Valmet DNA Engineering

Automation Language

Collection 2017 rev. 3

G5095_EN_03

Valmet FORWARD

Valmet Automation Inc.
P.O. Box 237, FIN−33101 Tampere, Finland
Tel. +358 (0)10 672 0000
www.valmet.com/automation

Document History

Date	Revision	Comment
13.10.2017	**3**	**Valmet DNA Collection 2017** **No changes in contents.**
16.08.2016	**2**	**Valmet DNA Collection 2016** **No changes in contents.**
29.09.2015	**1**	**Valmet DNA Collection 2015** **No changes in contents.**

Contents

1 Introduction 1

2 Automation Application 2

3 Concepts of the Automation Language 4

3.1 Modules 4

3.1.1 Automation Modules 4

3.2 Document Modules 6

3.3 Configuration Modules 6

3.4 Type Modules i.e. Types 7

3.4.1 Data Types 7

3.4.2 Function Block Types i.e. Function Blocks 11

3.4.3 Bundle Types 12

4 Application Program Elements 13

4.1 Data Points 13

4.1.1 Local Data Points 13

4.1.2 External Data Points 14

4.2 Ports 15

4.2.1 Direct Access Port 15

4.2.2 Interface Port 17

4.3 Function Blocks 17

4.3.1 Configuration Parameters 19

4.3.2 Connection Parameters 19

5 Application Program Written in the Automation Language 22

5.1 The Structure of an Automation Module 22

5.2 The Structure of a Configuration Module 24

5.2.1 Administration Part 24

5.2.2 Representation Part 26

5.2.3 Functional Part 35

5.3 Internal Connections in a Module 42

5.4 Communication Between Modules 46
5.4.1 Using a Direct Access Port in Communication 47
5.4.2 Using an Interface Port in Communication 49
5.4.3 Using a Viewpoint in Communications 50
5.5 An Example of the Structure, Connections and Communications of a Configuration Module 51

6 The Automation Language's Naming Conventions 55
6.1 General 55
6.2 Structure and Length of Module Name 55
6.3 Characters That Can Be Used in Module Name 55
6.4 Directory Identifier in Module Name 56
6.4.1 Automation Modules 56
6.4.2 Configuration Modules 56
6.5 Control Room Identifier in Module Name 58
6.6 Tag Part in Module Name 58
6.7 Designation of Display Modules 59
6.8 Designation of System Modules 59
6.9 Designation of Modules Produced by Operation Server 60
6.9.1 Designation of History Modules 60
6.9.2 Designation of Base Modules 61
6.10 Modules by Tools 61
6.11 Module Destination Data 63
6.11.1 Structure of the Destination Data 63

7 Fault Bit Conventions 64
7.1 Meanings of Fault Bits 64
7.2 What Different Fault Bits Indicate 64
7.2.1 ext – External Fault 65
7.2.2 ovf – Data Overflow 65
7.2.3 dis – Control Disabled 65
7.2.4 inv – Invalid Data 65
7.2.5 old – Old Data 66
7.2.6 der – Fault on Derived Data 66
7.2.7 sex – Source Exceptional 66

7.3 Signal Alarms from Fault Bits 66

7.4 Some Notes to Be Noted in Using Fault Bits 66

7.5 On Applying Fault Bit Conventions 67

7.5.1 Initial Values of Types 67

7.5.2 Initial Values in Modules 67

7.5.3 Data to Be Connected 67

Appendix 1 Primitive Types

Appendix 2 Common Structured Types

1 Introduction

This document has been written for the application engineer of Valmet DNA.

The document describes the basic concepts of the automation language and the structure of an application program configured in the automation language. In addition, it discusses the different features that support the user of the automation language, as well as the different formats of automation–language program modules in Valmet DNA.

Automation language is used to define the operation of Valmet DNA, in other words, to configure its application software. It is a connection type language based on function blocks and versatile types, and it can be extended, for instance, with expressions for calculations, logic operations and comparisons, and with a list–form representation of graphics.

The purpose of the automation language is to create a clear and logical model of Valmet DNA and its configuration data for the application designer. Owing to its limited range of applications, the language is able to give extensive support to the user. It provides a considerably higher efficiency in *automation design* than general purpose programming languages (such as C).

Special design software packages provide an effective user interface for the application designer. They support the application designer, for example, by offering tools for graphic design.

2 Automation Application

The automation application i.e. *automation software* consists of automation language *modules* (Figure 1).

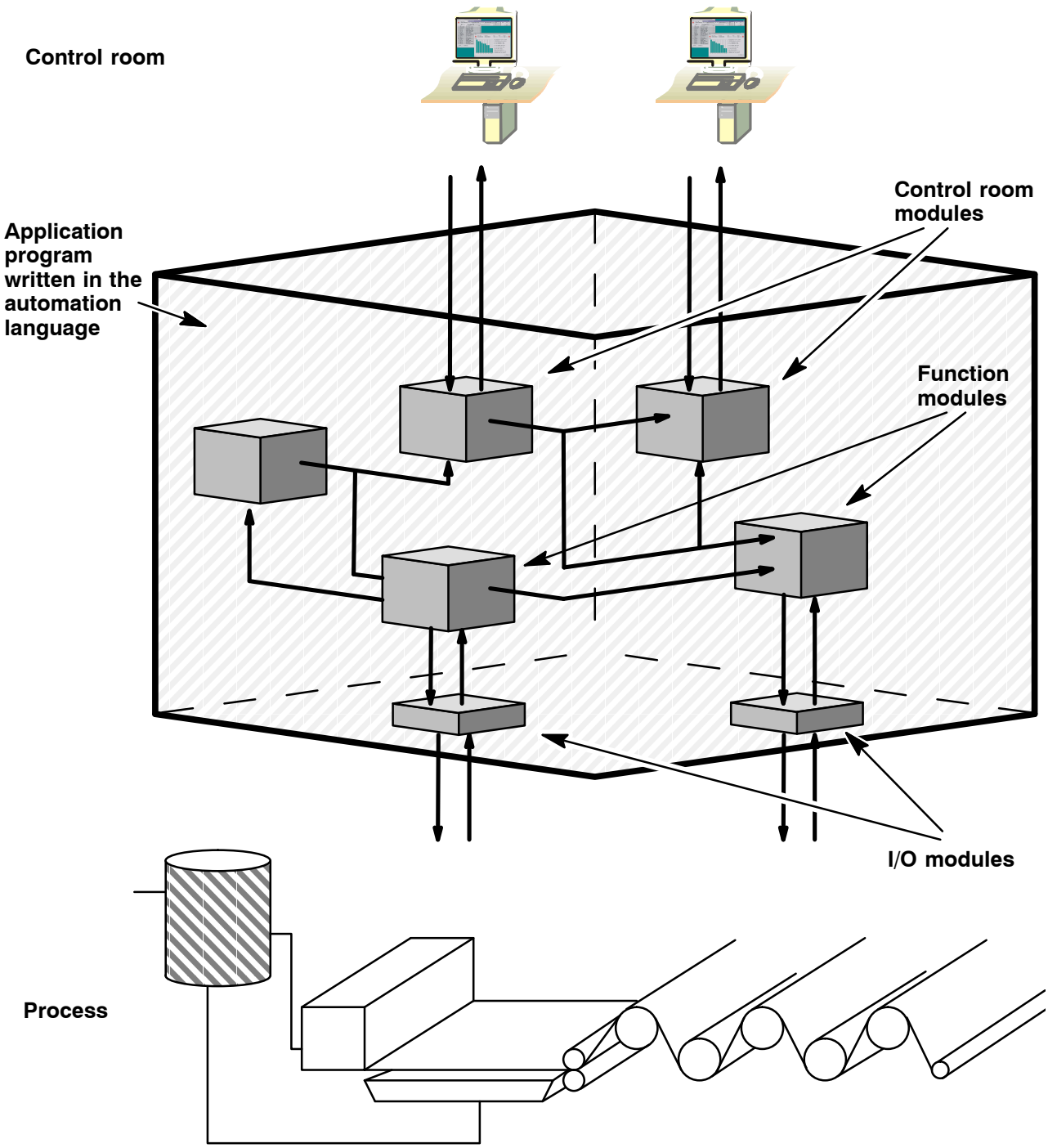


Figure 1 Automation–language application program

From the application enginener's point of view and in terms of process control, the module is a logical unit. Modules are the smallest program blocks that can be downloaded separately to the application servers.

A module can be an individual control or measurement loop, an entire sequence control program, a motor group control program or a display shown on the monitor screen.

The concepts of the automation language give the engineer a model of Valmet DNA and its configuration as shown in the following picture (Figure 2).

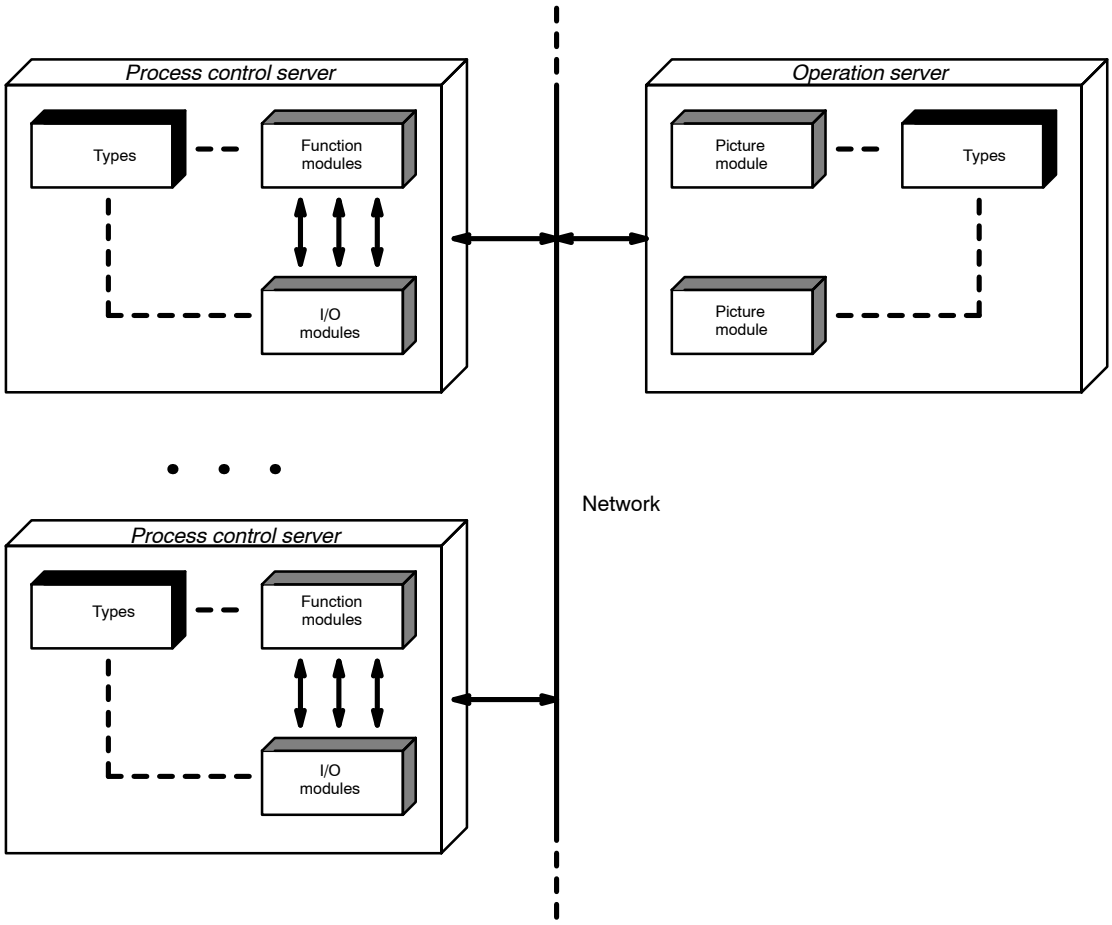


Figure 2 *A designer's view of Valmet DNA*

The engineer perceives activities and components in Valmet DNA, to which he places the modules he has built. Modules are made of different parts, which are defined by types. The application server type components always contain standard library type modules. The modules communicate with each other by copying required data and together constitute the automation application.

The modular structure of the automation language supports the engineer by enabling the use of defaults and previously configured blocks. However, the defaults are treated as defaults only at the engineering workstation; once the data is transferred to the application servers, there is no way for you to tell if it is a default or a value entered by the engineer.

3 Concepts of the Automation Language

3.1 Modules

Since the use of the automation language involves the handling of many different types of data, we have created specific concepts for each of them. The fundamental elements of the automation language include different modules. The modules are further divided to four main groups.

The basic concepts of the automation language are as follows:

- automation modules
- document modules
- configuration modules
- type modules i.e. types

3.1.1 Automation Modules

The user interface for the application engineer makes use of graphic utilities, which make the design more illustrative. Pictures produced with these graphic design tools are converted to an automation language program i.e. configuration modules, that can be downloaded to the Valmet DNA application servers. The pictures produced with the graphic tools are called *automation modules*.

An automation module is a graphic representation of a part of an application program. A graphic representation of a module can be returned to a list form (automation language), but not vice versa. In other words, it is not possible to convert a program listing to a graphic module.

The automation modules may contain a number of configuration modules. Nearly all modules related to a specific loop can be integrated in a single graphic automation module.

Automation modules can be designed using versatile design and editing functions and libraries containing symbols needed in automation modules.

The following figure (Figure 3) shows an example of an automation module. The automation module shown in this example will be converted to the following process control server and control room configuration modules:

- process control server function module
- process control server input and output modules
- control room tag module
- control room event module
- control room operation display module

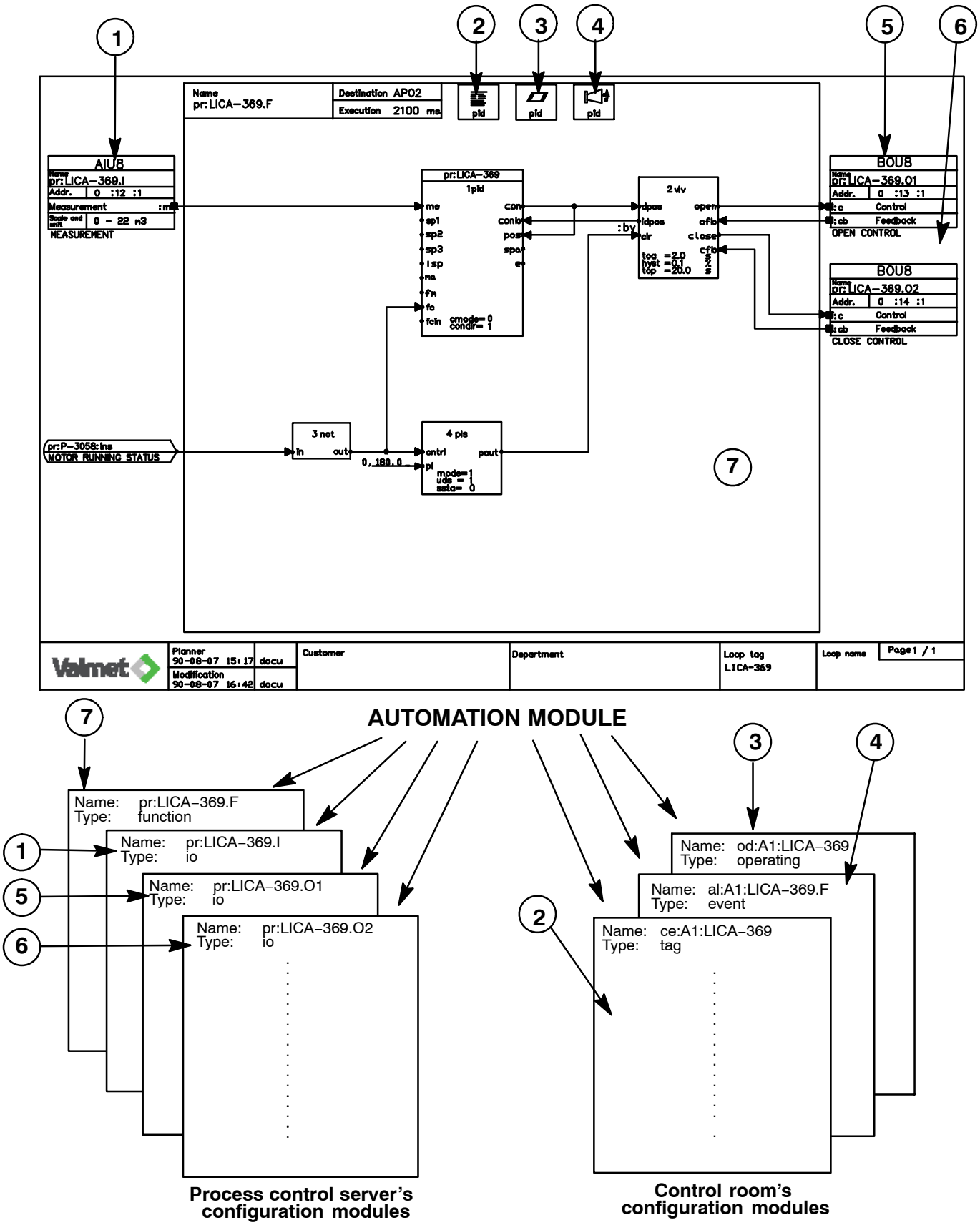


Figure 3 Automation module and related configuration modules

3.2 Document Modules

As their name suggests, the document modules are simply documents that describe an application or the structure of the network.

The document modules differ from automation modules so that no automation language program, i.e. configuration modules, is generated from document modules.

The following types of document modules are available:

- loop circuit documents
- circuit diagram documents
- logic diagram documents
- hardware documents
- field drawing documents

3.3 Configuration Modules

The operation of Valmet DNA is actually defined by an application program consisting of configuration modules. A number of modules is defined for the operations: function modules, I/O modules, picture modules etc.

Configuration modules are functional units and basic units of the automation language which are connected to create an automation–language application program. The application engineer is able to handle the modules on the engineering work station and download them to application servers without disturbing the operation of the application.

Configuration modules are built on function blocks, ports and data points. They can be completed with calculations, logic operations and comparisons by adding blocks written in a high level language; these blocks are however formulated like function blocks. The set of configuration modules resulting from the connections can also be called the application software of Valmet DNA.

3.4 Type Modules i.e. Types

Type modules define types used in the automation language: *function block types, data types and bundle types*. The application servers of Valmet DNA always have a selection of fixed library types; the application engineer cannot himself design new type modules.

The types of the automation language are defined on the basis of general type concepts. The types are divided into three categories on the basis of their role:

- data types
- function block types
- bundle types

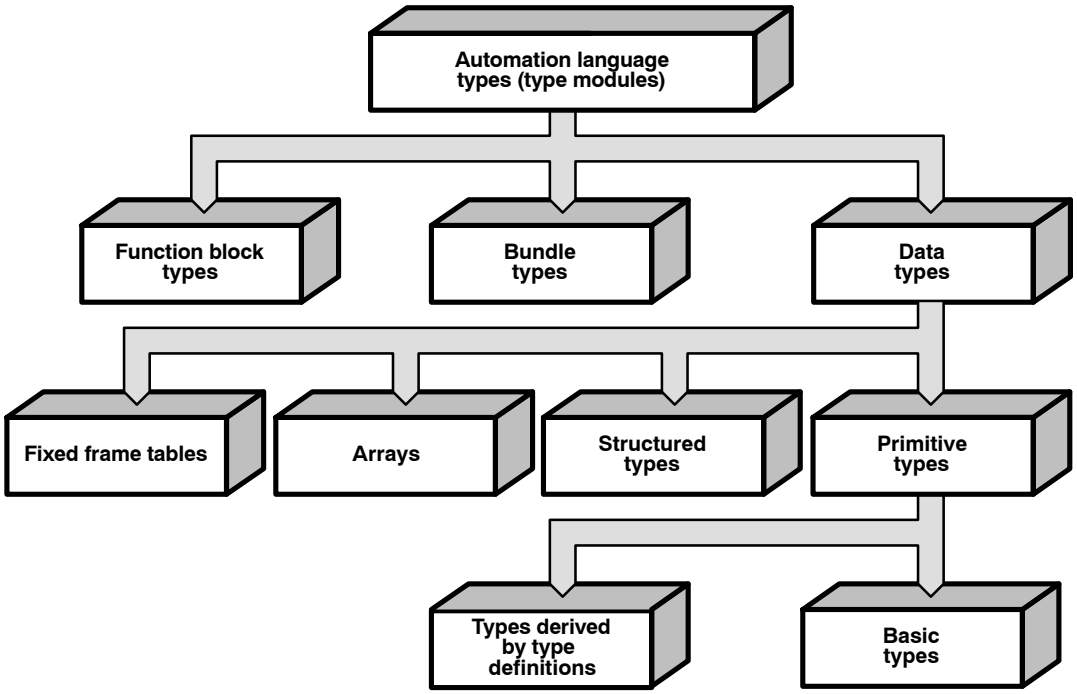


Figure 4 Type hierarchy of automation language

3.4.1 Data Types

A data type in the automation language is an integrated block of data. Automation language data types are:

- primitive types
- structured types
- arrays
- fixed frame tables

PRIMITIVE TYPES in Valmet DNA are *fixed types*, i.e *basic types* and simple *derived types* defined using the basic types.

Basic types include:

- uns16 = unsigned integer, size 2 bytes, range (0...65535)
- float = single precision floating point number, size 4 bytes, range $\pm$ (10^{-38}...10^{38})

An example of a *derived primitive type* is the primitive type *fails* derived from the fixed basic type unsigned integer (*uns16*). It is defined as follows:

```
fails TYPE uns16
```

Fails has the same characteristics as *uns16*, in other words, a derived primitive type is actually formed by renaming a basic type.

The primitive types of the automation language (both basic types and derived types) are presented in Appendix 1.

Structured Types

Structured types are defined on the basis of other types and they can contain several members. Possible members include all primitive types or structured types. However, if necessary, all members can be reduced back to primitive types.

Example:

The structured type *ana* consists of fault bits and signal.

Its first member is the fault bit field *f* and the second member the actual value of the analog signal *a* expressed in SI units. Member *f* is a derived type from the primitive type *fails*, and *a* is a derived type from basic type *float*.

The structure of *ana* is thus defined as follows:

```
ana
MEMBERS
      f TYPE fails
      a TYPE float
```

For instance, an *ana*–type signal (2,7.72) contains

2	f	fault data (input error or line failure)
7.72	a	signal value in SI units

Common structured types of the automation language are presented in Appendix 2.

Array

An array is a data structure whose elements are of one primitive or structured type. The elements can be referenced in a random order by using an index. Valmet DNA automation language permits using arrays with 1, 2 or 3 indices, i.e. 1, 2 or 3 dimensional arrays. The limits of each index are defined when introducing the array.

Example: A two dimensional array quantity is introduced as follows:

```
quantity (2, 3) TYPE ana
```

The limit to the first index is 2 and the limit to the second is 3. The array members are of the structured type *ana*.

The above examples can be combined (Figure 5):

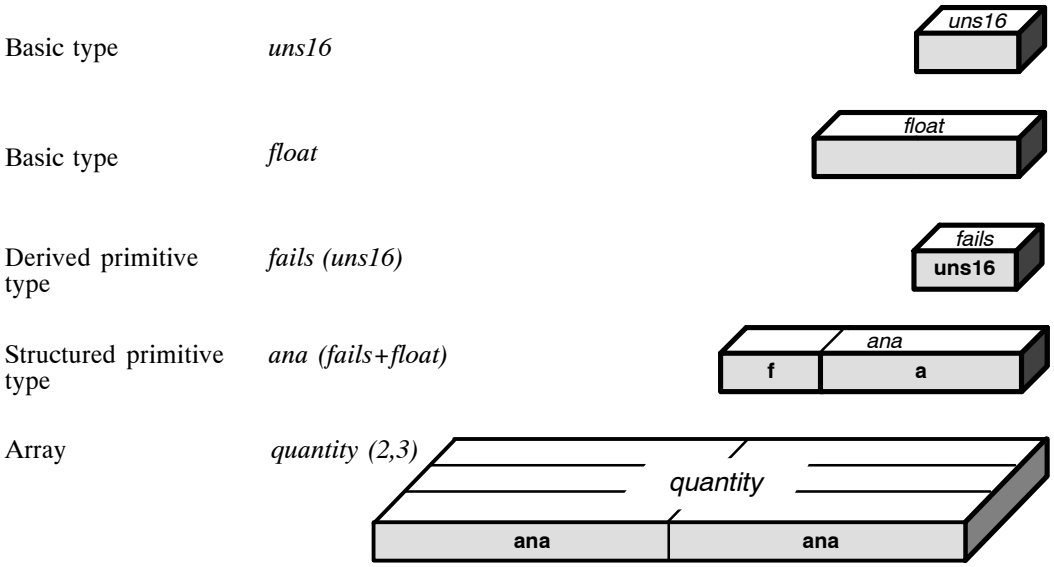


Figure 5 An example of a data type hierarchy

Fixed Frame Table

Fixed frame table is a structure with a fixed frame, specified by the type, containing a table whose structure can be changed at the configuration and execution time.

The type of the fixed frame table specifies the dimensions of the frame. The type also specifies the type of the elements both in the frame and in the table inside the frame. They cannot be changed at the configuration or execution time. Each required combination of element type and dimensions must exist as a separate type. The type collection of each Valmet DNA collection shows which table types that collection includes.

The name of the table type indicates both the type of the elements and the dimensions of the frame. The format of the name is typ_ijk, e.g. *ana_9* and *cha_536*.

First part of the name of the table type (typ) is an abbreviation of the type of the elements. Abbreviations are as follows (typ and the corresponding element type):

Basic types

cha = char	u32 = uns32
i8 = int8	b8 = bool8
i16 = int16	b16 = bool
i32 = int32	flo = float
u8 = uns8	fls = fails
u16 = uns16	–

Common types

ana = ana	inl = intl
bin = bin	bne = binev
ins = ints	ise = intsev

Application types

bo = bo

The second part of the name of the table type (ijk) indicates the dimensions of the frame, one character for each dimension. The frame table can be at most 3–dimensional. i, j and k are encoded exponents indicating a power of two. The codes of exponents and the corresponding values of dimensions are (code and dimension):

0 = 1	8 = 256
1 = 2	9 = 512
2 = 4	a = 1024
3 = 8	b = 2048
4 = 16	c = 4096
5 = 32	d = 8192
6 = 64	e = 16384
7 = 128	–

For example, *ana_9* is 1–dimensional table. The type of its elements is *ana* and the dimension of its frame is 512. *cha_536* is 3–dimensional table. The type of its elements is *char* and the dimensions of its frame are 32 x 8 x 64.

Frame table structure

As a data structure, the frame table has two parts. It has a header part (member *:header*) and an element part (member *:elem*). Every frame table has the same type (*table*) of the header part, regardless of the dimensions and the element types of the frame. Frame tables with different dimensions and element types are created by using different element parts and different default values in the header parts.

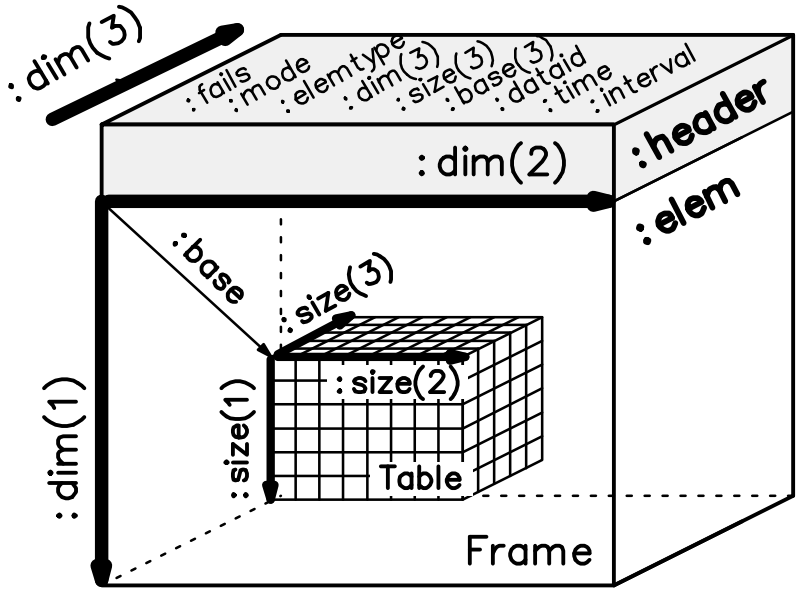


Figure 6 Structure of the frame table

- Header part

  The type of the header part (*:header*) of the frame table is *table*. It includes the structure data of the frame (*:elemtype* and *:dim*) and the table (*:mode*, *:size* and *:base*). It includes also separate scalar values (*:fails*, *:dataid*, *:time* and *:interval*) describing the table.

  The data in the header part can be read and written by addressing the corresponding member, e.g. pr:PM6:BW:header:size(3) or pr:PM6:BW:header:dataid or by addressing the whole frame table, e.g. pr:PM6:BW.

- Element part

  The element part of the frame table (*:elem*) is the frame of the table. It specifies the type of the elements of the frame and the dimensions of the frame. It also specifies the type of the elements of the table and the maximum size of the table.

  The data in the element part can be read and written by addressing the corresponding element, e.g. pr:PM6:BW:elem(3) or pr:QVT:elem(2,6,3) or by addressing the whole frame table, e.g. pr:PM6:BW.

  The element part of the frame table includes two nested parts. The outer one is a fixed frame, specified by the table type and the inner is a variable table, the variability of which is limited by the frame. The frame specifies the maximum size and the type of the elements of the variable table. The table can be changed at the configuration and execution time.

3.4.2 Function Block Types i.e. Function Blocks

Modules consist mainly of function blocks. Function blocks are linked to other module parts through connections. For example, the members of two different function blocks communicate via a connection. The description of a function block type presents the function block members, which determine the structure of connections in the function block.

The types of members in function blocks are defined in the type description (e.g. ana, bin, bo, float).

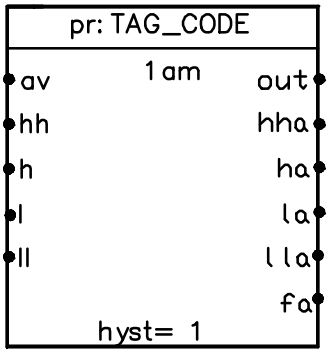
On the basis of their operation, function block members can be divided into:

- inputs
- outputs
- configuration parameters

A function block reads data from inputs and writes data to outputs. Connection between different function blocks is established by interconnecting members (inputs and outputs) of the function blocks. The basic principle in connection is that only members of the same type can be connected: for instance, an *ana*–type signal can be connected only to another *ana*–type signal, a *bin*–type signal to another *bin*–type signal, and so on.

Beside connected members, a function block type may include a number of configurable parameters. They cannot be changed during the running of the application program, since their values are specified once for all while configuring the application program.

In graphic design, the function blocks are represented with the symbol shown below. The function block's inputs (av, hh, ... ll) are shown at one side of the symbol, and the outputs (out, hha, ... fa) on the other side. At the center of the symbol, you can give the block's configuration parameters and their values (hyst = 1).



3.4.3 Bundle Types

Communications between modules make use of ports. Direct communication between function blocks is not possible between modules. A module may have several ports, which are its windows to the outside world. Other modules use these ports for communicating with this module. A module port may be of a bundle type or data type.

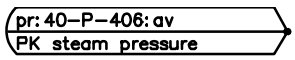
The members of a *bundle type* port permit the collection of data from different points inside the module into a bundle. The data of a bundle type port can be independent of each other, i.e. independent data to be connected which is gathered from inside the module. The data in a bundle are available to other modules through the port.

Bundle types are used, for instance, for communications between a process control server and a control room. For example, the process control server types *cntb* and *am.opb* do not contain any (function block) operations, but control room modules can address them in the same way as the corresponding function blocks *cnt* and *am*.

4 Application Program Elements

The automation and configuration modules consist of the automation language's basic elements which are divided into three groups:

- data points

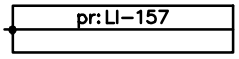


external data point

local data point

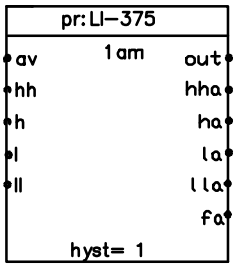
- ports

interface port



direct access port

- function blocks



A data point is defined on the basis of data type, a port on the basis of bundle type or data type, and a function block on the basis of function block type.

A strict typing is applied between the automation language elements inside a module; in other words, only elements of the same type can operate with each other. In communications between modules, this strict typing is eased off in a controlled manner. For a type, other types with which it can communicate are defined. Also different viewpoints are defined to a type: when data is requested from an element, only data corresponding to the specified viewpoint will be picked. Owing to this arrangement, only necessary data will be transmitted. These typings and viewpoints cannot be configured or changed by the application engineer.

4.1 Data Points

Data points can be local or external.

4.1.1 Local Data Points

Local data points are uniquely named areas of data in a module's data space. Function blocks and ports connect to these local data points to exchange data inside a module. A local data point is known only inside a module, not elsewhere. A local data point is needed in certain internal connections of a module such as connections between *calc*, *logic*, and *cmp* (comparison) function blocks.

In graphic design, a local data point is expressed by the following symbol (Figure 7):

Figure 7 Local data point symbol

4.1.2 External Data Points

External data points are located in the same data area as local data points. The communication routine of Valmet DNA copies data to these points from the data points of other modules via ports. The name of an external data point refers to a name, i.e. port, which is known in Valmet DNA.

In a module, all data that comes from outside the module is presented as external data points. Communication reads the data from the interface or direct access port of the source module to the external data point of the target module, or vice versa, from the external data point to the ports of other modules.

An external data point can be of the following type:

- input

  An input type external data point reads data from other modules and its data can be connected inside a module to function block inputs.

- output

  An output type external data point writes data to other modules via ports and it can be connected inside a module to a function block output.

- input–output

  An input–output type external data point can both read data from and write data to a function block and through ports to other modules. This kind of data transfer is used, for instance, in recipe control.

An external data point can communicate via ports with other modules by:

- continuous communication

  Continuous communication is used in most cases, in other words, the module configuration includes setting of the communication frequency (it must be the same as the execution frequency of the module connected to it).

- conditional communication

  Conditional communication is used if it is not necessary to update a data on a continuous basis (for example, in recipes and conditional copy function blocks).

In graphic design, an external data point is expressed by the following symbol type (Figure 8 shaded area).

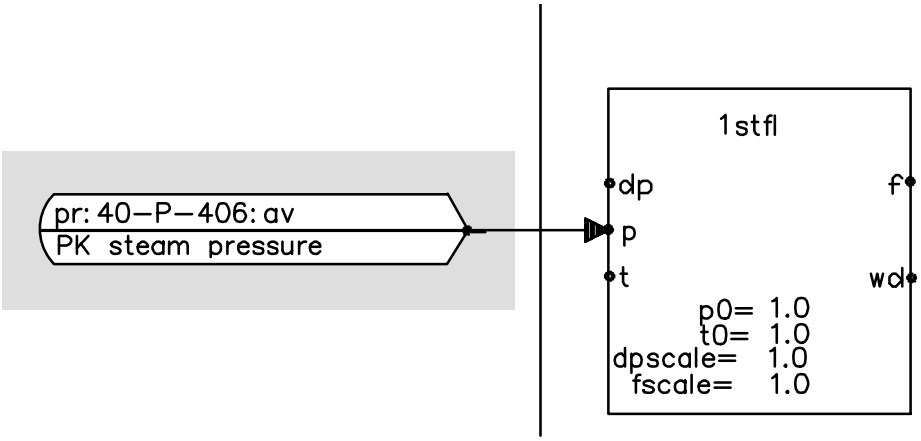


Figure 8 Connection of an external input to a function block

The continuous external input *pr:40–P–406:av*in the figure is connected to the *1stfl* function block input *p*. The function block to which an external data point refers must be defined in the module as a direct access port. The external input in the example uses the direct access port *pr:40–P–406*in module *pr:40–P–406.F*to access the member *av* of the function block (*am*) connected to the port.

An initial value can be given to both data point types (both external and local), and both ports can be accessed from within the module.

External data points should be given appropriate names, for example by deriving the names from tag identifiers. Local data points can be given even simpler names. For instance, in graphic design a local data point is addressed by the letter ”P” and data points are numbered consecutively: P1, P2, P3 etc.

Data points of a primitive type are indivisible; structured data points and array data points are divided into elements, which can be addressed by specifying the index of the element or member.

4.2 Ports

Ports are used for creating module interfaces. They allow different modules of the application network to communicate. Ports can be *direct access ports* or *interface ports*, named with an appropriate identifier.

4.2.1 Direct Access Port

A direct access port is known in the entire application network, in other words, its name is unique in the entire Valmet DNA. Data connected to it can be written/read by using the port name in the entire Valmet DNA. As the name of a direct access port is unique in the entire application network, the same name must not occur elsewhere in the application network.

Either a single piece of data or the entrire function block can be connected to a direct access port. If only a single data point is connected, only that data point is visible outside the module through the port. If the entire function block is connected, all members of the function block can be accessed by other modules through the port.

Following is an example of connecting a single signal to a direct access port (Figure 9). The direct access port (pr:40–FAQ–437) symbol is shown in the shaded area.

In this example connection, the status of output *o1* of the function block *2cmp* can be transferred outside the module via the port, which means that its data can be used in any part of the application network.

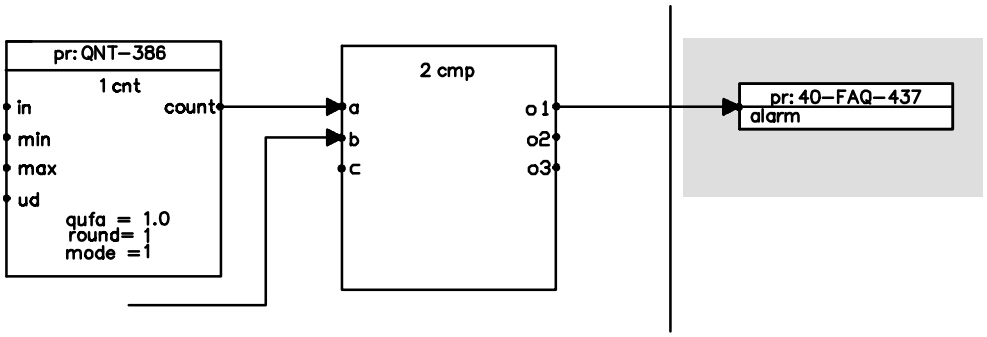


Figure 9 Connecting a direct access port to one signal

Another way to use a direct access port is to connect (name) the entire function block to it. Thus the values of all members of the function block (inputs/outputs/parameters) can be accessed by other modules simply by the name of the direct access port. A direct access port is used mainly by the control room to request display data from a process control server. In addition a process control server has a few function blocks that are almost always assigned a direct access port. These include:

- am = (analog measurement),
- pid = (PID controller),
- mtr, mtre = (motor control),
- mgv, mgve = (magnetic valve control),
- qcnt = (quantity counter),
- cnt = (counter).

The following figure shows a graphic symbol of a direct access port connected to the entire function block. The direct access port is represented by the rectangle (shaded area) at the top of the function block and its name (pr:QRC–1234) is shown in the middle of the area.

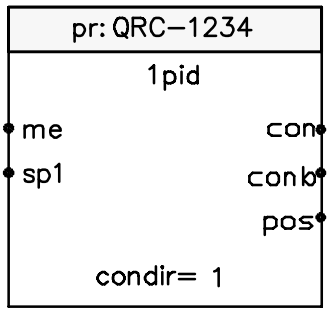


Figure 10 Expressing a direct access port connected to a function block

In the example, the function block members can be accessed by other modules by names, such as: pr:QRC–1234:me, pr:QRC–1234:con.

The modular structure of the automation language supports the engineer by permitting the use of defaults and previously configured blocks. However, the defaults are treated as defaults at the engineering environment only; once the information is moved to the application servers, defaults cannot anymore be distinguished from values entered by the engineer.

4.2.2 Interface Port

A single data point in a function block can be connected to the module's interface port. In this way, single pieces of data can be transferred between modules or application servers.

Data connected to an interface port can be written/read by a name which consists of the name of the configuration module and the name of the actual interface port.

Following is an example (Figure 11), where an interface port is connected inside a module to the output *out* of the function block *5ccob*. This function block output *out* is now accessible from any part of Valmet DNA by the module name and port name as follows pr:S–2237.F:out1. In this example, the module name is pr:S–2237.F and port name *out1*.

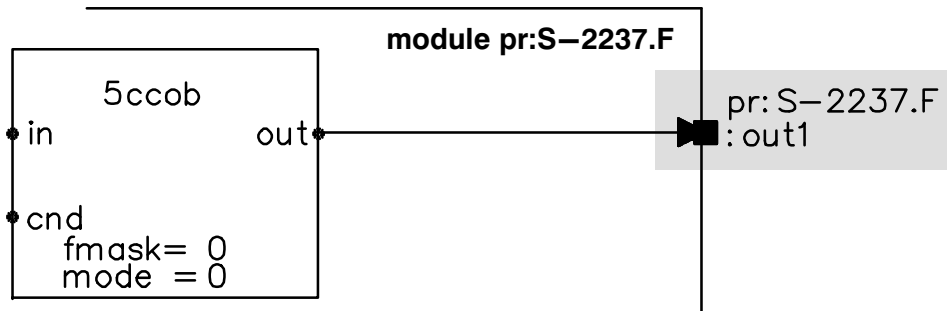


Figure 11 Graphic representation of an interface port

Ports are connected inside a module, so their values can be defined and read through the ports. In terms of their operation, direct access and interface ports are similar.

4.3 Function Blocks

Function blocks carry out a certain function such as control algorithms and connect to their environment through *connection points*. The connection points and configuration parameters of a function block are called function block members. Like other automation language objects, members are also typed.

If a function block writes data to its connection point, it is an output. If it reads data or both reads and writes data, the connection point is named as an input.

The operation of a function block is defined using configuration parameters. Parameters can be given constant values, but they cannot be connected to data points. Function block places its internal status data to its internal members. They are not visible to the user of the function block and they cannot be connected. Depending on its type, a function block contains instructions for handling the data connected to its connection point.

Both inputs and outputs as well as configuration parameters have types assigned to them, which means that only certain data types can be connected to them (this applies to inputs and outputs) or that only certain types of values can be assigned to them (configuration parameters).

The engineer should name function blocks by assigning them an identifier. The identifier is made of the block's number and a type code following it. Examples of possible identifiers are *1pid* and *99hys*. The first block has a number 1 and a type *pid*, the second block is numbered 99 and has a type *hys*.

Inside a configuration module, function blocks are executed in an ascending numerical order.

The following figure shows a process control server function block, which is used in analog measurements.

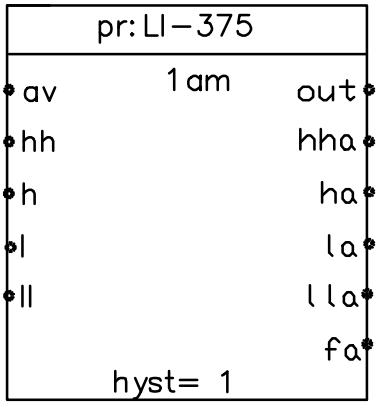


Figure 12 The symbol of analog measurement function block

The *am* function block includes the following:

- two configuration parameters, one of which (hyst) is shown in the symbol
- five inputs
- six outputs

In graphic design, the configuration parameters are entered inside the function block, and they usually define the function block's operation modes. The *am* function block's configuration parameters are:

- **hyst**
  specifies the magnitude of the block's hysteresis; this member is of the type *float*
- **un**
  engineering unit; a comment that does not affect the block's operation

Besides the configuration parameters, the function block includes inputs such as the following:

- **av**
  Value of the measured analog signal
- **hh**
  Value of the higher high alarm limit

A signal or constant value can be connected to each input. The *av* input is of the type *ana*, and the *hh* input of the type *float*.

In addition, the function block includes outputs such as the following:

- **out**
  The operated data, which is sent to the field; the type of *out* is *ana*
- **hha**
  Higher high limit alarm, whose type is *bin*

All members of this function block are provided with initial values that they will have until some data updates them.

The *am* function block can now be presented as follows (all function blocks of Valmet DNA are described in the same way in the function blocks manuals). This presentation format includes the function block's all members, their types and default values, as well as a description of each member's operation or function.

4.3.1 Configuration Parameters

hyst

Type: float
Default: 0.0
Description: Hysteresis

Alarm becomes redundant when analog signal returns to permissible side of alarm limit and differs from the limit by the amount defined by hyst.

un

Type: uns16
Default: 0
Description: Comment

Engineering unit; a comment that does not affect the operation of the function block.

4.3.2 Connection Parameters

Inputs

av

Type: ana
Default: 0 0.0
Description: Analog value

Value of the measured analog signal.

hh

Type: float
Default: 0.0
Description: Higher high limit

The value of higher high alarm limit. Function block allows crossing the alarm limits (Damatic XDi version 6.0 and up). For example high and low limit alarms can be on simultaneously

With a parameter in cpu−configuration file (hook −classic_pid_alarms) the function can be returned back to what it was before: crossing the alarm limits (e.g. hh < h) is forbidden i.e. function block turns the alarm limits back to right order (copies hh as the value of h).

h

Type: float
Default: 0.0
Description: High limit

Value of high alarm limit. For limit check, refer to hh.

l

Type: float
Default: 0.0
Description: Low limit

Value of low alarm limit. For limit check, refer to hh.

ll

Type: float
Default: 0.0
Description: Lower low limit

Value of lower low alarm limit. For limit check, refer to hh.

Outputs

out

Type: ana
Default: 48 0.0
Description: Output

Operateable signal taken to the field. Gets the initial value out:a = av:a at initialization, provided that input signal's fault bits have not been set.

hha

Type: bin
Default: 48
Description: Higher high limit alarm

ha

Type: bin
Default: 48
Description: High limit alarm

la

Type: bin
Default: 48
Description: Low limit alarm

lla

Type: bin
Default: 48
Description: Lower low limit alarm

fa

Type: bin
Default: 48
Description: Input fault alarm

Input signal's fault bits have been set.

The following figure shows an example connecting an *am* function block to other function blocks.

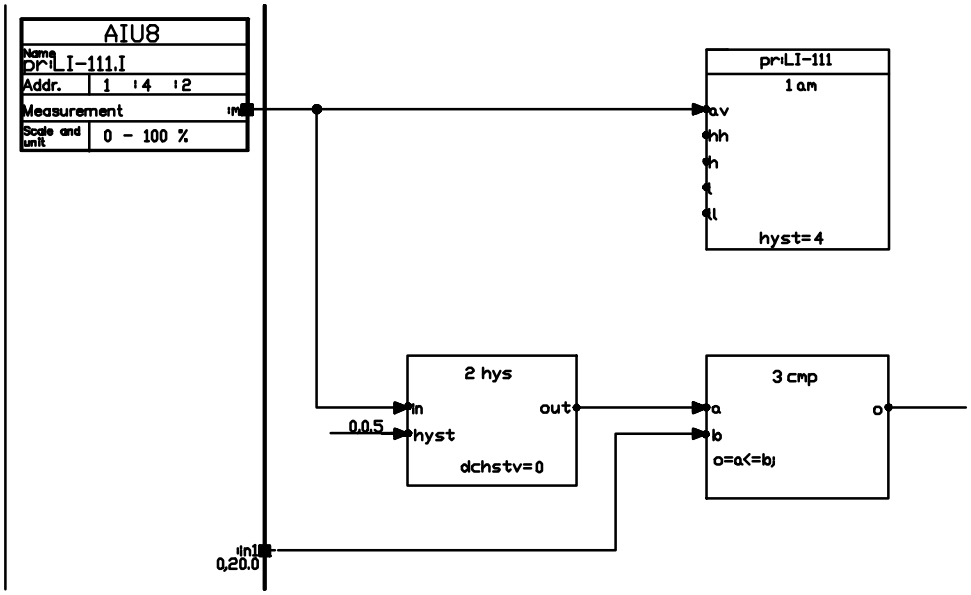


Figure 13 Example of function block connections

After the graphic representation of the function block the corresponding connection for the *am* function block is shown below in list form. (The listing only includes the *am* function block part of the connection.)

```
1am IS pr:LI-111
 hyst=4
 un= -
 av< pr:LI-111.I:m
 hh< -
 h< -
 l< -
 ll< -
 out> -
 hha> -
 ha> -
 la> -
 lla> -
 fa> -
;
```

5 Application Program Written in the Automation Language

An application program written in the automation language consists of units, i.e configuration modules which are sensible both from the point of view of the application engineer and process control. The modules consist of data points, ports, function blocks and algorithmic statement blocks used for calculations, logic operations and comparisons. Configuration modules communicate on the basis of port names which are known to the Valmet DNA data management.

Automation modules are graphic representations of the application software. They can consist of several configuration modules. An automation module can contain the most important configuration modules related with a single control loop, such as the following:

- process control server's function module and I/O modules
- control room's tag, operating display and event modules.

In automation modules, the parts of configuration modules are represented by illustrative symbols, and different signals are coded in different colours to make connection easier.

5.1 The Structure of an Automation Module

An automation module consists of the following parts.

- automation module administration part (1)
- function module administration part (2)
- connection field for external inputs and input modules (3)
- connection field for external outputs and output modules (4)
- connection field for function blocks (5)
- control room modules related with the automation module (6)
- page (7)

The following figure (Figure 14) shows an automation module with the module parts presented in the above list marked in numbers.

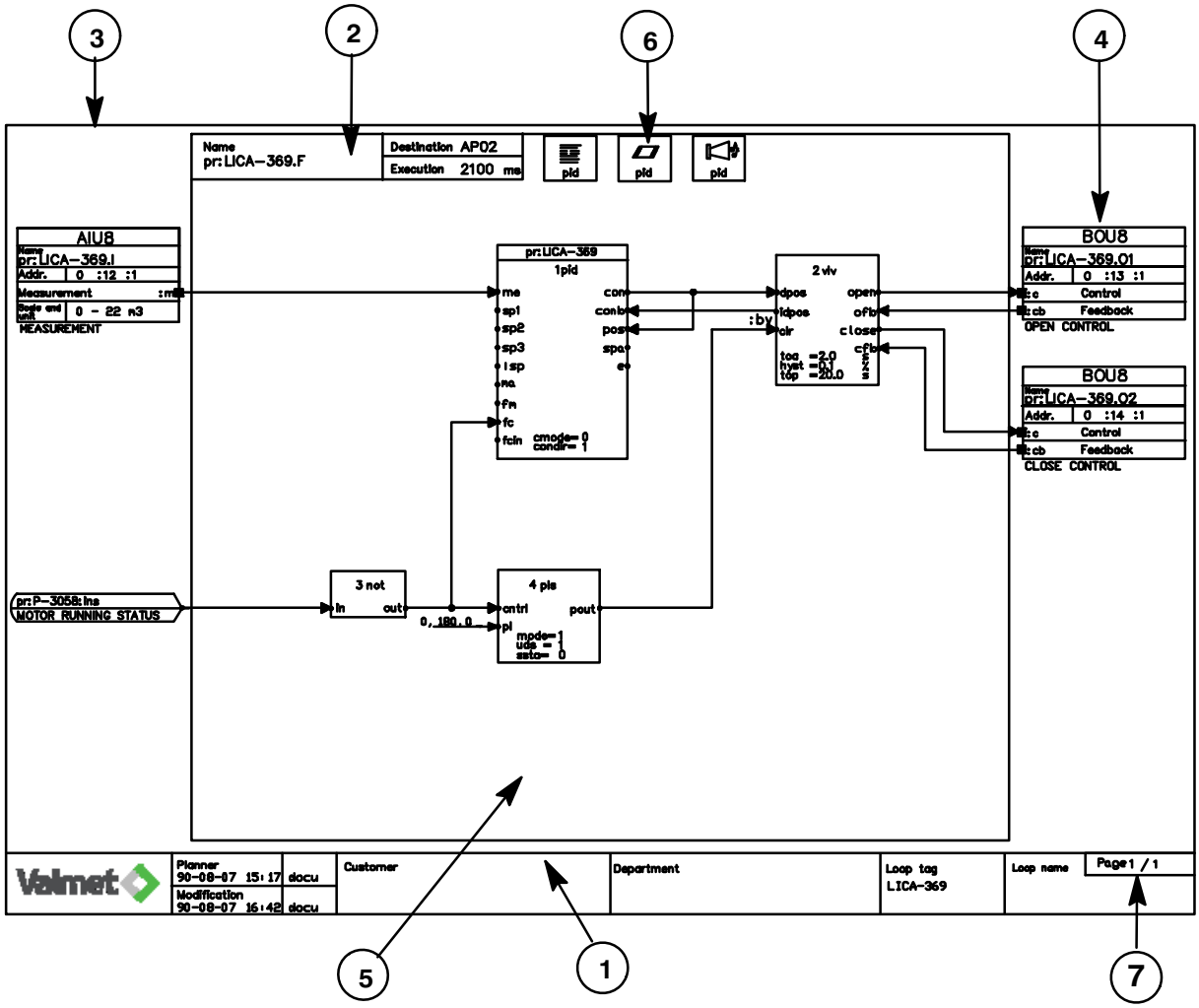


Figure 14 Automation module and its parts

5.2 The Structure of a Configuration Module

In a list form, an automation language configuration module contains three parts:

- administration_part
- representation_part
- functional_part.

5.2.1 Administration Part

The module's administration part contains data used in module and database management.

Following is an example of a module's administration part in a list form:

```
ADMINISTRATION_PART
   NAME:        pr:89LIC-2305.F
   TYPE:        function
   STATUS:      incomplete
   CREATOR:     tim
      CREATED:    89-08-23 15:34
   MODIFIER:    tim
      MODIFIED:   89-08-23 16:14
   DESTINATION: AP02
   EXECUTION:   400
      ORDINAL:    3
   DESCRIPTION:
```

The administration part of a configuration module is presented as a form, containing the following data:

NAME

- The unique name of a module consists of components separated with colons.
- The module naming conventions are described in Chapter 6 "The Automation Language's Naming Conventions".

TYPE

- the following module types are used:
  - **function**
    function modules
  - **io**
    I/O interface modules
  - **tag**
    modules used for tag text definitions
  - **header**
    modules used for generating monitor header texts and time
  - **operation**
    modules used for generating operating windows
  - **activity**
    event printer and monitor definition modules
  - **event**
    modules used for generating event log and alarm area identifiers
  - **registering**
    modules used for generating area identifiers for alarm list, event log and header and for controlling the alarm horn and event printers
  - **keyboard**
    modules used for defining direct selection keys

- **clock**
  modules used to define switching between summer and winter time
- **dio**
  modules used to define the I/O interface of the Damatic interface server
- **picture**
  modules used for generating monitor pictures
- **hierarchy**
  modules used for creating the picture hierarchy
- **path**
  modules used for creating hierarchy path
- **palette**
  modules used for defining colours used in monitors and for hard copies
- **menu**
  modules used for creating the hierarchy menu for operating window
- **sequence**
  modules used for generating sequence programs
- **card**
  PLU and MCP modules

STATUS

- module status
  - incomplete
  - complete
  - tested

CREATOR

- the person who created the module

CREATED

- date and time of creation in the form yy–mm–dd hh:ss

MODIFIER

- person who made the last modification to the module

MODIFIED

- date and time of the last modification in the form yy–mm–dd hh:ss

CREATOR, CREATED, MODIFIER and MODIFIED are updated automatically in the module upon creation or modification.

DESTINATION

- the module's package

  For more detailed information, refer to Chapter 6 "The Automation Language's Naming Conventions".

EXECUTION

- module execution interval in milliseconds, range 200 ms...64000 ms with increments of 100 ms

ORDINAL

- unsigned integer, which defines module execution order in the group of modules to be executed within the same period of time. In control tasks, modules are executed in an ascending order based on the ORDINAL field, i.e., after modules with an ORDINAL field value of 0.

  If several modules have the same ORDINAL field value, such as 0, these modules will be executed in alphabetical order based on the module name.

  In other words, modules are executed in the following order:
  Among modules within the same execution period, the modules whose ORDINAL field value is zero are executed first. If there are more than one modules of this kind, they will be executed in alphabetical order. Next in the order are those modules within the same execution period that have an ORDINAL field value of 1, and so on.

DESCRIPTION

- the engineer's description of the module. The description must be inserted between quotation marks (" ") in configuration modules. In automation modules, the DESCRIPTION is not enclosed in quotation marks. This field can also be left blank.

The module status, creation and modification data are used in module version management and project monitoring. The description contains comments on the destination and function of the module.

5.2.2 Representation Part

The module representation part defines the following things:

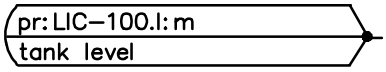
- External data points
  - external data to be connected to the module
- Local data points
  - named local or internal data of the module
- Graphic definition, for example, for picture modules
  - List form graphics are connected to a picture module after the GRAPHICS definitions.
- Direct access ports
  - A module's direct access ports connect with individual data points or refer to an entire function block.
  - Other modules read data from a module through a direct access port.
- Interface ports
  - A module's interface ports connect with individual data points.
  - Other modules read data from a module through an interface port.

Below is an example of a configuration module representation part combined from the examples presented hereafter:

```
REPRESENTATION_PART

  EXTERNALS
    pr:SC-265.I:m TYPE ana TRANSFER 192,10,0,0 ;
    pr:SC-265.oI:m TYPE ana TRANSFER 192,10,0,0 "open limit";
    pr:SC-265.cI:m TYPE ana TRANSFER 192,10,0,0 "close limit";
    pr:SC-265.oO:c TYPE bo TRANSFER 65,10,0,0 "close control";
    pr:SC-265.oO:cb TYPE bin TRANSFER 192,10,0,0 ;
    pr:SC-265.cO:c TYPE bo TRANSFER 65,10,0,0 "open control" ;
    pr:SC-265.cO:cb TYPE bin TRANSFER 192,10,0,0 ;

  LOCALS
    P1 TYPE ana ;

  DIRECT_ACCESS
    BLOCK pr:SC-265 ;

  INTERFACE
    MODSTAT TYPE ktstat "module status" < (1,1,0,1,1) ;
```

External Data Points

In graphic design, an external data point is marked with the following symbol:



Use:

If a module wants a signal from another module, it presents an external input whose type is the same as that of the desired signal's. An external input's communication mode is normally continuous (EXT IN CONTINUOUS).

Different symbols are used in graphic design for different types of external data points. The symbol's colour determines the external data point's type, while the symbol's shape defines the communication mode.

External data points are presented in the configuration module administration part after the keyword EXTERNALS.

The presentation of an external data point includes:

- **name of the external data point**

  The name of the external data point name, i.e. the name with which data is retrieved or transmitted is determined by the module port from which the data is read. The name in itself instructs communication to access the desired data.

  In the following example, we have the name of an external input which is determined by the input module interface port. In this example, the module name is pr:LIC–100.I and its interface port specifier is *m*. The components of the name are separated by colons, the standard separator in the automation language, so the external data point will be

  pr:LIC–100.I:m TYPE ana TRANSFER 192,4,0,0 "level";

- **data type**

  An external data point reserves a data area corresponding to its type, such as an area of the type analog (*ana*).

  In graphic design, the data point's colour determines the data type.

  In a list format, the type is entered after the TYPE keyword in the data point definitions. The following example shows the type entry for an external data point whose type is *ana*:

  pr:LIC–100.I:m **TYPE ana** TRANSFER 192,4,0,0 "level";

- **possible initial value of the data**

  A data point can be initialized, so it will have a value even if it has not yet been updated by communication.

  Initialization is accomplished by writing the "=" character after the type code and writing appropriate initial values for the type inside parentheses. Example:

  pr:LIC–100.I:m TYPE ana=**(0,0.5)** TRANSFER 192,4,0,0 "level";

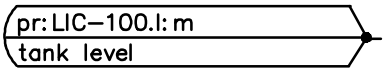
  In graphic design, parentheses are not used to give initial values for a data point.

  If an initial value is not specified, the data point is assigned the default value corresponding to its type when the module goes to the application server. The initial value will be valid until it is changed by a connection or communications.

- **definition of data transfer mode**

  Different types of external data points can be defined for different communication needs. The data transfer mode refers to whether the external data point is an input or output or both, and whether the data point communicates with others either continuously or only when data has changed.

  Continuous external input

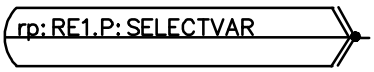
  

  Continuous external output

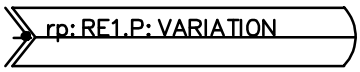
  

  Continuous communication is the most common mode. In this mode, data is transferred continuously between data points. If there is a lot of data to be transferred and continuous updating of the data is not required, it is also possible to use conditional communication.

  Conditional external input

  Conditional external output

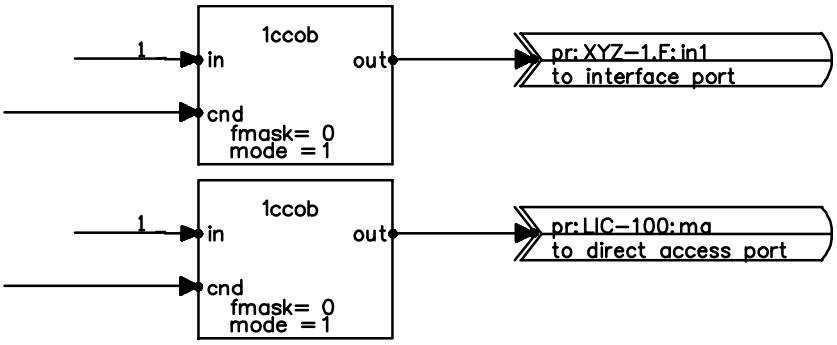
  

  A conditional external output can only be used in connection with conditional copy function blocks (the ccoX blocks).

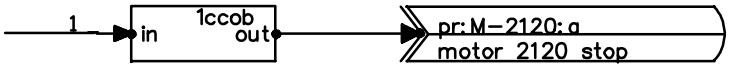
  A conditional output is executed always when the related ccoX function block's copying condition is in a state where the block is writing something to its output.

  Conditional outputs are used especially in sequences and recipes since they do not normally require continuous communication: the operation is achieved with conditional communication that loads the network less.

NOTE!
A conditional external output can only be used in connection with ccoX function blocks.

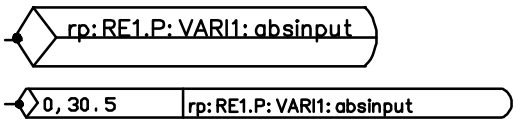


NOTE!
A conditional external output must not be used in the manner of a continuous output if the module is in continuous execution (e.g. a sequence step), because this would load the traffic too much.



Continuous input and output

If the communication mode is continuous input and output, communication is possible in both directions between data points. This communication mode is used, for instance, between the variation modules and parameter modules of recipes.



In high level language design, the data transfer is expressed with a group of four numbers which is encoded as follows:

The first number indicates the direction of data transfer

128 = retrieve data (input)
1 = transmit data (output)

and the nature of data transfer

+64 = continuous
+32 = conditional
+16 = event
+2 = direct addressing

For instance, if data is retrieved on a continuous basis, the first number will be formed as follows: 128 + 64 = 192.

The second number defines the data transfer interval in hundreds of milliseconds. The number may have the values 0...255. Zero defines a single transmission. The value 1 corresponds to 100 ms update interval; thus the maximum value 255 corresponds to 25.5 second data transfer interval.

The third number is zero.

The fourth number indicates the update mode

7 = event 1 –> 0 or 0 –> 1
6 = event 0 –> 1
5 = event 1 –> 0

If data transfer is continuous, the value of the fourth number is irrelevant; a zero will do.

In the following, the above signal is completed with the update mode:

pr:LIC–100.I:m TYPE ana **TRANSFER 192,4,0,0** "level"

The symbol used in graphic design determines the data transfer mode.

- **possible comment**

  The comment can also be omitted. In list format design, the comment is placed between quotation marks (" "). Quotation marks are not used in graphic design.

  pr:LIC–100.I:m TYPE ana TRANSFER 192,4,0,0 **"level"**;

- **data point definition is ended with a semicolon** (;)

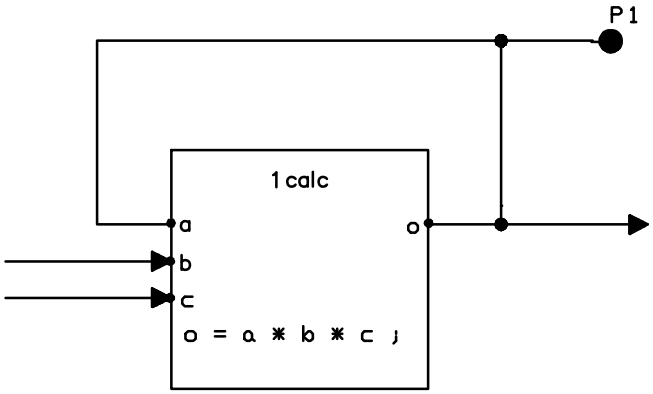
Local Data Points

In graphic design, the following symbol is used for a local data point:

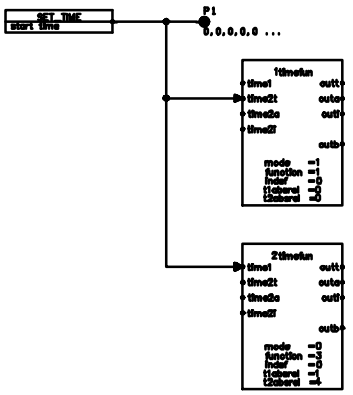
Use:

The need to use a local data point is rather rare in graphic design. In most cases, the CAD tool will itself create the local data points in the configuration module generated from the graphics. (Local data points are only shown in the configuration module, not in the graphic image.)

However, a local data point can be used e.g. in linking a signal



and in assigning an initial value:



Local data points are presented after the LOCALS keyword in a configuration module's RE-PRESENTATION PART.

The presentation of a local data point includes:

- **name**

  The name of a local data point must begin with a letter. The rest of the name can include letters, numbers, period (.), and underline character (_). An example of a local data point name: **P1**

- **possible dimension**

  If the local data point is an array, also dimensions and index limits are defined.

  The dimension is defined with two integers inserted in parentheses. In this example, quantity is a 2–dimensional array whose index limits are 2 and 3. A local data point reserves an area corresponding to its type and dimensions in the data space of the module. An example of a local array–type data point:

  quantity **(2,3)** TYPE float=(0,2.1,0,5.0,1,40.86) "2 dimensional array"**;**

  In graphic design, the dimension is given as a number without parentheses.

- **data type**

  A local data point reserves a data area corresponding to its type, such as a *float* type area. The type is marked in the data point definition after the keyword TYPE. Following is an example of expressing a *float* type external data point type:

  P1 **TYPE ana**=(0,2.1) "local data point";

  In graphic design, the data point's colour defines the data point's type.

- **possible initial value of the data**

  A data point can be initialized so that it will have a value even if it has not yet been updated by communications. Initialization is executed by entering the "=" character after the type code and by giving initial values suitable for the type inside parentheses. For example:

  P1 TYPE ana**=(0,2.1)** "local data point";

  In graphic design, an initial value is given as a number without parentheses and the "=" character.

  Initial value can also be omitted. If initial values are not specified, the data point will be given default values corresponding to its type when the module enters the application server. The initial values will remain constant unless they are changed through connection or communications.

- **possible comment**

  Comment can also be omitted. The comment is inserted between " " characters as follows:

  P1 TYPE ana=(0,2.1) **"local data point"**;

  In graphic design, a comment is not placed between quotation marks.

- **data point definition ends with a semicolon (;)**

Adding Graphics to a Module

In graphic design, the user does not have to define the graphic picture in list format. Instead, the generation utilities will automatically create a listing from the graphics.

In list format presentation of graphics, the drawing commands are given in the section following the word GRAPHICS. Before the drawing commands, you give the name of the picture after DEFINE PICTURE.

The picture template presented in the representation part is connected to the module through a function block using the name of the representation part.

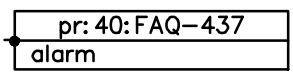
An example of adding list format graphics to a picture module:

```
REPRESENTATION_PART
   EXTERNALS
      .
      .
      .
   GRAPHICS
      DEFINE PICTURE pic1
         PELSIZE 1 1
         COLOR 0
         MOVE A 0 638
         LINE R 300 20
            .
            .
      STOP PICTURE
      ;

FUNCTIONAL_PART
      .
      .
   1draw
      nap < pic1
   ;
      .
      .
END
```

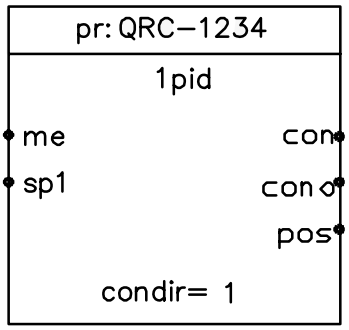
Refer to Appendix 3 for more information on the automation language's list format graphic commands and their use.

Direct Access Ports

In graphic design, direct access ports are marked with the following symbols:



direct access port connected to one point



direct access port connected to entire function block

Direct access ports are presented in the configuration module REPRESENTATION PART after the keyword DIRECT_ACCESS.

The presentation of a direct access port includes:

- **port name**

  Port name usually refers to a module name, however without a type specifier.

  Example:

  pr:XI–101 TYPE ana ”comment” < (0,100.0);

- **port type**

  A port type is defined as the signal type to be connected to the port. A port connected to a function block is not assigned a type. In this case, the port definition starts with the word BLOCK, followed by the port name alone.

  Example:

  pr:XI–101 **TYPE ana** ”comment” < (0,100.0);

- **possible comment**

  The definition of a port may include comment text, which is separated from the rest of the entries with quotation marks (””).

  In graphic design, the quotation marks are not used in a comment text.

  Example:

  pr:XI–101 TYPE ana **”comment”** < (0,100.0);

- **possible port connection** either to a local or an external data point or reference to a function block member

  If the port is connected either to a local or an external data point or function block member, the port connection is marked after it. The connection is defined by the ”<” character.

  Example:

  pr:XI–101 TYPE ana ”comment” **< (0,100.0)**;

  The definition of a port connected to an entire function block starts with the word BLOCK followed by the port name alone.

- **the definition of a direct access port ends with a semicolon** (;)

The complete definition of a direct access port:

```
REPRESENTATION_PART
.
.
  DIRECT_ACCESS
    pr:XI-101 TYPE ana < (0,100.0);
    BLOCK pr:LI-100;
.
.
FUNCTIONAL_PART

   1pid IS pr:LI-100
    pu= -
```

In the examples, the first direct access port pr:XI–101 is connected to the first element of a 2–dimensional array ”quantity”. Character ”<” indicates the connection.

A direct access port can also be defined in shorter format:

```
pr:XI-101 TYPE ana< - ;
```

In this case, the signal to be connected to the port is defined in the signal itself.

The second direct access port pr:LI–100 is presented as a port connected to an entire function block. The functional part (1pid IS pr:LI–100) defines the function block that the port refers to. All members of the *1pid* function block are thus made visible to other modules through the direct access port.

Use:

A direct access port referring to an entire function block can be connected to the following function blocks:

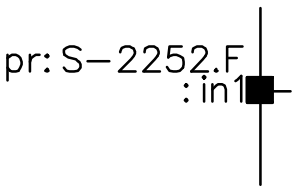
- am
- pid
- qcnt
- cnt
- mtr
- mtre
- mgv
- mgve

In general, all loops/signals shown in the control room are connected to a direct access port.

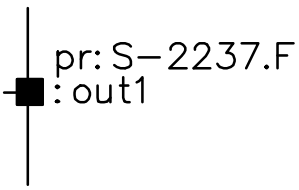
Typically, the ”source module” has a direct access port while the destination modules present an external input requesting the desired signal from the port.

Interface Ports

In graphic design, an interface port is marked with the following symbol:



input interface port



output interface port

Use:

Signals to be transferred to other modules are defined as interface ports. The requester of the data presents an external input asking for the desired signal (pr:NAME.F:interface port specifier).

In an application program listing (in its REPRESENTATION PART), interface ports are presented after the keyword INTERFACE.

An interface port presentation includes:

- **port name**

  The normal limitations in automation language also apply to the naming of ports. In general, the output ports of function modules are named *out* and identified with a number. For example, out1, out2. The input ports of a function module are named *in* followed by a number such as in1, in2. (In graphic design, the port name is preceded by a colon.)

  Interface ports can also be named using names derived from module names.

  out1 TYPE bin ”comment” < P1;

The following names are recommended in naming ports:

In input modules:

m = measurement

In output modules:

c = control
cb = feedback

Examples of entering port specifiers:

```
pr:LIC-100.I:m    (measurement)
pr:50-HS-115:c    (control)
pr:50-P-125:cb    (feedback)
```

In graphic design, the naming of ports occurs ”automatically”.

- **port type**

  Port type is defined as the type of the signal to be connected.

  In graphic design, the symbol’s colour defines the port type.

  out1 **TYPE bin** ”comment” < P1;

- **possible comment**

  A port definition can contain comment text, which is separated from the rest of the port definition by quotation marks (””).

  In graphic design, the quotation marks are omitted from a comment text.

  out1 TYPE bin **”comment”** < P1;

- **possible port connection** to local or external data point, function block, or constant

  out1 TYPE bin ”comment” **< P1**;

- **interface port definition ends with a semicolon** (;)

  out1 TYPE bin ”comment” < 5ccob:out**;**

In the above examples, the interface port *out1* is connected to the output *out* of the *5ccob* function block in a module.

The definition can also be given in shorter format:

out1 TYPE bin < –;

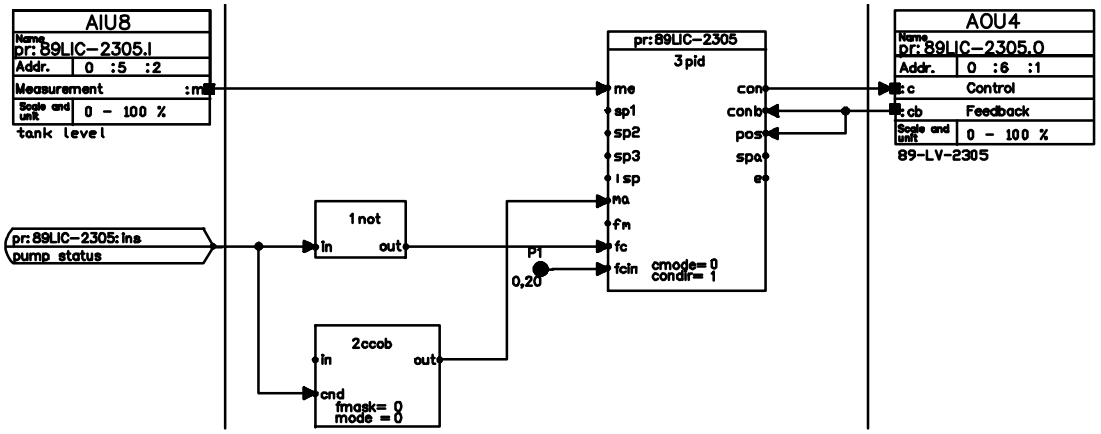
In this case, the signal to be connected to the port is defined in the signal itself.

5.2.3 Functional Part

A module’s functional part consists of function blocks. The desired module operation is configured by combining the function blocks and the data points and ports defined for them in the REPRESENTATION PART.

A module’s functional part consists of function blocks that can be provided with parameters and interconnected. The function blocks in a display module are usually independent and are only provided with parameters. A function module, on the other hand, may have a large number of connections.

The following figure shows a part of an automation module and the corresponding configuration module's functional part in list format.




```
FUNCTIONAL_PART

   1not
    in< pr:89P-2305:ins
    out> -
   ;

   2ccob
    mode= ( 1 )
    fmask= ( 0 )
    cnd< pr:89P-2305:ins
    in< ( 1 )
    out> -
   ;

   3pid ON pr:89LIC-2305
    pu= -
    squ= -
    iu= -
    deu= -
    ffu= -
    parx= -
    cmode= ( 0 )
    condir= ( 1 )
    aftfm= ( 0 )
    aftfc= ( 1 )
    fbact= ( 2 )
    fbmask= ( 12 )
    memi= ( 0.0 )
    mema= ( 100.0 )
    ffmi= -
    ffma= -
    comi= ( 0.0 )
    coma= ( 100.0 )
    bias= -
    conch= ( 1.0 )
    slow= -
    sp2u= ( 0 )
    sp3u= ( 0 )
    track1= ( 0 )
    track2= ( 0 )
    track3= ( 1 )
    cha1= -
    cha2= -
    cha3= -
    mau= -
    eau= ( 0 )
    coau= ( 0 )
    ahys= -
    kp< ( 0,0.75 )
    ti< ( 0,30.0 )
    td< -
    tdf< -
    kff< -
    me< pr:89LIC-2305.I:m
    mff< -
    sp1< -
    sp2< -
    sp3< -
    conb< pr:LIC-2305.O:cb
    colmi< ( 0,0.0 )
    colma< ( 0,100.0 )
    isp< -
    ma< 2ccob:out
    parch< -
    amc< ( 1 )
    mac< ( 1 )
    ion< -
    fm< -
    fc< 1not:out
    fcin< P1
    pos< pr:89LIC-2305.O:cb
    mehh< ( 100.0 )
    meh< ( 100.0 )
    mel< ( 0.0 )
    mell< ( 0.0 )
   ;
```

In the functional part, each function block is presented in a separate section. Function blocks are numbered consecutively (*1not*, *2ccob* and *3pid*). The operation of a function block can be explained in the list format by writing a comment inserted in quotation marks after the function block identifier and name, example: 3pid ”Controller”.

The operation of the function block *2ccob* in the example can be altered by the configuration parameters *mode* and *fmask*, to which have been given values after the ”=” in the list format. In this function block, only *mode* is initialized, so *fmask* will have the default specified in the general description of a *ccob* function block.

Function block connections are defined with characters ”<” (input) and ”>” (output) to members pointed to by the characters. In the above example, the external input pr:89LIC–2305:ins is connected to the input *in* of the function block *1not* (`in< pr:89P–2305:ins`).

The output *out* of the same function block is left unconnected, so it is followed by character ”–”. (If a function block template is used for making the listing, the members already have the ”–” characters.) A signal can be left undefined when the signal to be connected is defined at the other ”end” of the signal, i.e. in *fc* of *3pid* (`fc< 1not:out`).

The input *in* of the function block *2ccob* is connected to the constant 1, which is presented in the listing as follows: `in< ( 1 )`.

In graphic design, function block parameters are requested in clear text from the user and signal connections are made by connecting points with lines. A generator converts the graphic picture to list form configuration modules, so the above structures describing signal connections are formed automatically.

In configuring list form, modules help is available, for example, on editor commands and types (function blocks) to be used. The function block help texts (function block templates) can be picked to the module to be designed, so all members of the function block will be visible. A function block member must be initialized only if you want to change its default initial values. Likewise, only the members that are relevant in the operation of the module need to be connected.

A function block definition ends with a semicolon.

You should notice the following when defining a function block:

- Configuration parameters cannot be connected; you can only give numerical values for them. This is represented by the character ”=” in connection with the parameters. You should then notice, for instance, that the tuning parameters of a *pid* controller are not configuration parameters but inputs from the point of view of the pid function block. Thus they can be controlled e.g. through another function block.
- Only those configuration parameters (=), inputs (<) and outputs (>) to which you want to enter data must be presented. Others will have the default values specified in the function block definition.
- Inputs or outputs can also be connected with a constant, which is written inside parentheses. The parentheses are needed, for instance, to distinguish the constant (578.051) from the external data 578.051. Both can be connected to a function block. (In graphic design, it is not necessary to insert constants in parentheses.)

  (The use of calculation, logic and comparison function blocks differs slightly from the use of other function blocks.)

Calculation

The application programmer defines calculation algorithmically by entering the mathematical expressions directly in their normal format. He can utilize operator priorities, parentheses, etc.

Calculation can be performed on the following types of variables:

The functions include the following:

- *ana* analog variable
- *ints* 16 bit integer with fault bits
- *intl* 32 bit integer with fault bits.

The following operators can be used (listed by descending priority):

- *,/ = multiplication and division
- +,– = addition and subtraction

Calculations can also be grouped using parentheses, which have the highest priority.

Functions are:

- SIN = sine function
- EXP = e–base exponent
- LN = e–base logarithm
- SQRT = square root
- ABS = absolute value

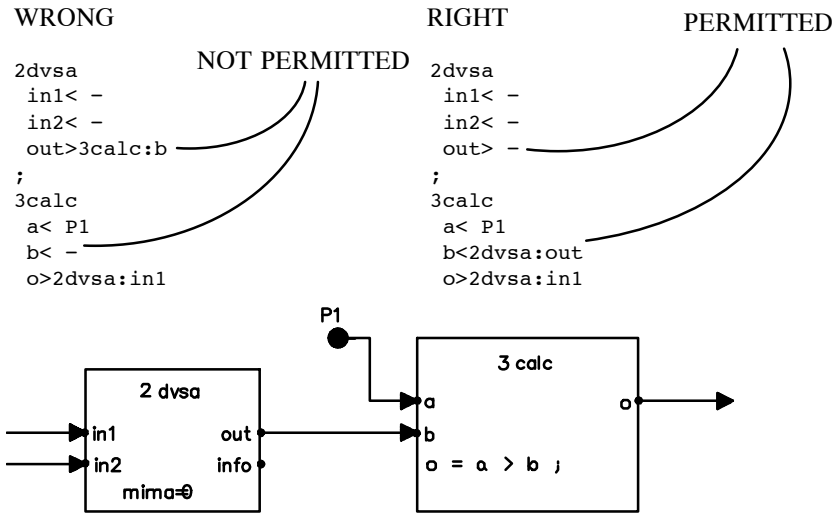
Calculation is defined as part of the function block list of the configuration module, as shown in the following example.

```
REPRESENTATION_PART

   EXTERNALS
      pr:F-129.I:m TYPE ana = (0,0.5) TRANSFER 192,4,0,0 ;

   LOCALS
      P1 TYPE ana = (0,0.6000) ;

   DIRECT_ACCESS
      pr:FF-128 TYPE ana < - ;

   INTERFACE
      out1 TYPE ana < - ;
      MODSTAT TYPE ktstat < (1,1,0,1,1);

FUNCTIONAL_PART
   CALCULATE 1calc
      CONNECT
         a TYPE ana < pr:FF-128 ;
         b TYPE ana < F-129.I:m ;
         c TYPE ana < P1 ;
         o TYPE ana > out1 ;
      FORMULAS
         o=a*b/c;
   STOP 1calc
.
.
END
```

As you see, calculation is defined in much the same way as function blocks. The application programmer only enters the identifiers of the connections and gives their types, because there is no predefined type specification as for function blocks in general.

There are some more limitations in the connection of calculation, logic and comparison function blocks than in the case of ordinary function blocks: the connection cannot be made from outside, in other words, their members cannot be marked with ”–” which would allow defining the connection in another function block. The connections of these function blocks must be defined in the same function block or else a local data point must be used for the connection.

The following example illustrates the wrong and the right way of making a connection:

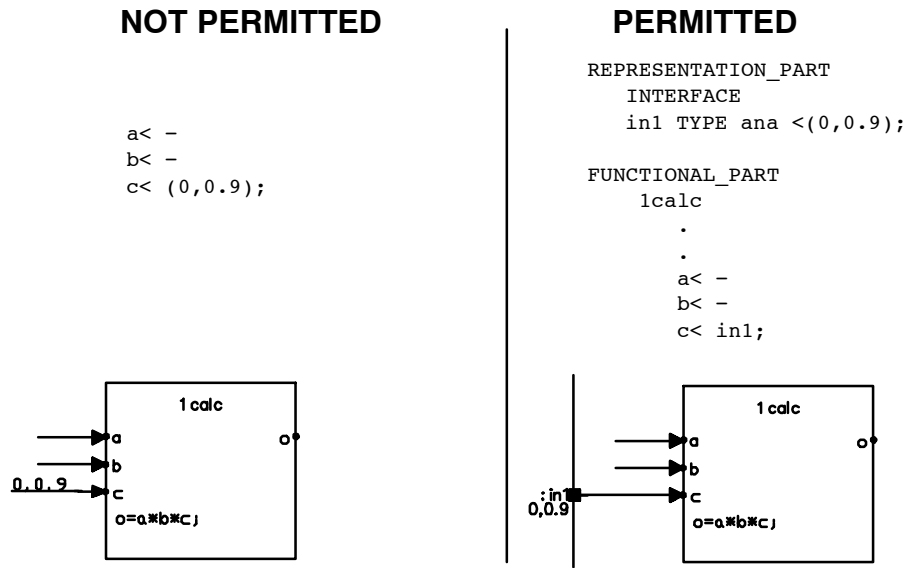


In graphic design, the compilation utilities create the permissible connections automatically.

Another limitation in the calculation, logic and comparison function blocks is that you cannot connect constants directly to their members. Instead, such connection has to be performed through a local data point or an interface/direct access port. You initialize the port or data point with the value that you want to connect to the function block, and connect the port to the function block.

A constant can be entered directly in a formula (e.g. o=a*b*100.0). However, the value of the constant cannot then be changed, for instance, with debugger.

If you want to adjust the value of a constant with the debugger, it is recommendable to connect the constant to an interface or direct access port and initialize the port with the desired constant.



The third limitation in calculation, logic and comparison function blocks is that they cannot be connected directly together. If there is a need to connect these function blocks together, the connections are made via local data points.

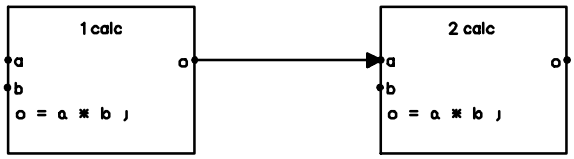
In graphic design, the local data point is generated automatically in the configuration module.

NOT PERMITTED

```
FUNCTIONAL_PART
  1calc
    o>2calc:c ;
```

PERMITTED

```
REPRESENTATION_PART
   LOCALS
     P1 TYPE ana =(0,0.9);

FUNCTIONAL_PART
    1calc
      o>P1;

     2calc
       c<P1;
```



Comparison

Comparison is defined in much the same way as calculation, but the inputs can be of the types *ana*, *ints* or *intl*, and the output is of the type *bin*.

The following comparison operators can be used for comparisons:

- **>=** greater than or equal to
- **<=** less than or equal to
- **==** equal to
- **!=** not equal to
- **>** greater than
- **<** less than

In addition, the following logic operators are available between *bin* signals generated inside a *cmp* function block:

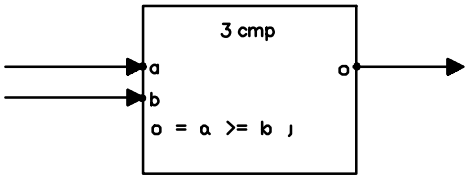
- **OR** logic or
- **XOR** logic exclusive or
- **AND** logic and
- **NOT** negation

Comparisons can be grouped by parentheses.

Like calculations, comparisons are defined as part of the function block list of the configuration module.

```
FUNCTIONAL_PART
.
.
   COMPARE 3cmp
    CONNECT
         a TYPE ana <1pid:con;
         b TYPE ana <2pid:con;
         o TYPE bin >out2;
    FORMULAS o = a >= b;
   STOP 3cmp
.
.
END
```

The corresponding example in graphic format:



Logics

Logics are defined in much the same way as calculation and comparison, but the type of the inputs and outputs is *bin*.

The following logical operators are available for creating logics:

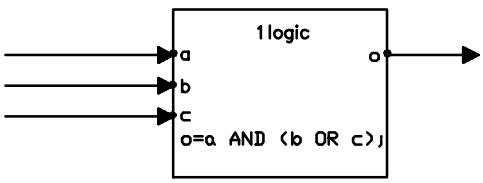
- **OR** logic or
- **XOR** logic exclusive or
- **AND** logic and
- **NOT** negation
- **SR(s,r,i)** and **RS(r,s,i)** SR and RS flip–flops

Logic operations can be grouped by parentheses.

Like calculation and comparison, logics are defined as a part of the function block list of a configuration module.

```
FUNCTIONAL_PART
.
.
   LOGIC 1logic
     CONNECT
           a TYPE bin <pr:507.102.I1:m;
           b TYPE bin <2pid:ma;
           c TYPE bin <in1;
           o TYPE bin >out1;
     FORMULAS o = a AND (b OR c);
   STOP 1logic
.
.
.
END
```

The corresponding example in graphic format:



5.3 Internal Connections in a Module

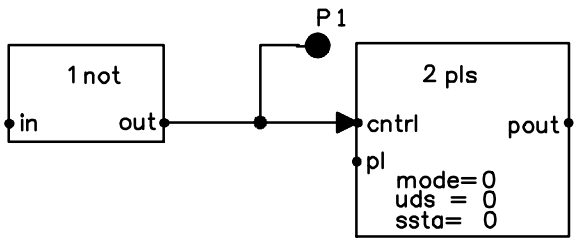
Data transfer inside a module is based on connection. Connection is established when two or more connection points refer to the same data point. The connection point and the associated data point, or part of data point separated with an index or specifier, must be of the same type.

Connections between function blocks can be made by

- defining a local data point and connecting the connection points of function block to it, or
- directly at a connection point of a function block by writing here the function block to be connected and its connection point. In this case, the connection is made through an automatically generated data point that is invisible to the user. If a value is written at the connection, a connection to a data point is generated, and the data point is initialized with this value.

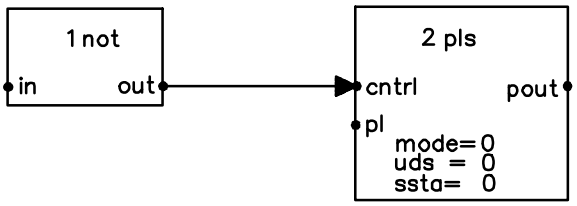
These methods are illustrated by the following two examples.

Example 1: Connection via a separately defined local data point




```
REPRESENTATION_PART
.
.
   LOCALS
      P1 TYPE ana;
.
.
FUNCTIONAL_PART
   1not
    in< -
    out> P1
   ;
   2pls
    cntrl< P1
    pout> -
   ;
.
.
```

Example 2: Direct connection between the connection points of function blocks

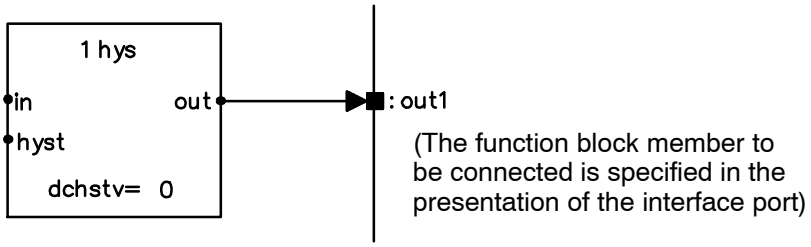



```
FUNCTIONAL_PART
   1not
    in< -
    out> -
   ;
   2pls
    cntrl< 1not:out
    pout> -
   ;
   .
   .
```

These examples primarily refer to the list format design. Only the method shown in Example 2 is used in graphic design.

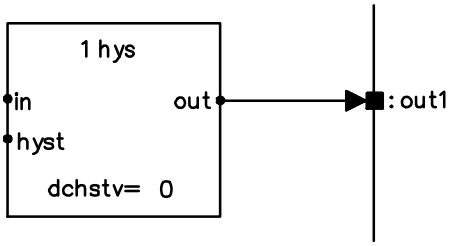
Correspondingly, **connections between function blocks and ports** are illustrated by the following examples:

Example 3: Connection is made directly between a port and a function block. The function block to be connected and its connection point are indicated in the presentation of the interface port.




```
REPRESENTATION_PART
.
.
   INTERFACE
      out1 TYPE ana< 1hys:out;
.
.
FUNCTIONAL_PART
   1hys
    in< -
    out> -
   ;
```

Example 4: Connection is made directly between the port and the function block. The interface port to be connected is written at the connection point of the function block.



(The interface port to be connected is defined at the function block connection point)

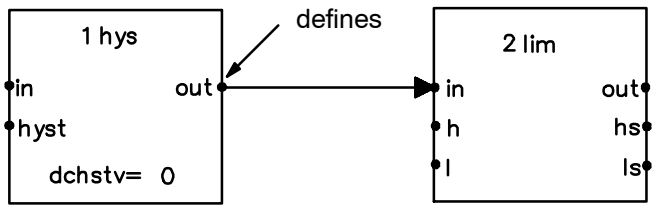
```
REPRESENTATION_PART
.
.
   INTERFACE
      out1 TYPE ana< -;
.
.
FUNCTIONAL_PART
   1hys
    in< -
    out> out1
   ;
```

The compiler tool of the configuration software converts the connections to mode 4.

When a connection is made to a connection point to which another connection point has been connected, it is possible to use the path to the other connection point. The path consists of the function block identifier and a specifier indicating the connection point. The identifier path enables connection to connection points in a function block that have not yet been defined; the actual connection is made later when defining the function block.

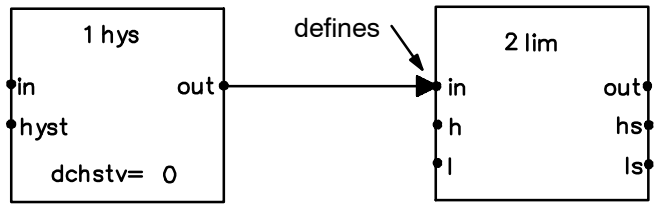
In the following examples, the same connection is made in three different ways by means of an identifier path.

Mode 1




```
.
.
   1hys
    out> 2lim:in
   ;
   2lim
    in< -
   ;
.
.
```

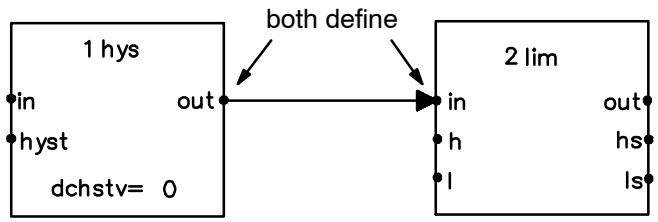
Mode 2




```
.
   1hys
    out> -
   ;
   2lim
    in< 1hys:out
   ;
.
.
```

The compiler tool of the configuration software converts the connections to mode 2 (not in the case of *calc*, *cmp* or *logic* function blocks, because connection to their members cannot be made from outside the function block).

Mode 3




```
.
.
   1hys
    out> 2lim:in
   ;
   2lim
    in< 1hys:out
   ;
.
.
```

In the case of a structured signal, it is possible to retrieve either the entire signal or its member. A member can be accessed by adding a specifier to the identifier of the signal to be connected.

Following is an example of a data point specifier used for extracting fault bits from an ana type signal:

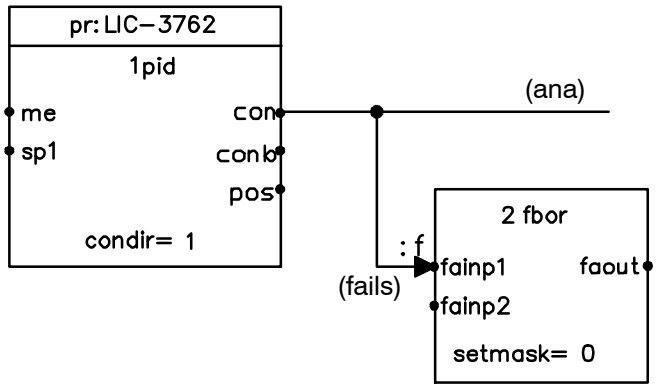


Figure 15 Example of a connection point specifier

The output signal of the PID controller *1pid* is of the type *ana*, and contains the fault data *f* and the signal value *a*. To extract the fault bits from the *ana* type signal the signal is added with the fault data identifier *f* as follows: 1pid:con:**f**.

The corresponding part of the module in list form

```
FUNCTIONAL_PART
.
.
.

  2fbor
   fainp1< 1pid:con:f
.
.
.
```

A signal specifier is usually added to the respective input of the function block that uses the signal.

5.4 Communication Between Modules

Communication between modules occurs through a pair consisting of a port and external data point. When a module needs data from another module, it presents the external data point where the data is to be copied from the other module through its own port. The identifier of the external data point defines the module from which the data will be copied, the port through which the data will be copied, and the data that will be copied.

To transfer all data of a certain connection point of a function block to a module port, the port can be defined as a block tied to this function block. Now all connection points of this function block can be accessed via the defined block.

As we noted, communication between modules is based on copying data. If the data requested from a module is structured and thus consists of several members, other modules can request the following parts of this structured data:

1. Entire data, i.e. all members
2. Named member
3. A group of selected members with a specific **viewpoint** (control room)

5.4.1 Using a Direct Access Port in Communication

The following figure (Figure 16) shows how two modules can communicate with each other using a direct access port and an external data point.

In module pr:50–QC–136.F, the entire PID controller is connected to the direct access port pr:50–QC–136. Now all members of the controller are accessible to other modules through their own external data points (e.g external data points pr:50–QC–136:me and pr:50–QC–136:sp1 in module pr:50–QC–222.F).

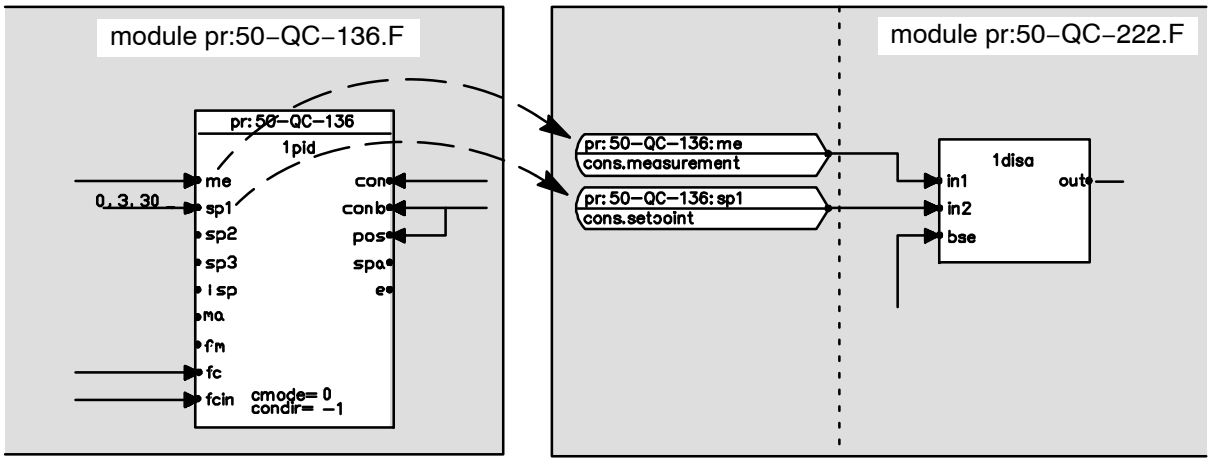


Figure 16 *Module communication from a direct access port to an external data point*

For communication to be possible, the list format module pr:50–QC–136.F must include the following definitions:

```
REPRESENTATION_PART
   DIRECT_ACCESS
      BLOCK pr:50-QC-136

FUNCTIONAL_PART
   1pid is pr:50-QC-136
    .
    .
```

- The representation part defines a direct access port named pr:50–QC–136, which refers to a function block.
- The functional part defines *1pid* controller as a direct access port pr:50–QC–136.

For communication to be possible, the list format module pr:50–QC–222.F must include the following definitions:

```
REPRESENTATION_PART
   EXTERNALS
      pr:50-QC-136:me TYPE ana TRANSFER 192,10,0,0;
      pr:50-QC-136:sp1 TYPE ana TRANSFER 192,10,0,0;

FUNCTIONAL_PART
   1disa
      .
      .
    in1< pr:50-QC-136:me
    in2< pr:50-QC-136:sp1
      .
      .
```

- These presented external data points make it possible to transfer data external to the module inside the module. The first external data point (pr:50–QC–136:me) is the *me* member of the *pid* function block whose data is connected to input *in1* in the *1disa* function block. The second external data point (pr:50–QC–136:sp1) takes the *sp1* member of the *pid* function block and connects it to the *in2* member of the *1disa* function block.

An external data point can also be used for transferring data to the other direction, to ports. The direction of data transfer and other specifications concerning the data transfer are specified in the transfer definition of the external data point. A transfer may be prompted by a change, or it can be cyclical or conditional.

This procedure generally applies only to sequences (in certain actions) and for writing data to output cards.

5.4.2 Using an Interface Port in Communication

The following example illustrates communication between modules using interface ports and external data points.

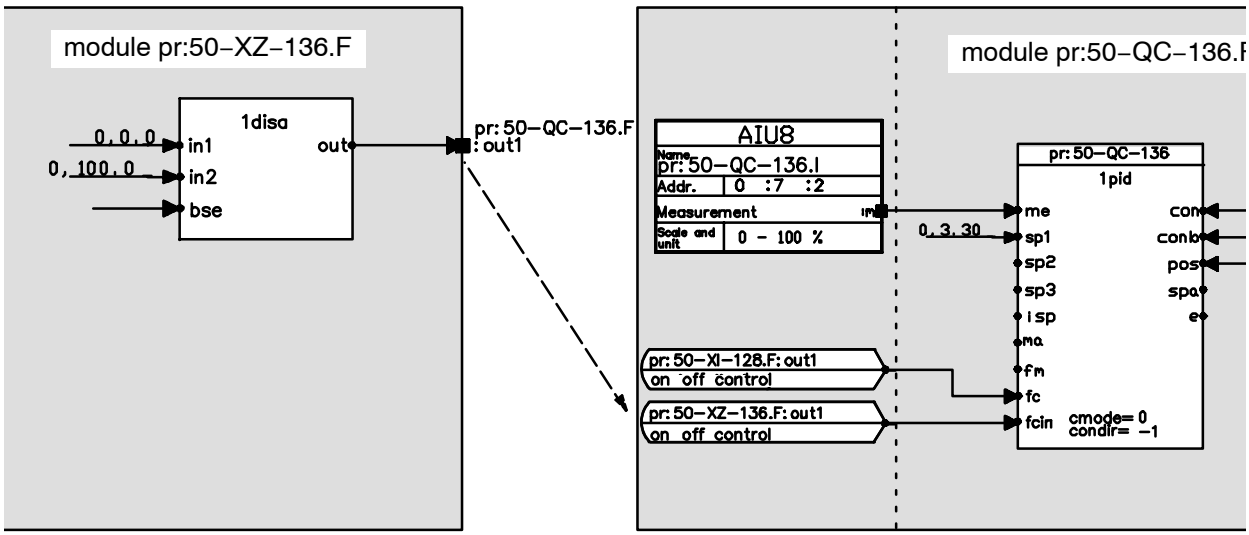


Figure 17 *Module communications from an interface port to an external data point*

For communication to be possible, the list format module pr:50–XZ–136.F must include the following interface port:

```
REPRESENTATION_PART
   INTERFACE
      out1 TYPE ana <1disa:out
      MODSTAT TYPE ktstat < (1,1,0,1,1);
```

- The representation part presents the *ana* type interface port *out1* and its connection to the *out* member of the *1disa* function block. In this way, member *out* in function block *1disa* is brought from inside the module to its interface and made accessible to other modules.

The list format module pr:50–QC–136.F must have an external data point to enable the use of the data from another module.

```
REPRESENTATION_PART
   EXTERNALS
      pr:50-XZ-136.F:out1 TYPE ana TRANSFER 192,10,0,0;
```

- The name of the external data point consists of the source configuration module's name pr:50–XZ–136.F extended with interface port name *out1* separated by a colon. In the data transfer definition the external data point is defined as a continuous input.

  Using external data inside a module is possible in the following way:

```
FUNCTIONAL_PART
   1pid
    .
    .
    .
    fcin< pr:50-XZ-136.F:out1
    .
    .
    .
```

5.4.3 Using a Viewpoint in Communications

Modules can request structured data from other modules on the basis of viewpoints. Data requested on the basis of a viewpoint consists only of specific members defined by the viewpoint. Data requested on the basis of a viewpoint is transferred to the other module's external data point whose type is compatible with the viewpoint type. By using viewpoints, you can extract the data actually needed in each particular situation. The control room is a typical user of viewpoints.

The following example shows how data can be transferred from a process control server.

Example: Process control server function module pr:LI–700.F :

```
ADMINISTRATION_PART
   NAME:         pr:LI-700.F
   TYPE:         function
   STATUS:       incomplete
   CREATOR:      simon
      CREATED:      89-06-13 09:32
   MODIFIER:     simon
      MODIFIED:     89-06-20 11:16
   DESTINATION:  AP01
   EXECUTION:    400
      ORDINAL:      3
   DESCRIPTION:  "test module"

REPRESENTATION_PART
   EXTERNALS
      pr:LI-700.I:m TYPE ana TRANSFER 192,4,0,0;
   DIRECT_ACCESS
      BLOCK pr:LI-700
   INTERFACE
      MODSTAT TYPE ktstat < (1,1,0,1,1);

FUNCTIONAL_PART
      1am IS pr:LI-700
       hyst= ( 4 )
       un= -
       av< pr:LI-700.I:m
       hh< -
       h< -
       l< -
       ll< -
       out> -
       hha> -
       ha> -
       la> -
       lla> -
       fa> -
      ;

END
```

Direct access port pr:LIC–700, which refers to the module's only function block (*1am*), is defined in the module. Other modules in Valmet DNA may request data from the module pr:LI–700.F to their external data points in three different ways:

Mode 1:
Requesting data of the direct access port, in other words, function block members, using a *bundle port* (the example has an *am.opb* type bundle port, so the following members can be requested: av, hh, h, l, ll, out).

```
REPRESENTATION_PART
   EXTERNALS
      pr:LI-700 TYPE am.opb TRANSFER 192,4,0,0;
      .
      .
```

Mode 2:
Requesting the **named member** *out* of the direct access port.

```
REPRESENTATION_PART
   EXTERNALS
      pr:LI−700:out TYPE ana TRANSFER 192,4,0,0;
       .
       .
```

Mode 3:
Requesting direct access port data using the viewpoint *pgd* (= graphic display viewpoint) which selects members *av* and *out* of the *am* function block (other viewpoints of the *am* function block include *pt* and *op*).

```
REPRESENTATION_PART
   EXTERNALS
      pr:LI−700:pgd TYPE am_pgd TRANSFER 192,10,0,0;
       .
       .
```

5.5 An Example of the Structure, Connections and Communications of a Configuration Module

In this example, we examine automation module XZ−108 (Figure 18).

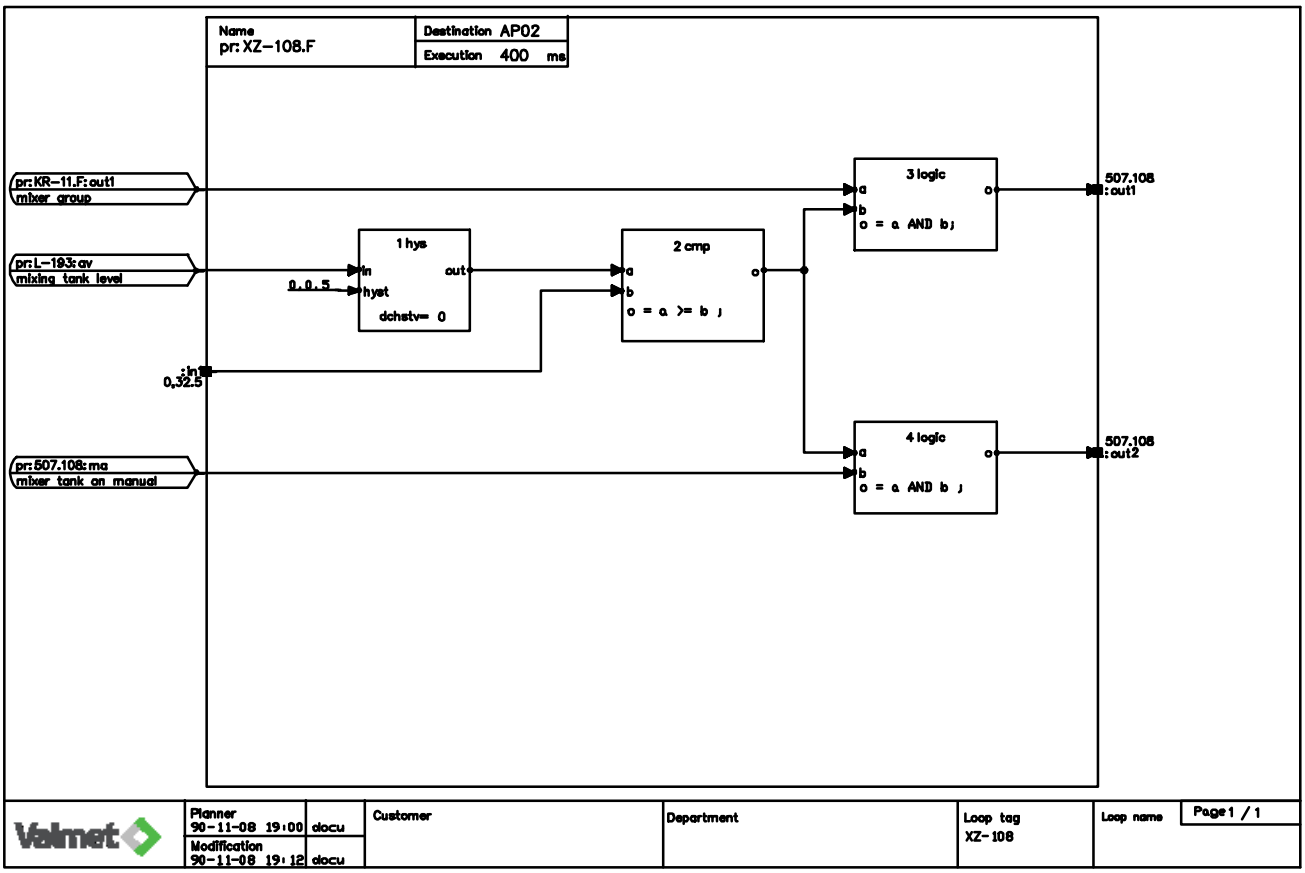


Figure 18 Module XZ−108

pr:L–193:av is an analog type (*ana*) external data point of the module, which is connected to input *in* of the function block *1hys*. Input *hyst* of the function block *1hys* has been given the value 0,0.5 in configuration phase. The function block output *out* is connected to the input *a* of the function block *2cmp*. The second input *b* of *2cmp* is connected with the interface port *in1*, which has been initialized to the value (0, 32.5).

All above–mentioned connection points of the function blocks are of the structured data type *ana*. Type *ana* consists of members *f*, which is of the derived primitive type *fails* and *a*, which is of the primitive type *float*.

Output *o* of *2cmp* is further connected to the local data point P1. This local data point is connected both to input *b* of the function block *3logic* and input *a* of the function block *4logic*. The result of the comparison in block *2cmp* is a logic value whose data type is *bin* (binary). Type *bin* is a derived primitive type (see Appendix 1). As only data of the same type can be connected, inputs *a* and *b* of the function blocks *3logic* and *4logic* are of the type *bin*. So are the external data pr:KR–11.F:out1 connected to input *a* of the function block *3logic* and the external data pr:507.108:ma connected to input *b* of the block *4logic*.

The results of the logic operations performed in the logic function blocks *3logic* and *4logic* – which are of the type *bin* – are connected from the function block outputs *o* to the interface ports pr:XZ–108.F:out1 and pr:XZ–108.F:out2 in the module interface. You should notice that in list format design a direct reference to the connection points of logic, comparison and calculation function blocks is not possible; the connection must be done through a local data point.

The following types appear in the module under examination:

```
   ana
      MEMBERS
         f TYPE fails
         a TYPE float

   bin    TYPE uns16

   hys
      MEMBERS
         in TYPE ana
         hyst TYPE ana
         out TYPE ana

LOCALS
   P1 TYPE bin;

INTERFACE
   in1 TYPE ana < (0,32.5);
   MOTSTAT TYPE ktstat < (1,1,0,1,1);
```

The number of inputs in comparison and logic function blocks and their data types are defined in configuration.

In our example, the block data structures are of the following types (2cmp and 3logic):

```
COMPARE 2cmp
   CONNECT
      a TYPE ana < 1hys:out;
      b TYPE ana < in1;
      o TYPE bin > P1;
   FORMULAS
      o = a >= b;
   STOP 2cmp

LOGIC 3logic
   CONNECT
      a TYPE bin < pr:50-KR-11.F:out1;
      b TYPE bin < P1;
      o TYPE bin > out1;
   FORMULAS
      o = a AND b;
   STOP 3logic
```

To illustrate the data types, the module can be shown in more detail as follows:

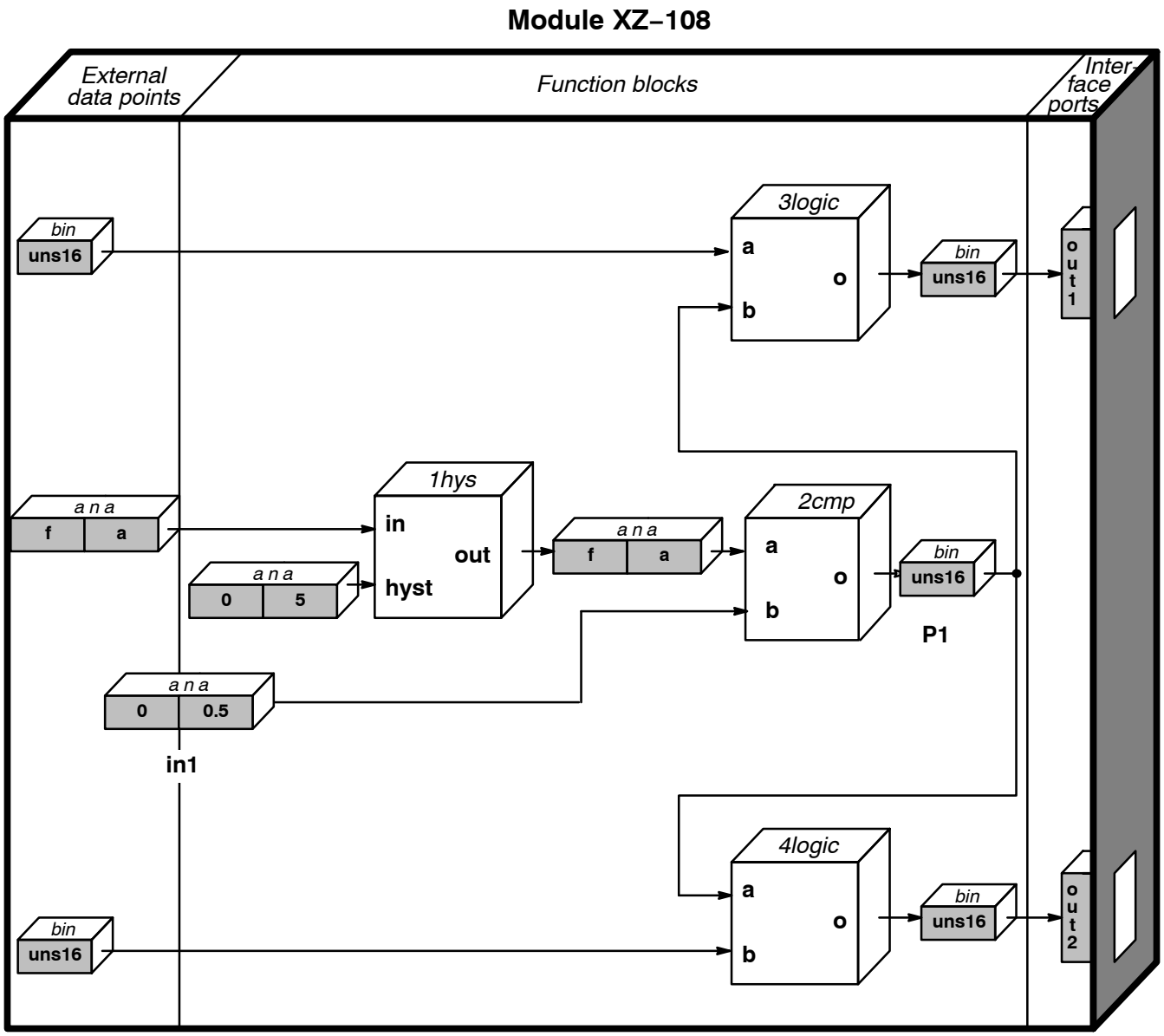


Figure 19 Module XZ–108

Function module pr:XZ−108.F is written in list form as follows:

```
ADMINISTRATION_PART
   NAME:        pr:XZ-108.F
   TYPE:        function
   STATUS:      done
   CREATOR:     tim
      CREATED:     89-01-02 08:36
   MODIFIER:    tim
      MODIFIED:    89-01-02 08:55
   DESTINATION: AP02
   EXECUTION:   400
      ORDINAL:     3
   DESCRIPTION: "EXAMPLE"

REPRESENTATION_PART
   EXTERNALS
      pr:L-193:av TYPE ana TRANSFER 192,4,0,0 "Mixing tank level" ;
      pr:KR-11.F:out1 TYPE bin TRANSFER 192,4,0,0 "Mixer group" ;
      pr:507.108:ma TYPE bin TRANSFER 192,4,0,0 "Mix.tank on man." ;

   LOCALS
      P1 TYPE bin;

   INTERFACE
      in1 TYPE ana < (0,32.5) ;
      out1 TYPE bin "507.108"< - ;
      out2 TYPE bin "507.108"< - ;
      MOTSTAT TYPE ktstat < (1,1,0,1,1);

FUNCTIONAL_PART
   1hys
    dchstv= (0)
    hyst< (0,0.5)
    in< pr:L-193:av
    out> -
   ;

   COMPARE 2cmp
      CONNECT
         a TYPE ana< 1hys:out;
         b TYPE ana< in1;
         o TYPE bin> P1;
      FORMULAS
         o = a >= b;
   STOP 2cmp

   LOGIC 3logic
      CONNECT
         a TYPE bin < pr:KR-11.F:out1;
         b TYPE bin < P1;
         o TYPE bin > out1;
      FORMULAS
         o = a AND b;
   STOP 3logic

   LOGIC 4logic
      CONNECT
         a TYPE bin < P1;
         b TYPE bin < pr:507.108:ma;
         o TYPE bin > out2;
      FORMULAS
         o = a AND b;
   STOP 4logic

END
```

6 The Automation Language's Naming Conventions

6.1 General

The automation language used in Valmet DNA supports name–based communications. In other words, data transfer in Valmet DNA relies on names.

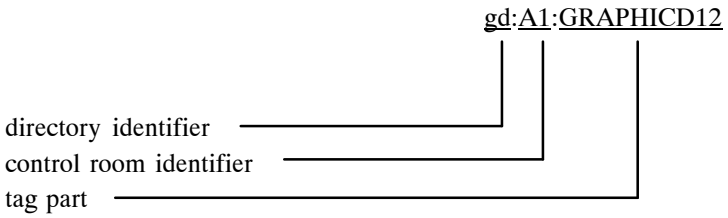
There are no strict limitations to the names that can be given to modules, but it is more practical to use a consistent designation.

A consistent designation has many advantages:

- the engineer does not have to develop his own designations
- troubleshooting and maintenance are easier for personnel external to the project

6.2 Structure and Length of Module Name

Module names consist of the following components:



Components are separated by colons (:).

The module name can be up to 63 characters long with all its components and separators.

An individual component between the colons can be up to 15 characters long.

6.3 Characters That Can Be Used in Module Name

Legal Characters

A – Z a – z 0 – 9 ' . / _ + =	No national characters!
, –	Components must not start with any of these characters!

Illegal Characters

control characters space tab character # ; " $ & () * ? ! % < >	–
:	Component separator

Characters That Are Not Recommended

The following characters can be used, but their use is not recommended:

@ [ ] ^ { } | \ ‘ ~

The reason why these characters are allowed is that in 7 bit character maps they are used as national characters.

6.4 Directory Identifier in Module Name

In order to be able to unambiguously tell the document or module from the name of an automation, documentation or configuration module, directory identifiers are used to help distinguish module types.

In automation modules, the directory identifier is written in uppercase letters. In configuration modules, it is written with lowercase letters. This makes automation modules and respective configuration modules unique in the engineering environment workspace.

6.4.1 Automation Modules

Directory identifiers are not used in the names of automation modules created with Function Block CAD.

LIC–100

The directory identifier for automation modules created with Sequence CAD is SQ.

SQ:SEQ–100

The directory identifiers for automation modules created with Picture Designer are *gd*, *rp* and *mw*.

gd:A1:GRAPHICD3	(graphic display)
rp:A1:RECIPED4	(recipe display)
mw:A1:MWINDOW5	(monitor window automation module)

6.4.2 Configuration Modules

Configuration modules are distinguished by directory identifiers and extensions (separated by a period at the end of the name).

NOTE!
XX stands for control room identifier, example: A1 (see step 6.5 “Control Room Identifier in Module Name”).

Following is a list of directory identifiers and module types by application server types:

APPLICATION SERVER	ID	EXTENSION	MODULE TYPE
PCS	pr:	.F	function module
	pr:	.I	input module
	pr:	.O	output module
	pr:	–	direct access port name
	sq:	.F	function module, sequence
	ph:	–	history module
	ph:me:	–	controller history module
	ph:spa:	–	controller history module
	ph:pos:	–	controller history module
	rp:	.V	recipe variation module
	rp:	.P	recipe parameter module
	rp:	.C	recipe calculation module
	rp:	.L	recipe loading module
DIS	xc:	.F	function module
	xi:	.IO	input/output module
CIS	ci:	–	function module
LIS	li:	–	function module
SIS	si:	–	function module
REP	rs:	.C	report collection, calculation and control module
	rs:	.R	report storing module
	rs:	.P	report printing module
ALS	al:XX:	.F	event module
	al:XX:	–	direct access port name
	sn:XX:	–	system module
OPS	ce:XX:	–	tag module
	cp:XX:	–	direct access port for tag module
	od:XX:	–	operating display module
	gd:XX:	–	graphic display module
	tr:XX:	–	trend display module
	rp:XX:	–	recipe display module
	sd:XX:	.st	step display module
	sd:XX:	.x	step module
	mw:XX:	–	monitor window module
	td:XX:	–	test display module
	sn:XX:	–	system module

The directory identifier of diagnostic sensors in Valmet DNA is *di*. The sensors are used in the diagnostic event modules.

6.5 Control Room Identifier in Module Name

The control room identifier is used in the names of modules in operation server and alarms and events server.

The control room identifier guarantees that the module name is unique in a network of many control rooms and buses. The control room identifier is a two character code where the first character must be a letter and the second a number. Legal characters are letters A...Z and numbers 0...9.

1. character = process area (design area) identifier A...Z
2. character = control room number 1...9

Example: gd:**A1**:GRAPHICD3

6.6 Tag Part in Module Name

Structure of the Tag Part

The tag part of a module name consists either of a tag identifier and device tag identifier with their extensions or of a name defined by the application engineer.

Example:

LC:**LIC–100**	(tag)
pr:**LT–100.I**	(device tag + extension)
gd:A1:**GRAPHICD3**	(display name)

Extension of the Tag Part

In configuration modules, the tag part is often followed by an extension, which is separated from the actual tag identifier with a period.

Example:

pr:LIC–100**.F**	(function module)
pr:LT–100.**I**	(input module)
pr:LV–100.**O**	(output module)

Extensions *F*, *I* and *O* are commonly used. They may also include a number to indicate, for example, the number of the output:

Example:

pr:LV–100.**O2**

As the example shows, when a device tag is used, the extension is not necessary to make the name unique. Still there are the following reasons why to use the extensions:

- Using extensions enables you, for example, to inquire all input module names from the repository. Without the extension or other unique marker, the inquiry would not succeed.
- The check tool of the engineering server uses extensions for calculating loading estimates and checking for duplicate card addresses in different I/O modules.

Extension may also come automatically to a tag part of configuration modules when engineering tools are used (see step 6.10 “Modules by Tools”).

Things to Be Considered in Tag Part

Tag and device tag identifiers are written in UPPERCASE LETTERS.

The tag identifier should contain possible process area or department code to make the name unique in the entire Valmet DNA. Think about the future!

Take care not to make the tag part longer than 15 characters. If you run out of space, remove some not so significant characters from the tag identifier:

Example:

LIC–100 –> LC–100
PIA–123 –> P–123

For a control room tag identifier, you can use the whole tag without process area or department code.

6.7 Designation of Display Modules

Display modules are named by display type. The name will be preceded by the directory identifier and control room identifier yy:XX. The directory identifier is written in lowercase letters. The directory identifier is derived from the display type as follows:

gd	**g**raphic **d**isplay
tr	**tr**end display
rp	**r**eci**p**e display
sd	**s**tep **d**isplay
od	**o**peration **d**isplay
td	**t**est **d**isplay
mw	**m**onitor **w**indow

In automation modules, the directory identifiers are written in uppercase letters.

6.8 Designation of System Modules

The directory identifier for system modules is either *sn* or an identifier derived from the display type. Following is a list of system modules, their destinations and types:

Name	**Destination**	**Type**
sn:XX:clomod	OPS,ALS	clock
sn:XX:hmod	OPS	hierarchy
sn:XX:syscrsX	OPS	activity
sn:XX:keycnfX	OPS	keyboard
sn:XX:pltmod	OPS	palette
sn:XX:menumod	OPS	menu
sn:XX:odheader	OPS	path
sn:XX:mdheader	OPS	header
sn:XX:login	OPS	user_id

Name	Destination	Type
td:XX:TDWHITE	OPS	test display
td:XX:TDGRID	OPS	test display
td:XX:TDBLACK	OPS	test display
td:XX:TDCARPET	OPS	test display
td:XX:TDGREYSCA	OPS	test display
td:XX:TDCS	OPS	test display
sn:XX:ALMLIST.F	ALS	alarm list
sn:XX:MSGLIST.F	ALS	message list
sn:XX:AREA.F	ALS	area definitions
sn:XX:HORN.F	ALS	horn definitions
sn:XX:printcon	ALS	printer definitions
sn:XX:ALMPRINT.F	ALS	printer control

6.9 Designation of Modules Produced by Operation Server

In testing and troubleshooting, it is useful to know the designation of the modules contained in Valmet DNA itself. This has been necessitated by user−configurable trends.

The operation server produces history modules to Valmet DNA.

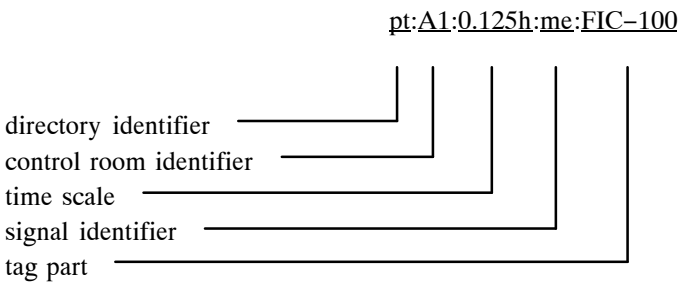
History modules are generated as a result of trend display selection operations.

The operation server uses base modules to ”generate” the modules.

6.9.1 Designation of History Modules

History module name consists of the following components:

Example:



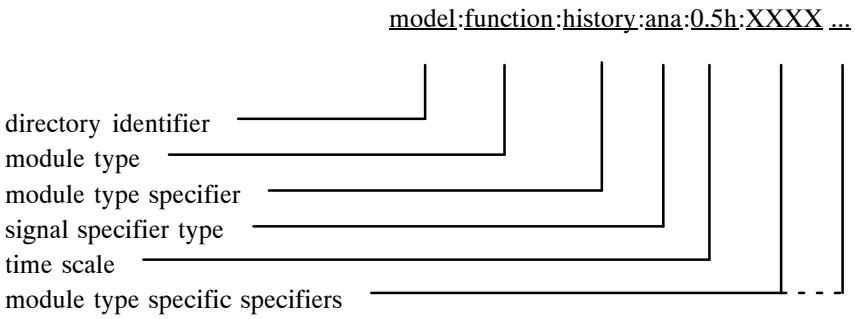
History modules are located at the application server indicated in the tag.

Scaling modules are located at the application server indicated in the tag.

6.9.2 Designation of Base Modules

Base modules are display or function module bases residing in the operation server. They are used for internal Valmet DNA configuration changes.

Example:



Base modules are part of the operation server product. Changes into them are made in the development department.

6.10 Modules by Tools

Following is a list of automation modules and configuration modules generated from them, sorted by tool.

NOTE!
XX stands for control room identifier, YYYY for application server identifier and Z for bus identifier.

TOOL	**NAME OF GENERATED MODULE/DOCUMENT**	**–**
Picture Designer	gd:XX:GRAPHICD3	graphic display module
	rp:XX:RECIPED4	recipe display module
	mw:XX:MWINDOW5	monitor window module

TOOL	**NAME OF GENERATED MODULE/DOCUMENT**	**–**
Function Block CAD	LIC–100	automation module/function diagram document
	pr:LIC–100.F	function module
	pr:LT–100.I	input module
	pr:LV–100.O	output module
	ce:XX:LIC–100	tag module
	od:XX:LIC–100	operating display module
	al:XX:LIC–100.F	event module
	RE1_V	automation module/recipe variation document
	rp:RE1.V	recipe variation module
	RE1_P	automation module/recipe parameter document
	rp:RE1.P	recipe parameter module
	RE1_C	automation module/recipe calculation document
	rp:RE1.C	recipe calculation module
	RE1_L	automation module/recipe loading document
	rp:RE1.L	recipe loading module
	REP1_C	automation module/report collection, calculation and control document
	rs:REP1.C	report collection, calculation and control module
	REP1_R	automation module/report storing document
	rs:REP1.R	report storing module
Sequence CAD	SQ:SEQ–100	automation module/sequence diagram document
	sq:SEQ–100.F	function module, sequence
	ce:XX:SEQ–100	tag module, main sequence
	ce:XX:SEQ–100.L	tag module, subsequence
	od:XX:SEQ–100	operating module
	al:XX:SEQ–100.F	event module, main sequence
	al:XX:SEQ–100.L.F	event module, subsequence
	sd:XX:SEQ–100.st	step display module
	sd:XX:SEQ–100.x	step module
Control Diagram CAD	CD:HEADBOX	document module/control diagram document
Hardware CAD	HW:50RK001	document module/hardware drawing document

6.11 Module Destination Data

In automation modules and configuration modules, the destination data is used to indicate in which Valmet DNA application server the configuration modules are located.

The destination data reserves four characters; the first character must be a letter. Legal characters in destination data are letters A...Z and numbers 0...9.

6.11.1 Structure of the Destination Data

1. character = process area (design area) identifier A..Z
2. character = application server identifier A...Z or control room number 1...9
3. character = application server number 0...9 or control room identifier A...Z
4. character = application server number 0...9

Application Server Identifier Letters

Application server name	Abbrev.	Identifier	Example
Process Control Server	PCS	P	A**P**01
Report Server	REP	E	A**E**01
Alarms and Events Server	ALS	A	A1**A**1,A1
Operation Server	OPS	O	A1**O**1,A1O,A1
Diagnostics Server	DIA	D	A**D**01
Backup Server	BU	B	A**B**01
Damatic Interface Server	DIS	I	A**I**01
Computer Interface Server	CIS	C	A**C**01
Logic Interface Server	LIS	L	A**L**01
Sensor Interface Server	SIS	S	A**S**01
Router Server	RTS	X	A**X**01

Designation of Application Packages in Operation Server and Alarms and Events Server

In the case of a control room, the destination may instead of one application server (A1O1) also refer to a group of several application servers (A1O). In this case, all modules marked with this destination identifier go to all application servers belonging to this group.

In operation server and alarms and events server application packages to be sent to application server groups are named as follows:

- picture modules, example: A1O

  the modules are loaded to all operation servers of control room A1
- tag modules, example: A1

  tag modules are loaded to all operation servers of control room A1 and to alarms and events servers

7 Fault Bit Conventions

The data created in Valmet DNA and provided with a connection that makes it possible to change data through application configuration, generally consist of two bundled members: the base type data and fault bits.

This chapter explains how fault bits should be treated.

7.1 Meanings of Fault Bits

Fault bits indicate faults detected in the signal processing chain — right from the transmitter. They are able to indicate several faults at the same time. Following is a list and short description of fault bits:

ext (2)

External fault:
fault in the transmitter or signal wire.

ovf (4)

Data overflow:
input signal out of the permissible signal range.

dis (8)

Control disabled:
signal cannot be controlled by Valmet DNA.

inv (16)

Invalid data:
signal value is not based on any measurement.

old (32)

Old data:
signal value has been frozen.

der (64)

Fault on derived data:
one or more of the input signals of a derived signal is faulty.

sex (128)

Source exceptional:
the data has been produced by exceptional data source, e.g. data is simulated.

The values of the fault bits in decimal system are presented in brackets.

7.2 What Different Fault Bits Indicate

On one hand, fault bits indicate the validity of the data, on the other hand, they indicate exceptional conditions in the signal path — from its source to the user.

Some fault bits are interpreted as faults while others are only informative. The decision between a fault or a ”for your information” situation is usually made case by case when using the data.

In general, only real faults require taking some remedial steps.

7.2.1 ext – External Fault

Common cause:

- Connection wires broken or short–circuited, wrong transmitter tuning or faulty transmitter.
- A fault outside Valmet DNA distorts the message read in Valmet DNA and prevents Valmet DNA from controlling the device.

If the fault affects the measured variable directly, FBC reacts by freezing the value to the value that preceded the fault, and sets the *old* fault bit to indicate this condition. If the external fault does not directly affect the measured variable, FBC does not freeze the measurement. Conclusions on the effect of the fault on the variable are made by the I/O unit and FBC.

In the case of a read–back of an output, FBC will not freeze the value but uses the value that the I/O unit tried to write in the output. Because of the external fault, the real output value is not known.

7.2.2 ovf – Data Overflow

Common cause:

- Measured variable out of the tuned transmitter range.

Is not generally considered a fault.

The signal read from the transmitter is under 0 % or over 100 % of the set signal range (e.g. under 4 mA or over 20 mA). This makes the signal inaccurate and its exact value cannot be measured, because the input unit does not measure values outside the signal range, and it is not known how much the value differs from the actual signal value. The signal value is set to low or high limit of the signal range, depending on whether the range was exceeded or undercut — if it has not been frozen for some other reason.

Is also used when time stamp of a binary signal is inaccurate.

This fault bit requires remedial steps only if it is necessary because of an inaccurate reading.

7.2.3 dis – Control Disabled

Common cause:

- Output has been set to local mode in the output unit or output has been set to a state in FBC where the data written from Valmet DNA is not sent to the field.
- The output cannot be controlled from Valmet DNA in the normal way, because the signal path is cut in the FBC (disabled control) or in the output (local control).

The fault bit requires remedial steps if it is necessary because of an unabled control.

7.2.4 inv – Invalid Data

Common cause:

- After starting the configuration, the value cannot be read or it cannot be accepted.

Because no acceptable signal value could be read, the signal retains its configured default value. Or the sender wants to tell the user of the data that the signal should not be used.

In general, the fault bit causes the value to be rejected.

7.2.5 old – Old Data

The current value of the signal is not known for sure, because it is not updated with the real value. To what extent such signal can be used is determined by the situation preceding the setting of the *old* fault bit: whether the last updated value was valid or invalid and whether the value has been updated at all.

The fault bit requires remedial action only if an update failure or an invalid data is a problem.

7.2.6 der – Fault on Derived Data

The signal is derived from other signals, and in some of the input signals the fault bit *inv*, *old* or *der* is set.

7.2.7 sex – Source Exceptional

Measurement

Value is not read from the transmitter but the signal path is cut in the FBC. The value is the value given by the input simulation.

Output

The read−back value does not come from the I/O unit, but the last written output value is used instead.

It is possible to set the input/output to simulation mode also in other units besides the FBC.

Not generally interpreted as a fault.

The fault bit requires remedial action only if the cut signal path is a problem.

When a measurement is simulated, the fault bits are also simulated. In other words, the simulation point transmits the fault bits defined by the user. In a simulated measurements, no conclusions on the validity of the signal can be made beyond the simulation point.

7.3 Signal Alarms from Fault Bits

Some data types are provided with signal alarms. The alarm is set when the signal is faulty.

The signal alarm can be used in the application to tell the operator that a fault has been detected in the connection of the signal.

As a rule, the signal alarm is given if any of the fault bits *ext*, *inv*, *old* or *der* is set. *ovf*, *dis* and *sex* do not set the signal alarm.

7.4 Some Notes to Be Noted in Using Fault Bits

If the fault bit *inv* is set alone, the signal value will not be shown on the control room monitor. If some other bit is set besides the *inv*, the value will be shown.

7.5 On Applying Fault Bit Conventions

7.5.1 Initial Values of Types

The initial value given by the type is applied if the engineer has not initialized the module and he has not connected the data, or the connection does not work.

7.5.2 Initial Values in Modules

The initial values for modules will remain set, if the application engineer has not connected the data to a point outside the module.

If the data is connected, the fault bits defined by the engineer only last until the connection is made with some exceptions that are listed in the following step.

If the data includes fault bits, the application engineer must always define both the value and the fault bits.

7.5.3 Data to Be Connected

In the case of data that the application engineer has connected outside the module, Valmet DNA works as follows:

If the connection cannot be made when the module is started, the fault bit *old* is set. Other fault bits remain as the application engineer set them, or if he did not initialize them, they will remain as they were set in the type defaults.

When the connection is made, the fault bits will be updated by the connected data.

If the connection breaks, the fault bit *old* will be set. Other fault bits remain as they were when the connection was still open.

Primitive Types

BASIC TYPES

TYPE	FUNCTION	RANGE	SIZE IN BYTES
char	unsigned character variable	character	1
int8	signed integer	–128...+127	1
int16	signed integer	–32768...+32767	2
int32	signed integer	–2147483648... +2147483647	4
uns8	unsigned integer	0...255	1
uns16	unsigned integer	0...65535	2
uns32	unsigned integer	0...4294967295	4
bool8	Boolean variable. Variable value is false if all bits of the variable are false. Value is true if any bit of the variable is true.	true/false	1
bool16	Boolean variable. Variable value is false if all bits of the variable are false. Value is true if any bit of the variable is true.	true/false	2
float	single precision floating point number By default, NaN values in PCS function blocks are replaced with zeros.	$MIN_- = -1.18*10^{-38}$ $MIN_+ = +1.18*10^{-38}$ $MAX_- = -3.4*10^{38}$ $MAX_+ = +3.4*10^{38}$	4

DERIVED TYPES

fails, type uns16

- The type *fails* contains 16 bits (b0–b15), 16 separate items of logical data, that can be referred to by using the conventions of the automation language.
- The symbolic names and functions of the bits are as follows:

BIT	NAME	FUNCTION
b0	b	≡ 0 (value of variable in derived type *bin*)
b1	ext	transmitter supply failure or line fault (external)
b2	ovf	inpu signal beyond limits (overflow)
b3	dis	signal cannot be controlled (control disabled)
b4	inv	unreliable data (invalid)
b5	old	data not updated (old)
b6	der	derived fault (derived)
b7	sex	source exceptional
b8	(not in use)	
b9	–	
b10	–	
b11	–	
b12	–	
b13	–	
b14	–	
b15	–	

- E.g.: If the variable ”pressure” is of structured type ”ana”, its fault bit ”pressure:f:inv” is invalid data.

bin, type uns16

- The type *bin* contains 16 bits, 16 separate items of logical data. Bit b0 is the value of type *bin*. When referring to a variable of the type *bin* in the automation language, you refer to this value. Bits b1–b15 are fault data of type *bin*; this data is defined in connection with the type *fails*.
- Automation language allows the handling of fault data in separate logical entities. For instance, if variable *lim* is of the type *bin*, ”lim:b” is its value and ”lim:der” is its fault bit (derived value).

Common Structured Types

Type	Function	Members	Function of members	Type of members
ana	analog signal	f a	fault bits value	fails float
ints	short integer	f s	fault bits value	fails int16
intl	long integer	f l	fault bits value	fails int32
time	time and date	hundus tenms sec min hour wday day month year vers	100 microseconds 10 milliseconds seconds minutes hours day of week day month year time version (b7 – b5) and time zone (b4 – b0)	uns8 uns8 uns8 uns8 uns8 uns8 uns8 uns8 uns8 uns8
bo	binary output	bv pw	value of binary output pulse width of binary output	bin uns16
binev	time stamped binary signal	binstat bintime	value time stamp	bin time
anaev	time stamped analog signal	anastat anatime	value time stamp	ana time
intsev	time stamped short integer	intsstat intstime	value time stamp	ints time
intlev	time stamped long integer	intlstat intltime	value time stamp	intl time