opening the required file, or it can be directly keyed into the edit field.

Once a program has been downloaded, it can be started by checking [RunEnable].

7.3.3 RunEnable

The [RunEnable] check box reflects the state of the boolean variable *RunEnable*, in the selected *Calculator Channel*. Whenever *RunEnable* is true, indicated by a tick in the check box, the *Calculator Program* will be running. When the downloading of the program is complete, *RunEnable* is automatically set to false, so that the calculator program doesn't start until [RunEnable] is checked.

The *RunEnable* variable is stored in a memory type called *RAMInitEEPROM*. This means, that there are in fact two variables, one in “RAM” and one in “EEPROM”. It is the state of *RunEnable* in “RAM” that determines, whether the *Calculator Program* is running or not. It is the state of *RunEnable* in “RAM” that is mirrored by the [RunEnable] check box.

After a reset of a *UPI* or *Weight* node, the state of *RunEnable* is copied from EEPROM to RAM. If the state is “TRUE”, the *Calculator Program* will automatically start running.

The state of *RunEnable* in EEPROM can be set “True”, by clicking the <ON> button under [Autostart after reset]. If *RunEnable* in RAM was “False”, it will be set “True” for a short period, during this operation.

The state of RunEnable in EEPROM can be set “False” by clicking the <OFF> button under [Autostart after reset]. If *RunEnable* in RAM was “True”, it will be set “False” for a short period, during this operation.

7.3.4 Download

Clicking the <Download> button will begin the procedure of downloading the code file, specified in the [Calculator code file] edit field, to the selected channel.

Before the program is downloaded, the *Calculator* is stopped, by setting [RunEnable] to “False”. *RunEnable* remains “False”, so that the program doesn't start automatically after downloading has been completed.

7.3.5 Reset node.

Clicking the <Reset node> button will reset the *UPI* or *Weight* node. The button is provided as a convenient way to ensure that the calculator behaves, as it should, following a reset. That is, whether it autostarts or not.

7.3.6 Starting the Calculator Download program

Calculator Download may be started from within the *Calculator Assembler*, or it may be launched from within the *MIBOCX*, via the right mouse button menu, when the *Calculator Channel* in a *PD 3221 UPI* or a *PD 3230 Weight* module has been selected. These *Calculator Channels* are of object type 7.

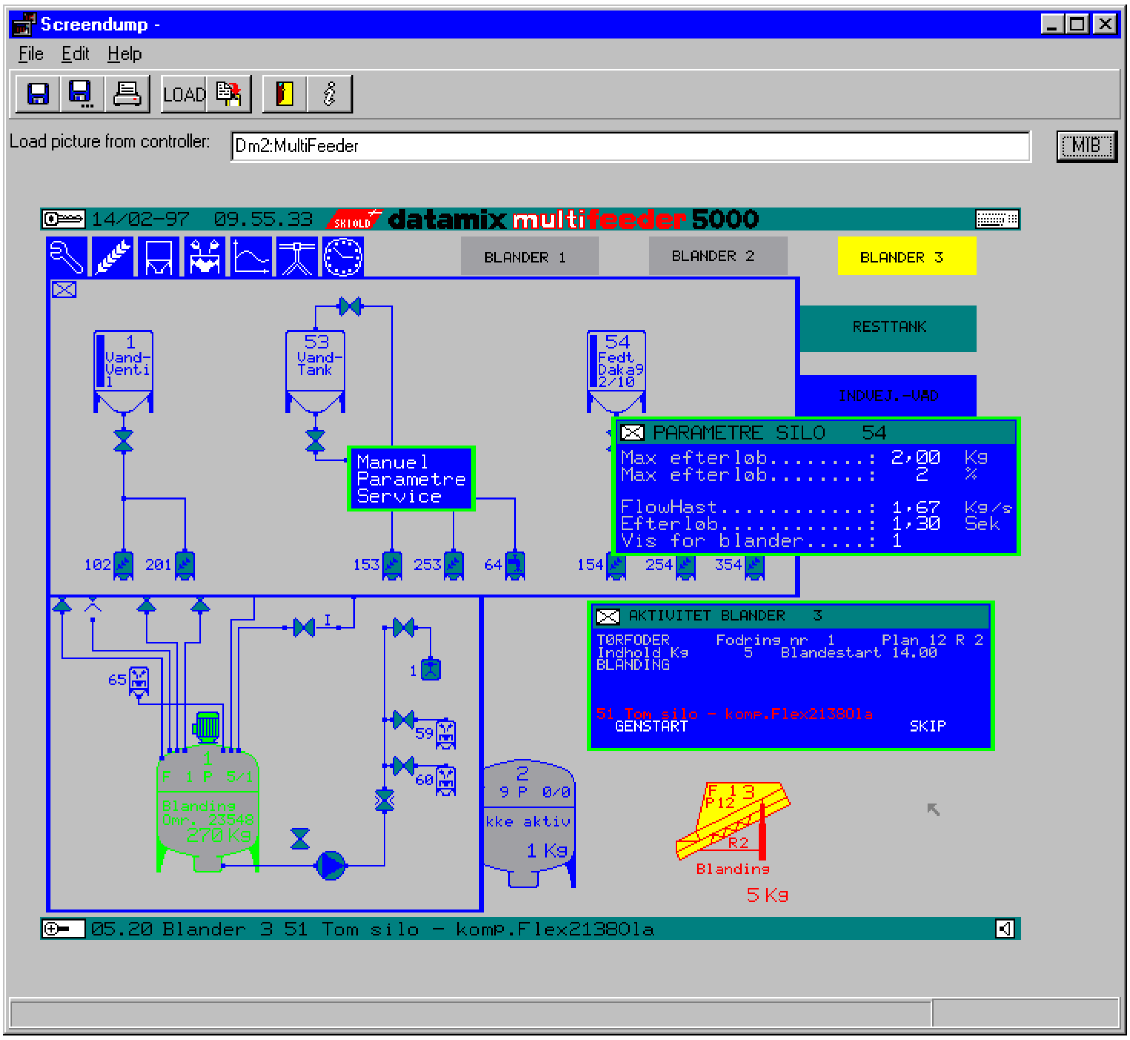
Calculator Download may also be started using a previously set up shortcut. In this case, it is possible to include two command line parameters, by amending the shortcut properties. The first parameter is the identity of channel, to which the program is to be downloaded. The value of this parameter will be shown in the [Download to channel] edit field. The second parameter is the name of the *Calculator* code file, and will be shown in the [Calculator code file] edit field.
If *Calculator Download* is started from within the *Calculator Assembler*, these two parameters are automatically transferred.

When the *Calculator Download* program is closed down, the contents of the Channel and code file edit fields are saved in a file called {PD}CALCULATORDOWNLOAD.CFG. This file is placed in the current folder, which is typically the VIGO folder. When *Calculator Download* is started up again, these values are restored, if no new parameters have been specified.

7.4 Screen Dump

The *Screen Dump* utility is used to up load screens displayed on P-NET controllers, manufactured by PROCES-DATA A/S. Once a picture has been up loaded, it can be printed out, saved in a file, or transferred to the clipboard. The program is useful when creating documentation applicable to *Process-Pascal* application programs.

Screen Dump is able to up load display screens from *PD 3010, PD 4000, PD 5010, PD 5015* and *PD 5020* controllers. However, to obtain a picture from a *PD 5020*, which has a larger screen, a special task and set of variables (VGALOAD) must be incorporated in the *Process-Pascal* program in the controller.



Screen Dump can be started from the *MIBOCX*, using the right mouse button menu, when a node of the following controller type is selected: *PD 3010, PD 4000 or PD 5010, PD 5015* or *PD 5020*. These form one of the Object Types - 3000, 4000 or 5000.

If this procedure is used, the Controller Identifier will be automatically transferred to the [Load picture from controller] edit field.

Screen Dump may also be started from a shortcut, in which case the controller identifier may be transferred as a parameter.

When the identifier of a controller has been specified, the display screen can be up loaded by pressing the <LOAD> button. After a picture has been loaded, it can be stored in a Windows bitmap file ".BMP", by means of the <Save> or <Save as> menu items.

The picture can also be printed by means of the <Print> command in the <File> menu. If required, the picture can be copied to the clipboard, and then imported into other programs, such as Paintbrush, WordPerfect or Word.

The size and location of the *Screen Dump* window on the PC screen may be changed. When the *Screen Dump* program is closed the actual size and location of the window is saved, along with the controller identifier, and the name of the file last used. These values are restored the next time *Screen Dump* is started. The controller identifier is not restored, if *Screen Dump* is launched from within the *MIBOCX* or with a parameter.

7.4.1 Save / Save as

The *Save / Save as* dialogues can be called by clicking the <Save> or <Save as> toolbar buttons, or from the <File> menu, or by pressing <Ctrl> + <S> on the keyboard.
If a file name has not yet been defined, this can be done using the file dialogue. Once a file name has been chosen, it will be shown in the header of the *Screen Dump* window.
The file is saved as a Windows bitmap file, with the default extension ".BMP".

7.4.2 Print

A loaded screen can be directed to print from within the *Screen Dump* program, by means of the *Print* dialogue. This dialogue is called by clicking the <Print> button, or from within the <File> menu, or by pressing <Ctrl> + <P> on the keyboard.

7.4.3 Load

Once a controller identifier has been inserted into the controller edit field in the program window, the current controller screen image can be up loaded by clicking the <LOAD> button.

Loading pictures from a *PD 3010, PD 4000, PD 5010* or *PD 5015* only takes a few seconds. However, loading pictures from a *PD 5020* can take up to several minutes. If the loading of a picture from a *PD 5020* is cancelled (by pressing the <Cancel> button), a new picture load must not be initiated during the next 30 seconds.

7.4.4 Copy to clipboard

Once a picture is loaded, it can be transferred to the standard Windows clipboard. From here, it can be imported into other Windows programs. The loaded picture is copied to the clipboard by pressing the <Copy to clipboard> button, or from the <Edit> menu, or by pressing <Ctrl> + <C> on the keyboard.

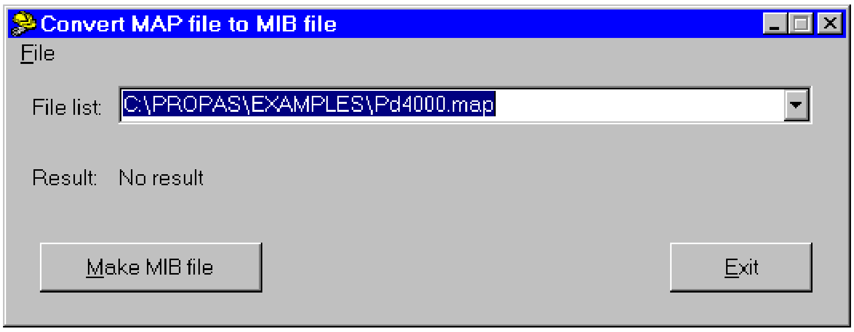
7.4.5 Load picture from controller

The [Load picture from controller] edit field is used to insert the identifier of the controller holding the screen image to be up loaded.

The selected controller must be of type *PD 3010, PD 4000, PD 5010, PD 5015* or *PD 5020*. The Controller Identifier can be inserted using four alternative methods. If this program is launched using the right mouse button menu, the identity of the controller is automatically inserted. It can also be inserted using the *MIB*, by clicking the <MIB> button, and then double clicking on the required Controller Identifier within the *MIBOCX*. The Controller Identifier can also be inserted during *Screen Dump* start up, by using a saved parameter, or it can be directly keyed in into the edit field.

7.5 MapToMIB

The *MapToMIB* program is a conversion utility used to convert *MAP* files to *SMB* files. This utility is only required for programs compiled by *Process-Pascal* compiler versions prior to 4.00.
A *MAP* file is generated by older versions of the *Process-Pascal* compiler, and contains an ASCII text description of the variable names and their types, as declared in a *Process-Pascal* program.



The *SMB* file (produced by the *MapToMIB* program), is a binary representation of the same information, but in a format that can be read by the *MIB Edit* program. *SMB* is short for SubMIB.

A *SMB* file is used to update or create a new Node type in the *MIB* database.

Using the *MapToMIB* program

The following section describes the functionality of the menus and buttons in the program window. For a detailed description of how to create or update a Type in the *MIB* database, refer to 'Step-by-step Instructions'

[File List]
The [File List] contains a list of *MAP* files for selection. These are the *MAP* files that are to be converted by the *MapToMIB* program. To add or remove files, use the <File | Open> and <File | Clear File List> menu commands.

The program can either convert one file or all the files in the [File List] (<File | Make> or <File | Make All>). Clicking the down arrow to the left of the [File List], and then clicking the filename can select a single file.

[Result]
This field contains a message indicating the result of a conversion, e.g. an error message.

<Make SMB File>
Pressing this button will convert the selected *MAP* file into a *SMB* file. This is the same as selecting <File | Make> in the menu.

<Exit>
Pressing this button will terminate the program.
<File | Open>
The <File | Open> command will show an open-dialog box. The selected file will be added to the [File List].

<File | Clear File List>
This command clears the [File List].

<File | Open>
The <File | Open> command will show an open-dialog box. The selected file will be added to the [File List].

<File | Clear File List>
This command clears the [File List].

<File | Make>
The <File | Make> command will convert the selected *MAP* file in the [File List] into a *SMB* file, which will be placed in the same folder as the *MAP* file. This is the same as clicking the <Make SMB File> button.

<File | Make All>
All the *MAP* files in the [File List] will be converted to *SMB* files, and will be placed in the same folder as the *MAP* files

<File | Exit>
This command terminates the program.

<File | About>
Selecting this command will show an About box, stating the program name and the current version.

Step-by-step Instructions
The following procedures should be used to update or create a node Type in the *MIB* data-base, based on the variables and types declared in a *Process-Pascal* program.

1. Generate a *MAP* file with the *Process-Pascal* compiler.

2. Start the *MapToMIB* program. Add the *MAP* file to the [File List] by using the <File | Open> command. Ensure that the *MAP* file is selected in the [File List]. If it is not, select it by clicking the down arrow in the [File List] and then clicking the filename.

Note, that the contents of the [File List] are preserved between each session of the *MapToMIB* program, so once a file has been added, it will remain there until <File | Clear File List> is selected.

3. Convert the *MAP* file into a *SMB* file by clicking the <Make SMB File> button. If no errors occur, terminate the *MapToMIB* program by clicking the <Exit> button.

4. Select the [MIB Edit] tab and select <View | Show Types> in the menu. This should reveal all the Types (red icons) currently defined in the *MIB* database.

5. If it is a new Type, it should be created as described in this section. If it is an existing type that is to be updated, the following steps (5.a to 5.e) should be skipped.

   a. Right click the project icon (e.g. *SampleProject*) and select New from the menu.

   b. Ensure that <Add New as | Sub Element> is selected.

   c. Select “Node Type” as [New Kind].

   d. Key in a Type name for the new node in the [New Name] field.

   e. Press the <OK> button.

6. Right click the new type and select <Update Type> from the pop up menu. This will show an *Open File* Dialog box. Select the *SMB* file created by the *MapToMIB* program.

   The type is now updated / created

7. If the node is a PD 3000 or a *PD 4000* Controller, the following steps (7.1a to 7.1d) should be performed:

   1a. Right click the new node type and select <Properties> from the pop up menu.
   The *Properties* window will now open.

   1b. In the *Properties* window select the [Type Info] tab and enter the following values:

Capabilities:	130
Object Type	3000 or 4000

   1c. Close the *Properties* window.

   1d. Select <View | Show Nodes/Virtual Names> in the [MIB Edit] window.

If the node is a *PD 5000* Controller, the following steps (7.2a to 7.2f) should be performed:

2a. Right click the new node type and select <Properties> from the pop up menu.
The *Properties* window will now open.

2b. In the *Properties* window select the [Type Info] tab and enter the following values:

Capabilities:	32
Object Type	5000

2c. Expand the node-tree in the [MIB Edit] tab window by clicking the <+> sign to the left of the new type name.

Use the scroll-bar on the right-hand side of the *MIB Edit* window, and scroll down to find the following channel types:

Channel Name	Object Type
Service	1
LedCh	2
AlarmCh	2
OpSysCh	11
PPProcCh	11

The Object type of each channel name should be changed in accordance with the above table. To do this, perform the following procedure (2d) for each channel name.

2d. Right click the channel name in the *MIB Edit* window, and select <Properties> from the pop up menu. In the *Properties* window, note the Type under [Element info]. This is the *Typename* of the channel, e.g. 'TypeNo117'.

Use the scrollbar on the right-hand side of the *MIB Edit* window to find the [Typename] of the channel. Left click the [Typename] to update the *Properties* window with the type information of the channel typename.

Select [Type Info] in the *Properties* window and change the [Object Type] to the value in the above table.

Repeat this for each channel name in the table.

2e. Close the *Properties* window.

2f. Select <View | Show Nodes/Virtual Names> in the *MIB Edit* window.

8. An instance of this newly created node type can now be incorporated within the project. If it is a new Node, it should be created as described in this section. If the node of this node type already exists, the node type will have already been updated, and this step can be skipped.

To create a new Node, right click the project icon in the *MIB Edit* window and select <New> in the pop up menu. In the [Add Element] dialog box, make sure that [Add new as: Sub Element] is selected. Select “Node” in the [New Kind] combo box. Type a name for the node in the [New Name] edit box. Now click the <OK> button.

Right click the new node icon and select <Properties> from the pop up menu. Under the [Element Info] tab in the *Properties* window, select the new node type name in the [Type] combo box. Select the network to which the Node is connected in the [Net] combo box.

Close the *Properties* window.

9. Right click the [WorkSpace] icon in the *MIB Edit* window, and select <Refresh> from the pop up menu.

The new node is now ready to be used.

8 Error codes

This section describes the relationship between the Errorcode and ErrorString properties of the VIGO.STD and VIGO.PRO objects. Different parts of the VIGO system can generate error codes and strings. This section is divided into a subsection for each error code generating part of VIGO.

8.1 P-NET errors

0x0100=Historical Data Error
0x0200=Actual Data Error
0x0300=Actual and Historical Data Error
0x0800=Conversion Error
0x0900=Historical and Conversion Error
0x0A00=Actual and Conversion Error
0x0B00=Actual, Historical and Conversion Error
0x0400=No Response
0x0408=Time Out
0x0410=Too Busy
0x0418=Wait Too Long
0x0420=Buffer Full Or Empty
0x0428=Data Format Error
0x0430=SWNo Error
0x0438=Node Address Error
0x0440=Read- or WriteProtected
0x0448=InfoLength Error
0x0450=Instruction Error
0x0480=Error Detect Failure
0x0488=Overrun-Framing Error
0x0490=Net Short Circuit
0x0498=Port Not Master
0x04A0=Out Of Sync
0x04A8=RS-232 Handshake error
0x04E8=No interrupt from P-NET card
0x04F0=P-NET card has been reset
0x04F8=Defect P-NET card
0xFFFE=No free controlcards
0xFFFD=No receiver program - VIGO version in server less than 4.00 ?
0xFFFB=Set Event failed
0xFFFA=Busy
0xFFF9=No MODEM connection
0xFFF8=IP/IPX error - No connection ?
0xFFF7=No legal address
0xFFF6=No P-NET address field
0xFFF5=No contact with P-NET card
0xFFF3=Internal Packet failure
0xFFF1=Device driver not loaded
0xFFF2=No connection or No legal address

0xFFEB=MODEM Responded NO CARRIER
0xFFEC=MODEM Responded NO DIALTONE
0xFFEE=MODEM Responded BUSY
0xFFEF=MODEM Responded ERROR
0xFFF0=MODEM Failed too disconnect phone line

8.2 HUGO errors

0xFF00=Out of memory
0xFF02=File not found
0xFF03=Path not found
0xFF05=Attempt to link dynamically
0xFF06=Library requires separate data segment
0xFF08=There was insufficient memory to start the application
0xFF0A=Incorrect Windows version
0xFF0B=Invalid executable file
0xFF0C=OS/2 application
0xFF0D=DOS application
0xFF0E=Unknown executable file type
0xFF0F=Attempt to load file in protected mode
0xFF10=Attempt to load a second instance
0xFF11=Attempt to load a file in large EMS mode
0xFF12=Attempt to load file in real mode
0xFF13=Attempt was made to load a compressed exe file
0xFF14=Dynamic-link library (DLL) file was invalid
0xFF15=Application requires Microsoft Windows 32-bit
0xFF16=Could not load driver
0xFF20=Response structure not decreased
0xFF21=Local free failed
0xFF22=String copy failed
0xFF23=No response number
0xFF24=Response no out of range
0xFF25=No response information added to Controlcard
0xFF26=No response type defined (Message, Callback or Polling)
0xFF27=Max length for extra data exceeded
0xFF30=Can't open license file
0xFF31=Illegal version of software
0xFF32=License no longer valid
0xFF40=MIB project file not found
0xFF41=No Default project loaded by VIGO.EXE
0xFF42=No valid handle to the Target Specification
0xFF43=The MIB project number out of range
0xFF44=Not able to load project into MIB
0xFF45=No default project found
0xFF80=Can not load Hugo2, because out of memory
0xFF82=Hugo2 not found

0xFF83=Path for Hugo2 not found
0xFF85=Hugo2 can not find driver
0xFF86=Hugo2 requires separate data segment
0xFF88=Insufficient memory (Hugo2)
0xFF8A=Incorrect Windows version for Hugo2
0xFF8B=Hugo2 invalid executable type
0xFF8C=Different operating system (Hugo2)
0xFF8D=DOS application
0xFF8E=Unknown executable type
0xFF8F=Hugo2 attempt to load in real mode
0xFF90=Attempt to load a second instance of Hugo2
0xFF93=Hugo2 attempt to load a compressed executable file
0xFF94=Hugo2 DLL files were invalid
0xFF95=Hugo2 requires Windows 32
0xFF96=Hugo2 just loaded
0xFF97=Hugo2 failure (Hugo2 unstable)
0xFF98=Controlcard not in timeout queue
0xFF99=No response, timeout
0xFFA0=Application buffer too small
0xFFB6=Controlcard contents can not be reused
0xFFB7=Postmessage failure (Response type = message)
0xFFB8=Error DLL file not found
0xFFB9=Error DLL already loaded
0xFFBA=No translate function found within Error DLL
0xFFBB=Wrong identifier type within Controlcard
0xFFBC=No error field within Controlcard
0xFFBD=Rewind not allowed, because no error detected
0xFFBE=Retry not allowed
0xFFBF=Not a Hugo2 address within Controlcard
0xFFC0=Illegal number of masters
0xFFC1=Illegal node address
0xFFC2=Could not load P-NET device driver
0xFFC3=Thread error in driver
0xFFC4=Illegal IO_CONTROL_FUNCTION
0xFFC5=Illegal parameter size
0xFFC6=Interrupt not available
0xFFC7=No contact to P-NET card
0xFFC8=Illegal LPT-Port or COM-Port number
0xFFC9=Illegal BASE-IO-ADDRESS
0xFFCA=Registration database key not found - is P-NET card installed?
0xFFCB=Illegal baudrate
0xFFCC=Illegal HUGO path
0xFFCD=Socket Error
0xFFCE=LPT-Port or COM-Port busy
0xFFCF=No Workspace created
0xFFD0=P-NET driver error - check IRQ and I/O in Driver parameters window
0xFFD1=IRQ error - Restart your computer as administrator and start VIGO

0xFFD6=No Net file found
0xFFD7=No driver loaded
0xFFD8=Project already loaded
0xFFD9=No project connected
0xFFDA=No current port configuration
0xFFDB=Initiate function failure
0xFFDC=Application not running
0xFFDD=Application has no access to application list
0xFFDE=No relation between window and task
0xFFDF=Max length for request data exceeded
0xFFE5=AED field do not fit total field length
0xFFE6=No Confirmation function found
0xFFE7=No Indication function found
0xFFE8=No Conclude function found
0xFFE9=No Initiate function found
0xFFEA=Max length for name exceeded
0xFFEB=Application number out of range
0xFFEC=Application number not equal to Requester
0xFFED=Driver not loaded
0xFFEE=Application number already in use
0xFFEF=Named file not found
0xFFF0=Response number out of range (Hugo2 unstable)
0xFFF1=No free response number
0xFFF2=Identifier type not allowed
0xFFF3=Application is not owner of Controlcard
0xFFF4=Response data too large
0xFFF5=Control number out of range
0xFFF6=Response fata not allowed (Unconfirmed request)
0xFFF7=Unconfirmed request not allowed
0xFFF8=Request not legal

0xFFF9=No free control number
0xFFFA=Control number is not in use
0xFFFB=No Response application
0xFFFC=No Request application
0xFFFD=No field within Controlcard
0xFFFE=Application has no access to control number
0xFFFF=Indication or confirmation handle invalid

8.3 MIB errors

0x0001=Error doing project allocating
0x0000=Allocation OK. First load
0x0002=Allocation OK. Not firstload
0x0003=Can't open MIB file, it is already opened in edit mode by:

// LoadProject
0x0020=Project Info Pointer not found
0x0021=No file name
0x0022=No file type
0x0023=Unknow file type

// UpdateBlockIndexes / UpdateListIndexes
0x0030=Wrong Element size
0x0031=Wrong Element size
0x0032=Variable without type declaration
0x0033=Wrong Element size

// ReadSubBlockFromFile
0x0040=Error reading header
0x0041=Error in version
0x0042=Error in block size
0x0043=Reserved header name
0x0044=Update with another name failed
0x0045=File block name read failed
0x0046=Access to file denied
0x0047=File not found
0x0048=Unknown file
0x0049=Load subblock as project
0x004A=Load project as subblock
0x004C=File block read error
0x004D=Error when setting Read-File-Position
0x004E=Error in the project number
0x0241=Error in version, Use VIGO 3.0 for conversion
0x0242=Error in version, can't be converted

// OpenMIBBlockFile
0x004B=Project script file not found

// WriteSubBlockToFile
0x0050=Write header failed
0x0051=Write database failed

// PutElementFields:
0x0053=Name already exists
0x0169=The Port is already defined, can not be redefined
0x0170=Editing the PortName is not allowed

// DeleteElement
0x0054=Move block to bottum failed

// MoveNameToBottom
0x0055=Error in the Name string

0x0056=Error in the Name string format

// FindLocalName
0x0057=Error in the Name string

// FindName
0x0058=No name specified
0x0059=Name not found
0x005A=Subelement not found
0x005B=Invalid ProjectPtr or NameSpace index error

// FindNameStr and FindNameStrType
0x0060=No name
0x0061=Field name not found in record type

0x0062=Type name not found
//0x0063=Name not found
0x0064=Element name not found
0x0065=Error in Array index
0x0066=Missing '['
0x0067=Error in Array index
0x0068=Name is a type
0x0069=Name to string format wrong
0x006A=Not an array type error
0x006C=Missing ']'

//FindPhysId
0x006B=TypeName In Reference

// AddElementWithSize
0x006D=No more nets allowed

// GetNameIndex:
0x0070=Name missing
0x0071=Type missing

// FillTargetSpecification:
0x0080=Project missing
0x0081=Name string missing
0x0082=Wrong version
0x0083=Invalid name space
0x0084=Consistency error, go to the MIB Edit Tab in VIGO.exe and run a Consistency Check on the project and save the project
0x0085=Project is not enabled
0x0086=Cannot set SubPhysId, PhysId is not SWNo, Register or Channel
0x0087=PhysID is a group

// SubFillTargetSpecification:
0x0090='.' expected
0x0091=Error in Type name
0x0092=Error in Index range
0x0093=Missing ']'
0x0094=Missing '['
0x0095=Invalid name
0x0096=Missing '[' or '.'
0x0097=Syntax error
0x0098=No subtype
0x0099=Unknown kind
0x009A=Invalid type name
0x009B=Invalid use of alias
0x0165=No access to internal values in BufferType
0x0168=Alias OverFlow

// MAP2MIB. MakeNodeTypeFile , ReadMapFile:
0x0100=MAP file not found
0x0101=MAP file could not be opened

// CheckConsistency
0x0109=Invalid basic type
0x0110=Name to long
0x0111=Unexpected type name
0x0112=Missing type name
0x0113=Invalid SubPtr
0x0114=SubPtr out of block
0x0115=SubPtr out of database
0x0116=Inconsistent element size and block size
0x0117=Inconsistent project block size and database size
0x0118=Inconsistent element size
0x0119=Unexpected SubPtr
0x0120=Inconsistent sub kind
0x0121=Invalid sub element offset
0x0122=No filler needed
0x0123=Hole in record type

0x0124=Missing SubPtr
0x0125=Name is not a type

0x0126=Wrong Index type sized
0x0127=Array type size wrong
0x0128=Element type wrong
0x0129=Full name is not valid
0x0132=Invalid Net name
0x0133=No net types specified
0x0134=Invalid Node Address

0x0135=Invalid Port reference

0x0140=Invalid element kind
0x0141=Invalid element index

0x0142=No size expected
0x0143=Size expected

0x0144=Unexpected UpPtr error
0x0145=Missing UpPtr error
0x0146=Inconsistent type size
0x0147=Unexpected offset
0x0170=No element (SWNo or Channel) with the same swno as this PortTemplate

// HUGOMIB
0x0150=No Default Project Found
0x0151=No Project Found
0x0152=No Project File Found
0x0153=No Valid Handle
0x0154=Project Not Open
0x0155=Netdescription Not Found

// Kind table/ subkinds
0x0160=Invalid kind error
0x0161=Invalid sub kind error
0x0162=Invalid sub kind for this kind error

//MIBClipBoard
0x0163=ClipBoard Is Empty
0x0164=Memory copy failed

// convert
0x0200=Can not convert MIB file in View Mode

// Net
0xFFEE=No "IDC" Information in registration database
0xFFEF=No "NetType" Information in registration database
0xFFF0=No Specific NetTypes Key in registration database
0xFFF1=No "NetTypes" Key in registration database
0xFFF2=No "Projects" Information in registration database
0xFFF3=No "Default Portnumber" Information in registration database
0xFFF4=No "Current Workspace" Information in registration database
0xFFF5=No "Workspaces" Key in registration database
0xFFF6=License NetName used
0xFFF7=No route to net
0xFFF8=No fieldbus address
0xFFF9=Illegal combination of nettypes

0xFFFA=No address
0xFFFB=No nets on this PC
0xFFFC=Too many nets
0xFFFD=Error in registration database
0xFFFE=Unknown nettype

0xFFFF=Netname not found

8.4 MIBOLE interface errors

// error codes for Read
0x0001=Error while opening MIB file
0x0002=Could Not Read Element Fields
0x0003=No parent element
0x0004=No next element
0x0005=No Previous element
0x0006=No child element
0x0007=Error while saving project
0x0008=Error while deleting element
0x0009=Error while adding Subelement
0x000A=Error while adding next element
0x000B=Error in Common Data for element
0x000C=Error In Element data (Kind: None)
0x000D=Error In Element data (Kind: ArrayType)
0x000E=Error In Element data (Kind: EnumeratedType)
0x000F=Error In Element data (Kind: Node)
0x0010=Error In Element data (Kind: Alias)
0x0011=Error In Element data (Kind: Constant Or EnumeratedName)
0x0012=Error In Element data (Kind: UserData)
0x0013=Error In Element data (Kind: NodeType)
0x0014=Error In Element data (Kind: Register Or RecordField)
0x0015=Error In Element data (Kind: SWNo Or Channel Or VirtualName)
0x0016=Wrong Kind
0x0017=Error while setting Index
0x0018=Error while reading or writing elementFields
0x0019=Error while running ConsistencyCheck
0x001A=Error In SetPhysId
0x001B=PortNo Out Of Range
0x001C=Obsolete Property
0x001D=NetNo Out Of Range
0x001E=Application Name to long
0x001F=Can't open MIB file, it is opened in edit mode by:
0x0020=Can't convert MIB file when "AllowEdit" = FALSE
0x0021=Invalid name
0x0022=Project not allocated
0x0023=Error while copying element
0x0024=Error while pasting element

0x0025=Error while creating file
0x0026=Editing not allowed (AllowEdit = FALSE)
0x0027=Can not read this property (Read Only)
0x0028=Port Name is too long
0x0029=Invalid Port Name
0x0030=Invalid Net Type
0x0031=No child element
0x0032=Can not write to this kind
0x0033=Not Supported
0x0034=Invalid Value
0x0035=Invalid Property for this object
0x0036=Invalid Methode for this object

8.5 P-NET IDC errors

0xFF00=No Fields
0xFF01=Error in opening file
0xFF02=Not a Program channel
0xFF03=Wrong Codetype
0xFF04=Wrong interpreter version
0xFF05=Program too large
0xFF06=Program too small
0xFF07=Application error
0xFF08=Unknown error
0xFF0A=Sequence stopped
0xFF0B=Not implemented yet
0xFF0C=Not supported
0xFF0D=Not connected
0xFF0E=VIGO not connected
0xFF0F=No VIGO License

8.6 VIGO errors

[DRIVER ERRORS]
0xFF01=Parameter missing
0xFF02=Not implemented yet
0xFF03=No Instruction/Data Converter or net selected
0xFF04=Read protected
0xFF05=Write protected
0xFF06=MIB error
0xFF07=Memory error
0xFF08=Size error
0xFF09=Unable to open file
0xFF0B=VIGO object not connected
0xFF0D=Conversion error
0xFF0E=VIGO not connected
0xFF0F=Value out of range
0xFF10=Incompatible types

0xFF11=Simulation file error
0xFF12=Invalid string
0xFF13=Not a number
0xFF14=Cannot find/load IDC
0xFF15=Invalid datatype, TestAndSet requires boolean
0xFF16=No PhysId found, cannot create Sim file
0xFF17=Negative size can not be written

9 Appendix A

The following tables will prove to be useful to programmers designing application programs, who wish to use the facilities, offered by VIGO.

Table 1 contains the Properties, which can be used in VIGO STANDARD (*VIGO.STD*).

Table 2,3 and 4 show the Properties and Methods, which can be used in VIGO PROFESSIONAL (*VIGO.PRO*).

Table 5 and 6 contain a short description of the *Kinds/elements* that are used in the *MIB* database.

Table 7 contains the data types that are defined for P-NET modules, which VIGO also uses.

Table 8 contains the *Object Types* for all Standard P-NET Channels. In addition, the *Object Types* for some company specific channel types, owned by PROCES-DATA A/S, are also given.

Further information about the P-NET Standard Channels can be found in the *"P-NET Standardized General Purpose Channel Types"* manual, from the International P-NET User Organization.

Table 9 contains the values for *Capabilities and ObjectType* for a selection of PROCES-DATA modules, which may be used in VIGO. In the *MIB* definitions, the *Capabilities* and *Objecttype* for all *NodeTypes* must be set to the correct value, in order to list the appropriate relevant tools from the right mouse menus in the *MIBOCX*.

IMPORTANT NOTE TO PROGRAM DEVELOPERS:
Most application programmes using VIGO create objects using VIGO Standard (*VIGO.STD*). These applications use the *PhysID* to select the variable to access. By setting the *PhysID* property for an object, all other associated properties are automatically set in accordance with the *MIB* contents.

In cases where a program developer is using VIGO Professional (*VIGO.PRO*) objects, and, for some reason, wants to modify some of the properties, it is entirely the programmers' responsibility to set ALL the other related properties to ensure that these are compatible. Otherwise, errors may occur in transmission or data conversion.

Name	Data type	Description	OLE2 Type
PhysId ErrorCode ErrorString	String Integer String	Identifies the physical object. This function fills out the specification of the physical object obtained from the Manager Information Base. This variable contains a unique error code, in case an error occurs when accessing an object property or method. This variable contains an error message in plain text. This could be used in a Message Box.	Property Property Property
Value	Variant	Used to operate on all data types. This Property must be used for directly receiving and sending data to variables in VIGO STANDARD.	Property

Table 1 Properties for VIGO Standard (*VIGO.STD*)

Name	Data type	Description	OLE2 Type
PhysId ErrorCode ErrorString	String Integer String	Identifies the physical object. This function fills out the specification of the physical object obtained from the Manager Information Base. This variable contains a unique error code, in case an error occurs when accessing an object property or method. This variable contains an error message in plain text. This could be used in a Message Box.	Property Property Property
Value	Variant	Used to operate on all data types. This Property must be used to receive and send data directly to variables in VIGO PROFESSIONAL.	Property
DoRead DoWrite SubPhysId InValue	Void Void String Variant	Used to start reading data into the Buffer. Data can be read using InValue Used for writing to the contents of the Buffer. Data can be send to the buffer using InValue Identifying a simple element in a complex structured variable, identified and obtained by PhysId, and located in a Virtual Object. Used to operate on all data types. This Property must be used to receive and send data to the Buffer, when using the DoRead or DoWrite Methods in VIGO PROFESSIONAL.	Method Method Property Property
DataReady Progress StopSequence EnableExceptions	Boolean Float Void Boolean	This property indicates if the command has finished and the data are ready when the Buffer is used to Read / Write data. Indicates the progress of a certain command, e.g. how much (%) of a file is downloaded. Used to stop a sequence running within the IDC, e.g. download or upload. Used to enable exception handling procedures build into the virtual object.	Property Property Property Property

Table 2 Properties and Methods for VIGO PROFESSIONAL (*VIGO.PRO*)

Name	Data type	Description	OLE2 Type
Download	Void	Download a domain to a node. Function returns immediately. Filename must be written in the Fileame property before Download is called. The Progress property can be read during the Download process.	Method
Upload	Void	Upload a domain (program) from a node. This method is not implemented in VIGO 5.0.	Method
DeleteDomain	Void	Delete a domain within a node.	Method
Start	Void	Start a program execution.	Method
Stop	Void	Stop a program execution.	Method
Resume	Void	Resume a stopped program.	Method
Reset	Void	Reset a stopped program.	Method
Kill	Void	Kill a program execution	Method
SelectProgram	Void	Select a domain to be part of a program invocation. Domain is passed as parameter.	Method
UnselectProgram	Void	Unselect the domain within a program invocation. Domain is passed as parameter.	Method
ProgramState	Integer	Program invocation state.	Property
ProgramName	String	The name of the domain used within the Program Invocation.	Property
FileName	String	File name for the file to Down Load or Up Load	Property
Vendor *	String	Not implemented	Property
ModelName *	String	Not implemented	Property
Revision *	String	Not implemented	Property
ExAnd	Variant	P-NET specific. The passed parameter is and'ed with the data specified by PhysId, eg. VigoObj.And(Var)	Method
ExOr	Variant	P-NET specific. The passed parameter is or'ed with the data specified by PhysId, eg. VigoObj.Or(Var)	Method
TestAndSet	Variant	P-NET specific. The value of the returned parameter depends on the TestAndSet conditions, eg. res =VigoObj.TestAndSet(Var)	Method

Table 3 Properties and Methods for VIGO PROFESSIONAL (*VIGO.PRO*)

* Vendor, ModelName and Revision can be obtained from the ServiceChannel. (Ref. P-NET Standard).

Name	Data type	Description	OLE2 Type
IDCNo	Integer	This property contains information about which Instruction/Data Converter is to be used for communication with the end Node. This property is mainly used by VIGO to determine which IDC is to be called.	Property
NodeAddress	String	This property contains a network address determined by the HUGO2 format. It contains the route to the end Node.	Property
InternalAddress	LongInteger	This is an address that is local for the single Node. E.g. SoftWire number or socket number.	Property
Offset	Long	This is an internal node parameter, e.g. used to access a specific offset for a complex variable.	Property
SubOffset	Long	This property is similar to the Offset property, but it is only used when SubPhysID is used.	Property
BitNo	Byte	This is an internal node parameter.	Property
Size	Long	This property contains the size of the data structure in bytes.	Property
SubSize	Long	This property is similar to the Size property, but it is only used when SubPhysID is used.	Property
ObjectType	Integer	Description of a specific node object, e.g. for P-NET it could be a analogue channel, digital channel, etc.	Property
DataType	Integer	Defining the type that is to be requested within a node. Only used for non standardized types.	Property
SubDataType	Integer	This property is similar to the DataType property, but it is only used when SubPhysID is used.	Property
InformationInErrorCode	Boolean	Enables that Historical Errors are visible in the ErrorCode.	Property
ReadAccess	Boolean	Indicating that the Property/Method is read only.	Property
WriteAccess	Boolean	Indicating that the Property/Method is write only.	Property
MaxRetry	Short	Reserved for future use.	Property
NodeCapabilities	String	Indicating the capabilities of the node, e.g. support bitno, offset, etc. Readout is hexadecimal.	Property
PhysAddress	Boolean	Enables physical addressing. The physical address must be set using InternalAddress.	Property
OnlineAccess	Boolean	Enables that the data are accessable from a file and not from the network. This may be used for simulation purposes.	Property
SetVIGOMessage	Void	This method is used to specify a message to receive when a DoRead or DoWrite method has completed	Method

Table 4 Properties and Methods for VIGO PROFESSIONAL (*VIGO.PRO*)

Kind	Type	Description
Project	Project	The Project holds information about the whole Project description. The data are stored in a MIB-file.
BasicType	Type	Boolean, Byte, Char, Word, Integer, LongInteger, Real, LongReal, OldReal, Timer, RealDate and OdDate are all BasicTypes, from which all other types are constructed.
NodeType	Type	NodeTypes holds information about the entire data structure to use in a node. The elements (channels and softwire numbers) inside are all set up as SubElements
ChannelType	Type	A ChannelType holds a description for a channel in a Node. The elements (Registers) inside are set up as SubElements.
RecordType	Type	Complex structures for variables are set in a RecordType. The elements (RecordFields) inside are set up as SubElements.
Enumerated	Type	Enumerated holds identifier for logic names used in reeling off. The elements inside (EnumeratedName) are set up as SubElements.
ArrayType	Type	In ArrayType an array of a specified type can be created.
BufferType	Type	In BufferType a buffer of a specified buffer element type can be created.
BitArrayType	Type	In BitArrayType a boolean array can be specified. Instead of occupying a byte to each boolean value, the BitArrayType only uses one bit for a boolean.
SetType	Type	To give special values to a type, SetType is used.
StringType	Type	StringType is used to define a string, which consists of a length (in bytes) and an array of char.
VirtualRecordType	Type	The VirtualRecordType is used to give a Virtual description of the physical plant. The elements inside VirutalRecordType (Alias, Constant, VirutalRecordType and VirtualArrayType) are set up as SubElements, and can be accessed using these subnames.
VirtualArrayType	Type	The VirutalArrayType is used to give a Virtual description of the physical plant. The elements inside VirutalArrayType (Alias, Constant, VirtualRecordType and VirtualArrayType) are set up as SubElements, and can be accessed by using index values.
BitMapType	Type	To define a Bitmap of a specific type, BitMapType is used.
PointerType	Type	A Pointer to another Type definition is created from PointerType
Procedure	Type	To reserve a name for a Procedure in an application the Kind: Procedure is used.
Function	Type	To reserve a name for a Function in an application the Kind: Function is used.

Table 5 Kinds of *MIB* elements used in a Project Description

Kind	Type	Description
Node	Variable	Nodes are used to gain contact to modules within the physical plant. Nodes are only set up from the kind: NodeTypes.
Channel	Variable	Channels are used in the description of a NodeType. Channels are set up from the kind: ChannelType.
Register	Variable	Registers are used to describe all variables inside a ChannelType. Registers can be set from all Type Kinds, except ChannelType and NodeType.
SWNumber	Variable	If the SoftWire number is known, this can be given directly. SWNumbers are used in NodeType.
RecordField	Variable	A RecordField holds all Type Kinds, except ChannelType and NodeType. RecordFields can only be used inside a RecordType.
Constant	Constant	To set a Constant in the Project description the Constant element can be used.
EnumeratedName	Constant	As SubElements to Enumerated, EnumeratedNames are used. The Names represent a specified value.
Alias	Variable	Aliases are used to set up pointers, which can act as short cut to elements within the Project description.
VirtualName	Variable	In Virtual descriptions the VirtualName consist the VirtualRecordType or the VirtualArrayType to use.

Table 6 Kinds (Variables) in the *MIB*

P-NET Data Type	Number		Length in bytes
	Hex	Decimal	
Empty	0x20	32	-
Integer	0x22	34	2
LongInteger	0x23	35	4
Real	0x24	36	4
LongReal	0x25	37	8
RealDate	0x27	39	8
String	0x28	40	-
Boolean	0x2B	43	1
OldDate	0x2E	46	8
Byte	0x31	49	1
Word	0x32	50	2
UserDefined	0x3D	61	-

Table 7 List of datatypes for P-NET modules.

The object types for P-NET Standard Channels and some company specific channels are found in the following table.

Object	Description
0	Object type is not used or the object type is a non-standard type
1	Service channel
2	Digital IO channel
3	Common I/O channel
4	Analog measurement channel
5	Current output channel
6	PID-regulator channel
7	Calculator channel
8	Pulse Processor channel
9	Printer channel
10	Weight channel
11	Program channel
12	Power Monitor channel
14	Communication channel
32769	PROCES-DATA specific Data channel
32770	PROCES-DATA specific Common I/O channel
32771	PROCES-DATA specific Thyristor Switch
32773	PROCES-DATA specific Key/Mouse
32774	PROCES-DATA specific Display
32775	PROCES-DATA specific GateWay
32776	PROCES-DATA specific Generator Switch

Table 8 Object types for P-NET Channels

The following table shows the object type and the capabilities for a selection of standard modules from PROCES-DATA A/S.

Module number	**ObjectType**	**Interpretation of capabilities**	**Capabilities**	**Hexadecimal value**
PD340	340	NoOffset,NobitAddress,OldType	7	7
PD1611	1000	NoOffset,NobitAddress,OldType	7	7
PD3100	1000	NoOffset,NobitAddress	3	3
PD3120	1000	NobitAddress	2	2
PD3150	1000	NoOffset,NobitAddress	3	3
PD3221	1000	NobitAddress,NoOffsetInlong	130	82
PD3230	1000	NobitAddress,NoOffsetInlong	130	82
PD3240	1000	NobitAddress	2	2
PD3250	1000	NobitAddress	2	2
PD3260	1000	NobitAddress	2	2
PD3920	1000	NobitAddress	2	2
PD3930	1000	NobitAddress	2	2
PD3940	1000	NobitAddress	2	2
PD3000	3000	NobitAddress, ExtendedPNET	34	22
PD4000	4000	NobitAddress, ExtendedPNET	34	22
PD4500	4500	ExtendedPNET	32	20
PD5000	5000	ExtendedPNET	32	20
PD5010	5000	ExtendedPNET	32	20
PD5015	5000	ExtendedPNET	32	20
PD5020	5000	ExtendedPNET	32	20
PD600	6000	ExtendedPNET	32	20
PD601	6000	ExtendedPNET	32	20
PD602	6000	ExtendedPNET	32	20

Table 9 Object types for PROCES-DATA modules.