# Table of Contents

# QVIEW 6.0

# 1. Introduction

The QVIEW (QMOVE Viewer) development environment, QCL language (QEM Control Language) and LADDER language (contact logic) are the tools needed for developing, debugging and editing projects dedicated to the QMOVE system. The potential of this combination of hardware and software allow the creation of controls for industrial automation, with particular regard to handling management (electrical, hydraulic, CNC or ON / OFF).

## 1.1 QVIEW: QMOVE VIEWer

As with any compiled and structured text language, even QCL and Ladder require an editor to type in lines of code and a compiler that translates machine language syntax checking; QVIEW is the development environment that, in addition to supporting the QCL language and Ladder language allows to transfer the program to the CPU of the system QMOVE to monitor and modify all the variables and parameters of the program and to stop the execution to understand and correct any unwanted operations (DEBUG).

## 1.2 QCL: QEM Control Language

QCL is a structured language with syntax and commands that will be familiar to programmers involved in planning or that already had the opportunity to experience a high level language. However, anyone who is approaching Structured Text languages for the first time will find in this manual a valuable support.

## 1.3 Ladder

*Ladder* is a logic contact language familiar to anyone acquainted with PLC programming. If approaching the Ladder method for the first time, it would be best to take a general course in PLC programming to integrate this manual.

## 1.4 Devices

*Devices* perform typical functions in industrial automation (e.g. reading an analog input, a signal from a position translator, executing actual axis positioning). Each device has a specific manual, please consult them for specific details.

## 1.5 QPaint

An application of products QEM in an industrial automation may need to be of a project developed in Qview (for the control and automation) and is of a design for all that concerns the interface with the operator. QPaint is the environment for the development of the graphical interface for the operator terminals QEM. The two projects are related to each other because they refer to the same data structure (variables, arrays, etc.).

# 2. Introduction to the environment

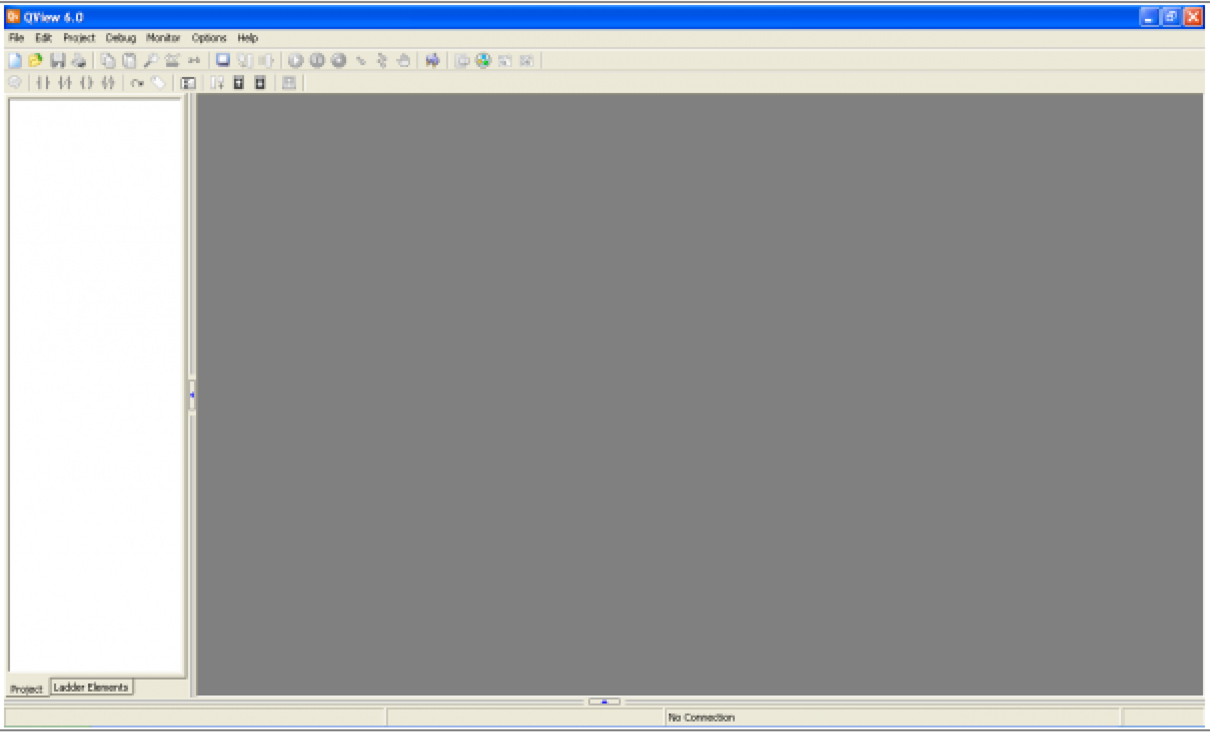
This section will introduce the development environment (QVIEW) by presenting a simple demo example, that can be used as a practice tool during learning.
The basic steps that will be developed:

- Opening and creating a demo project.
- Compiling the demo project.
- Connecting a PC to QMOVE.
- Downloading the compiled project on the CPU.
- Monitoring its execution.

## 2.1 Opening and Creating a Demo Project

QVIEW 6.0 is part of the QWorkbench software package. When QVIEW 6.0 is installed and started up, the following opening screen is viewed (Figure 1).

Figure 1: First startup



- There is a menu at the top of the screen with all the operations possible.
- Below there is a tool bar with icons for the most common operations.
- The bottom of the screen has a status bar gives any most important information.
- The grey window is the work area where all the windows are opened for development, debugging and monitoring.
- The white side window lists the units in the project or the ladder elements that can be used.

### 2.1.1 Creating a new project

To create a new project select:

- File - New Project

enter a project name, e.g. “FirstApp”. A window opens to enter the project information. It can be left empty for now and completed later. The project can now be saved to create the project file. Select:

- File - Save

select a directory to save the project. Now create two basic *units* to make a minimum project that can be compiled. Select:

- File - Add Unit - Configuration Unit

leave *Unit Name* as CONFIG and press OK. A new unit will appear in the white window and an editor window will open in the *Work Area*. This new window lists the code lines in the unit. This unit will now be modified. Only one Configuration Unit can exist in a project (no others of this kind can be added) and it is used to declare the project variables or data structure, to declare the devices being used and the type of hardware for the project.

### 2.1.1.1 Configuration unit

The configuration unit is used to declare:

- constants, variables, datagroups, digital inputs and digital outputs. Each of these symbols can be read or written anywhere in the project (see below on how to declare variables with local scope);
- the hardware;
- all *devices* used in the project.

Now introduce some changes to the configuration unit. Search the unit for the *SYSTEM* keyword (search tool hotkeys CTRL+F). This is the existing code:

```
;---------------------------------------------------------------------
; Definitions of SYSTEM variables
;---------------------------------------------------------------------
SYSTEM
```

change to:

```
;---------------------------------------------------------------------
; Definitions of SYSTEM variables
;---------------------------------------------------------------------
SYSTEM
   count   L   ;This variable is used as a counter
```

This syntax declares a *count* variable, it is an *L (Long)* type and belongs to the *SYSTEM* group.

- *SYSTEM*: a group of variables that memorise their value when the system is shut-off. The value is maintained when the system starts again.
- *Long*: a variable type that takes up 4 bytes in memory. This type of variable can contain values between -2147483648 and 2147483647;
- *count*: the name assigned to the variable.

**All code lines starting with “;” are comments. They have no effect on project execution.**

Another action is to indicate the make up of the QMOVE system. First of all, search for the keyword *BUS*.

```
;---------------------------------------------------------------------
; BUS configuration
;---------------------------------------------------------------------
BUS
```

Change this to:

```
;---------------------------------------------------------------------
; BUS configuration
;---------------------------------------------------------------------
BUS
   1  1P51  30
   2  .     .
   3  1MG8F .
   4  .     .
```

These code lines depend on the type of hardware used in the project. In the documentation for installing the qmove control in use, you can get information about the configuration of the BUS section.

### 2.1.1.2 QCL unit

Add a new unit to the project. Select:

- File - Add Unit - QCL Unit

leave Unit Name as UNIT1 and press OK. A second unit is created with the following default code lines:

```
MAIN:
  WAIT 1
  JUMP MAIN
END
```

change this to:

```
MAIN:
  count = count + 1
  WAIT 1
  JUMP MAIN
END
```

A project can have several of this type of task unit. The code lines between the *MAIN* label and *JUMP MAIN* instruction is repeated continuously in an infinite cycle. The code line *count = count + 1* increases the *count* variable at each cycle.

### 2.1.2 Compiling the Demo Project

Before downloading the project onto the CPU it has to be compiled. Compilation can be started from the menu:

- Project - Compile

At the end of compilation messages give the outcome. The demo project does not have errors and so the message should be *Project compiling OK!*.

## 2.2 Connecting a PC to QMOVE

Before opening the connection, the QRM (Qem Resource Manager) has to be running. Its icon must be visible on the Windows bottom toolbar to show QRM is running. Otherwise start it from the QWorkbench package directory.

The connection can be made by serial port using the IQ009 USB-Serial converter convertitore IQ009, or by a PC ethernet port to the Qmove ethernet port (if mounted).

To download the compiled project onto the CPU connect the PC to the QMOVE system. Select:

- Options - Open connection

Double click on the correct resource from list.

## 2.3 Downloading the compiled project to the CPU

At this stage the compiled project is ready to be downloaded onto the CPU. Select:

- Project - Download

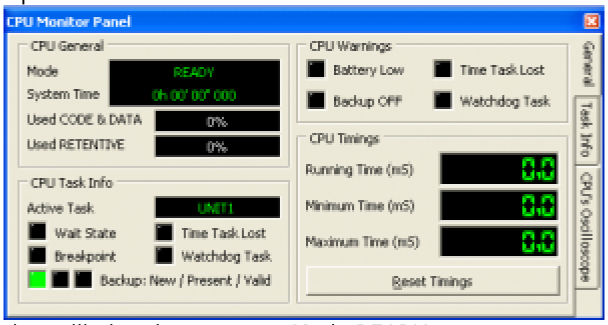
and the project will be downloaded. A progress bar shows the download status.

## 2.4 Monitoring execution

The system is now ready to execute the application. Select:

- Monitor - CPU

Open the window



CPU Monitor Panel

CPU General
Mode: READY
System Time: 0h 00' 00" 000
Used CODE & DATA: 0%
Used RETENTIVE: 0%

CPU Task Info
Active Task: UNIT1
Wait State
Time Task Lost
Breakpoint
Watchdog Task
🟩 ■ ■ Backup: New / Present / Valid

CPU Warnings
Battery Low
Time Task Lost
Backup OFF
Watchdog Task

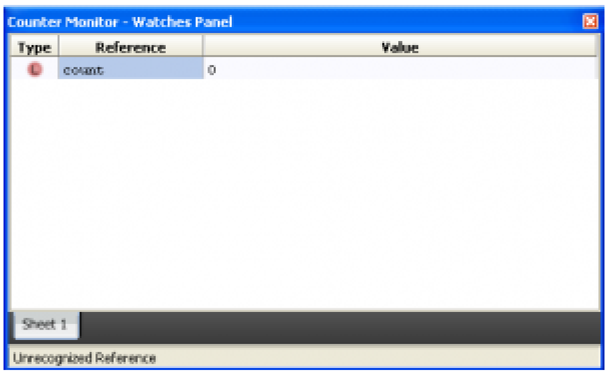
CPU Timings
Running Time (mS): 0.0
Minimum Time (mS): 0.0
Maximum Time (mS): 0.0

that will give the message *Mode READY*.

Select the command

- Debug - Run,

the application is executed; the CPU status becomes RUN. Now click and enter a new *Watches Panel* with the name *Counter Monitor* (or any name). In the first line of the *Watches Panel* enter the name of the variable in the demo project *count*.


Counter Monitor - Watches Panel

| Type | Reference | Value |
|---|---|---|
| ● | count | 0 |

Unrecognized Reference

When the CPU is in RUN the variable value will increase.

# 3. Introduction to programming

## 3.1 Declarations

The following declarations are possible in a QMOVE project:

- symbol declarations;
- device declarations;
- hardware declarations.

These declarations can be made

- in the configuration unit, or
- in other task units of the project that also contain the code lines.

*Symbols* are names assigned to variables, arrays, datagroups, constants, inputs or outputs. The various declarations are distinguished by the keywords placed before each group of declarations. The keywords:

- CONST
- SYSTEM
- GLOBAL
- ARRSYS
- ARRGBL
- TIMER
- INPUT
- OUTPUT
- INTDEVICE
- DATAGROUP (only in configuration unit)
- BUS (only in configuration unit)
- REFERENCES (only in configuration unit)

### 3.1.1 Configuration Unit

The configuration unit is a component of a QMOVE project. Only one configuration unit can exist in a project.
The configuration unit has to contain the declarations of variables and constants used in the application and it also defines the hardware in the QMOVE system, specifying the CPU card type and card models.
The variables, arrays, datagroups, devices and constants declared in the configuration unit are available throughout the project. Their scope is the whole project and therefore they are readable and writeable by all the other units.

### 3.1.2 Task Units

A QMOVE project can have several other task units, containing the code lines that determine the project's operating logic. Moreover the same type of declarations as in the configuration unit can be made at the start of the task units (except the hardware, input and output declarations).
The variables, arrays, datagroups declared in the task units can be given additional accessibility properties. Normally, when a variable is declared in a task unit it can only be used in that unit, however it can also be assigned an additional property in the declaration:

- IN, the symbol is accessible in write/read by other task units, while it is only accessible in read by the unit declaring the symbol
- OUT, the symbol is accessible in write/read by the declaring unit, while the other task units can only use it in read
- INOUT, the symbol is accessible in write/read both by the declaring unit and all other task units.

## 3.2 Variable types

All the information necessary to declare variables is provided below.

| |
|---|
| **In the syntax descriptions the parts in <…> are mandatory. The parts in […] are optional, while the parts separated by "/" indicate "use one of these".** |
| **All code lines starting with ";" are comments. They have no effect on project execution.** |

QCL has five main data types:

### 3.2.1 Flag

FLAG type data is used to define boolean variables with a whole values range between 0 and 1. Its memory space depends on their quantity.
The syntax for the definition of a FLAG type variable:

```
<variable name>  F  [OUT/IN/INOUT] [;comment]
```

### 3.2.2 Byte

BYTE type data is used to define variables with a whole number range between -128 and +127. It takes up one byte of memory.
The syntax for the definition of a BYTE variable:

```
<variable name>  B  [OUT/IN/INOUT] [;comment]
```

### 3.2.3 Word

WORD type data is used to define variables with a whole number range between -32768 and +32767. It takes up two bytes of memory.
The syntax for the definition of a WORD variable:

```
<variable name>  W  [OUT/IN/INOUT] [;comment]
```

### 3.2.4 Long

LONG type data is used to define variables with a range of whole values -2147483648 and +2147483647. It takes up four bytes of memory. LONG variable definition syntax:

```
<variable name>  L  [OUT/IN/INOUT] [;comment]
```

### 3.2.5 Single

SINGLE type data is a floating point value as defined by the *IEEE 754 single precision*.
The syntax for the definition of a SINGLE variable:

```
<variable name>  S  [OUT/IN/INOUT] [;comment]
```

### 3.2.6 Double

DOUBLE type data is a floating point value as defined by the *IEEE 754 double precision*.
The syntax for the definition of a DOUBLE variable:

```
<variable name>  D  [OUT/IN/INOUT] [;comment]
```

### 3.2.7 Overview of Variable types

**Assigning an out of range value to a variable creates an overflow condition.**

| Data type | Code | Memory space (Bit) | Range |
|---|---|---|---|
| FLAG | F | 1 | 0 - 1 |
| BYTE | B | 8 | -128 - 127 |
| WORD | W | 16 | -32768 - 32767 |
| LONG | L | 32 | -2147483648 - 2147483647 |
| SINGLE | S | 32 | See specification *IEEE 754 single precision* |
| DOUBLE | D | 64 | See specification *IEEE 754 double precision* |

## 3.3 Identifiers

Identifiers are names used to refer to objects or labels. An identifiers is made up of one or more alphanumeric characters, always starting with a letter. The characters that can be used:
- “A” to “Z”
- “a” to “z”
- “0” to “9”

- "_"

### 3.3.1 Names

Names are used to help identification of an object within the QMOVE system. Object is intended as any part that can be managed by the language and having its specific the physical characteristics: e.g. variables, inputs, outputs and devices. Names can have a maximum length of 64 characters.

## 3.4 Constants

Constants are a character or character string used as a value in a project. The constants are declared after the keyword *CONST*. The syntax for defining constants:

```
CONST
<nome costante> <valore> [OUT] [;comment]
```

dove:

| | |
|---|---|
| CONST | Keyword for defining constants |
| <constant name> | Name of constant |
| <value> | value associated to the constant |
| [OUT] | A declaration in the task unit used to make the constant accessible in read also in other task units |
| [;comment] | Optional comment recommended for better comprehension of the software |

A typical declaration of constants:

```
CONST
  TM_SECOND  1000  OUT  ;These are the number of milliseconds in a second.
  DIM_ARRAY  7     OUT  ;Used as array dimension.
  DIM_PROG   10         ;Used as number of recipe.
  DIM_STEP   5          ;Used as number of step for each recipe.
```

## 3.5 The SYSTEM variables

**SYSTEM variables are retentive, i.e. they maintain their value even after the system is shut off.**

SYSTEM groups all retentive variables, they can be FLAG, BYTE, WORD, LONG or SINGLE type. They have to be placed after the keyword *SYSTEM*. The syntax for defining SYSTEM variables.

```
SYSTEM
  <variable name>    <F/B/W/L/S>  [OUT/IN/INOUT] [;comment]
```

where:

| | |
|---|---|
| SYSTEM | Keyword indicating the definition of *SYSTEM* variables |
| <variable name> | variable name of *SYSTEM* group |
| <F/B/W/L/S> | Variable type |
| [OUT/IN/INOUT] | Accessibility type. |
| [;comment] | Comment for a better comprehension of the software. |

Example:

```
SYSTEM
  count      L       ;This is used as a counter.
  sbSeconds  B  OUT  ;Seconds
  sbMinutes  B  OUT  ;Minutes
  swHours    W       ;Hours
```

## 3.6 GLOBAL variables

***GLOBAL* variables are non-retentive, i.e. they are reset to zero each time the is system starts up**

GLOBAL groups all non-retentive variables, they can be FLAG, BYTE, WORD, LONG and SINGLE type. They are declared after the keyword *GLOBAL*. The syntax for defining GLOBAL variables.

```
GLOBAL
  <nome variabile>   <F/B/W/L/S>  [OUT/IN/INOUT] [;comment]
```

dove:

| | |
|---|---|
| GLOBAL | Keyword indicated for defining GLOBAL variables. |
| <nome variabile> | Name of GLOBAL variable. |
| <F/B/W/L/S> | Variable type. |

| | |
|---|---|
| [OUT/IN/INOUT] | Accessibility type. |
| [;comment] | Comments are essential to aid programmers understand the software |

Example:

```
GLOBAL
  gfMyFlag       F   OUT   ;Non retentive flag variable
  gbYourByte     B   IN    ;Non retentive byte variable
  gwHisWord      W   INOUT ;Non retentive word variable
  glYourLong     L   IN    ;Non retentive long variable
  gsTheirSingle  S   OUT   ;Non retentive single precision variable
```

## 3.7 ARRAY SYSTEM variables

An Array System variable is a group of the same type of retentive variables. They have the same size and can be accessed by a common name and refer to a specific element by an index. FLAG type variable arrays cannot be used.
As with SYSTEM variables, they are retentive variables and have to be declared after the keyword *ARRSYS*.
The syntax for defining ARRAY SYSTEM variables:

```
ARRSYS
  <nome variabile>   <F/B/W/L/S>   <numero_elementi>  [OUT/IN/INOUT] [;comment]
```

dove:

| | |
|---|---|
| GLOBAL | Keyword for the definition of ARRAY SYSTEM variables. |
| <nome variabile> | Name of ARRAY SYSTEM variable. |
| <F/B/W/L/S> | Variable type |
| <numero_elementi> | Number of elements making up the variable. |
| [OUT/IN/INOUT] | Accessibility type. |
| [;comment] | Comment for a better comprehension of the software. |

An array can have a maximum number of elements of 65535. Negative or fractional dimensions cannot be used. The ARRAY dimension can use constants already defined in the CONST section, and if it contains decimal digits they are rounded down: e.g. 200.34 is forced to 200.
Example:

```
ARRSYS
  asbMyArray    B 10         INOUT  ;Array system declaration of 10 bytes in size.
  aslYourArray  L DIM_ARRAY  IN     ;Array system declaration of DIM_ARRAY bytes in size.
```

It can be seen that the second array uses a constant for the dimension declaration.

## 3.8 ARRAY GLOBAL variables

An Array Global variable is a group of the same type of non retentive variables. They have the same size and can be accessed by a common name, referring to a specific element by an index. FLAG type variables cannot be used.
As with GLOBAL, they are non retentive variables and have to be declared after the keyword “ARRGBL”.
The syntax for defining ARRAY GLOBAL variables:

```
ARRGBL
  <nome variabile>   <F/B/W/L/S>   <numero_elementi>  [OUT/IN/INOUT] [;comment]
```

dove:

| | |
|---|---|
| ARRGBL | Keyword for defining ARRAY GLOBAL variables. |
| <nome variabile> | Name of ARRAY GLOBAL variable. |
| <F/B/W/L/S> | Variable type |
| <numero_elementi> | Number of elements in the variable. |
| [OUT/IN/INOUT] | Access type. |
| [;comment] | Optional comment recommended for better comprehension of the software. |

Example:

```
ARRGBL
arwMyArray  W  15 OUT  ;Array global declaration of byte of 15 dimension.
```

## 3.9 TIMER variables

TIMER variables are used to create timers that can be assigned a whole number value (expressed in ms) that represents the time that has to pass (from the assignment time); the “Timing ended” (1) or “Timing in course” (0) statuses are provided in read.
Timer variables are declared after the keyword “TIMER”.

The syntax for defining TIMER variables:

```
TIMER
  <timer name> [OUT/IN/INOUT]  [;comments]
```

dove:

| | |
|---|---|
| TIMER | Keyword for defining TIMER variables |
| <timer name> | Name of the TIMER variable |
| [OUT/IN/INOUT] | Accessibility type |
| [;comment] | Optional comment recommended for better comprehension of the software. |

Example:

```
TIMER
  tTimer1   INOUT  ;First timer.
  tTimer2   OUT    ;Second timer.
  tSeconds  OUT    ;3rd timer.
  tDelay    INOUT  ;4th timer.
```

When a timer variable is placed on the left of the assignment operator a value is set in the timer (in milliseconds):

```
tMyTimer = 1000  ;Set timer tMyTimer to 1 second.
```

When a timer variable is placed on the right of the assignment operator or in an expression only the status is read (0 = Timing in course, 1 = Timing ended):

```
  gfIsTimerEnd = tMyTimer ;Assign to gfIsTimerEnd variable the timer state.
```

or

```
IF(tMyTimer)  ;If timer tMyTimer has passed exec the code
  ;Put here the code
  .....
ENDIF
```

It is also possible to read the remaining time value before the timer ends (the read value is expressed in milliseconds):

```
< Timer Name >.remain
```

Example:

```
glRemainTime = tMyTimer.remain
```

## 3.10 INPUT and OUTPUT variables

All variables referring to digital inputs and outputs. Their declaration must come after the keyword *INPUT* for inputs or *OUTPUT* for outputs.

**The digital inputs and outputs can only be declared in the configuration unit.**

The syntax for defining INPUT and OUTPUT variables:

```
INPUT
  <variable name>   <F/B/W>   <io address>
OUTPUT
  <variable name>   <F/B/W>   <io address>
```

**The I/O addresses are given in the hardware technical data of the card.**

where:

| | |
|---|---|
| INPUT | Keyword for defining INPUT variables |
| OUTPUT | Keyword for defining OUTPUT variables |
| <variable name> | Name of variable, i.e. the symbol associated to the input or output |
| <F/B/W> | Variable type |
| <io address> | INPUT or OUTPUT address comprising:<br>*Slot number.name: Slot number* is the number of the slot with the card containing the hardware resource.<br>*name*: the reference name of I/O's physical address (defined in hardware references). |

An interesting application of the digital inputs and outputs is to group them in under a single identifier. This identifier is similar to an eight or sixteen bit variable where each input or output represents a bit.
Example:

```
INPUT
  ibInput  B  3.INPB     ;8 digital input grouped in a single byte
```

```
OUTPUT
  obOutput  B  3.OUTB      ;8 digital output grouped in a single byte
```

If the card has more than eight digital inputs or outputs Example:

```
INPUT
  ibInput1  B  3.INPB1   ;First 8 digital input grouped in a byte (1-8).
  ibInput2  B  3.INPB2   ;Second 8 digital input grouped in a byte (9-16).
  iwInput   W  3.INPW    ;A word of 16 digital input grouped in a word (1-16).
```

```
OUTPUT
  obOutput1  B  3.OUTB1   ;First 8 digital output grouped in a byte 8 (1-8).
  obOutput2  B  3.OUTB2   ;Second 8 digital output grouped in a byte 8 (9-16).
  owOutput   B  3.OUTW    ;A word of 16 digital output grouped in a word (1-16).
```

## 3.11 DATAGROUP variables

Datagroup variables are a special data structures, that are stored in retentive memory. They are declared in the configuration unit and are suitable to represent a recipe database.
Datagroup variables contains two kinds of variable:

### 3.11.1 Static Variables

```
DATAGROUP
  <DataGroup Name>
  DATAPROGRAM
    <number of recipes>
;Static variables declaration
    <variable name> <F/B/W/L/S>
    <variable name> <F/B/W/L/S>
    <variable name> <F/B/W/L/S>
```

Static variables reside in DATAGROUP subsector named DATAPROGRAM. The first value of this sector is a number or an integer constant, and represents the number of recipes in archive. Static variables are to be considered as containers of a certain value for each recipe, and are accessed as an array, where the index is the number of desired recipe. Example:

```
DATAGROUP
  Name
  DATAPROGRAM
    100
;Static variables declaration
    Variable1     L
    Variable2     S
    Variable3     F
```

To refer to variable *Variable3* in recipe 5, the used code is:

```
Variable3[5]
```

We can imagine the memory structure of a variable DATAGROUP, in our example with static variables, as the following table:

| | **Variable1** | **Variable2** | **Variable3** |
|---|---|---|---|
| **Recipe 1** | | | |
| **Recipe 2** | | | |
| **Recipe 3** | | | |
| **…** | | | |
| **Recipe 100** | | | |

### 3.11.2 Indexed Variables

```
DATAGROUP
  <DataGroup Name>
  DATAPROGRAM
    <number of recipes>
  STEP
    <number of steps>
;Indexed variables declaration
    <variable name> <F/B/W/L/S>
    <variable name> <F/B/W/L/S>
    <variable name> <F/B/W/L/S>
```

Indexed variables reside in DATAGROUP subsector named STEP. Their function is to introduce for each recipe, the concept of step. The first value of this sector is a number or an integer constant, and represents the number of steps for each recipe in archive. Indexed variables are to be considered as containers of a list of values for each recipe, and are accessed as a 2-dimensional array, where the first value is the number of the recipe, the second the number of desired step. Example:

```
DATAGROUP
  Name
  DATAPROGRAM
    100
;Static variables declaration
    Variable1     L
    Variable2     S
    Variable3     F
  STEP
    10
;Indexed variables declaration
    Variable4     W
    Variable5     B
```

To refer to variable *Variable4* in recipe 5 to step 9, the used code is:

```
Variable4[5, 9]
```

We can imagine the memory structure of a variable DATAGROUP, in our example with static and indexed variables, as the following table:

| | Variable1 | Variable2 | Variable3 | Variable4 | Variable5 |
|---|---|---|---|---|---|
| **Recipe 1** | | | | | |
| **Recipe 2** | | | | | |
| **Recipe 3** | | | | | |
| **...** | | | | | |
| **Recipe 100** | | | | | |

Other informations:

- In a DATAGROUP all variables, both static and indexed, are retentive (maintain the value at power off).
- The Datagroup variables may be more than one, in this case it is necessary to insert more DATAGROUP keywords.
- The subsection DATAPROGRAM is mandatory, while the STEP is optional.
- The maximum number of recipes can be set is 65534.
- The maximum number of steps can be set is 65534.
- Compared to a common array, static variables can also support a Flag size data (F).

## 3.12 BUS Section

The BUS section in the configuration unit is essential to declare the QMOVE hardware model being used.
In accordance with the declared hardware model it is also possible to obtain its hardware resources, e.g. the number of digital inputs and outputs, analog inputs and outputs, or count inputs . The QMOVE may also have other resources like serial interfaces or removable mass storage slots.
The declaration made in the *BUS* section is divided in *slots*. *Slot 1* indicates which CPU is being used, while the other *slots* declare the electronic cards installed in the model. Each card is identified by a keyword that identifies the type of hardware. The keywords are provided in the hardware installation and maintenance manuals.

The syntax:

```
BUS
   1  <ID_CPU>  <FW>
  [2  <ID_CARD>  .]
  [3  <ID_CARD>  .]
  [4  <ID_CARD>  .]
```

Example:

```
BUS
  1 1P51F  30     ;QMOVE J1-P51F with firmware 30
  2 .       .     ;No card is installed in this slot
  3 1MG8F   .     ;This card has 32 digital input, 32 digital output, 6 analog output, etc.
```

# 4. Variable scopes

The scope of symbols declared in a project is very important, so it is worth dedicating a specific chapter to this topic.
As explained, the following groups can be declared

```
CONST
SYSTEM
GLOBAL
ARRSYS
ARRGBL
TIMER
INPUT
OUTPUT
INTDEVICE
```

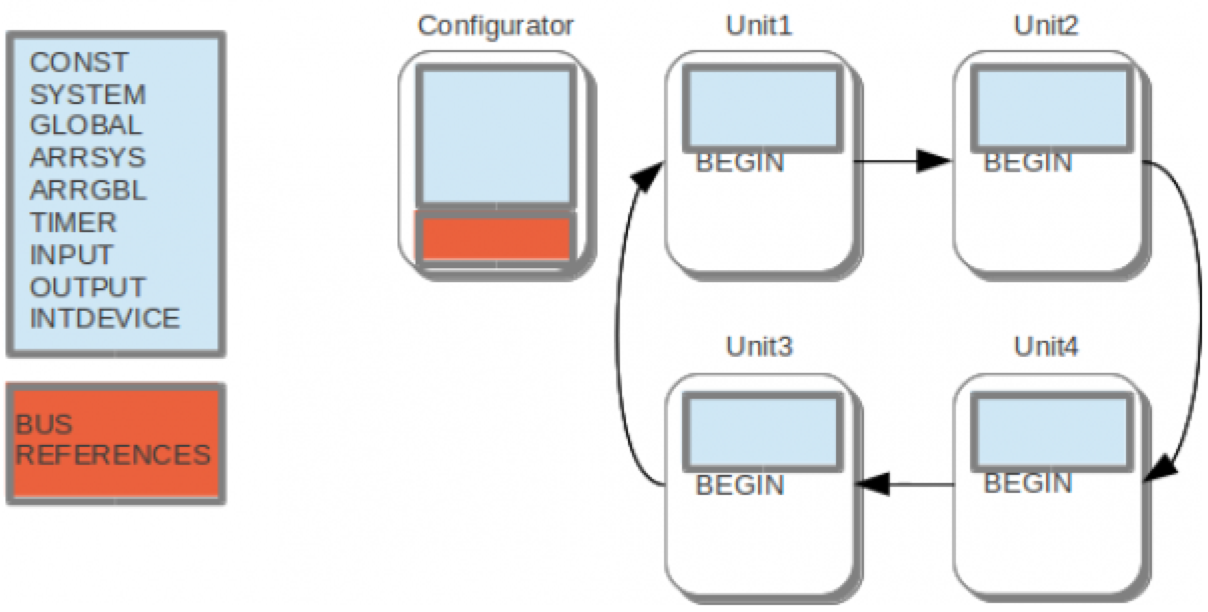
both in the configuration unit and in task units included in the project. The other declaration groups

```
BUS
REFERENCES
```

can only be included in the configuration unit.

**The addition of the *REFERENCES* keyword demands a detailed explaination, which is given at the end of this chapter.**

The figure below shows a chart of the various declaration groups in a project.



## 4.1 Configuration variables

Symbols declared in the configuration unit have their scope extended to all the task units. This example declares a variable in the SYSTEM group:

```
  SYSTEM
    ConfigurationVariable  L  ;This variable is declared in the configuration unit
```

This variable can be used in any task unit of the project both to assign it a value (i.e. write) and for use as a value in an expression (i.e. read).

```
  ConfigurationVariable = 5                   ;Assign the value 5 to the variable
  Unit1Variable = ConfigurationVariable * 2   ;Assign twice variable value to another variable
```

In practice, the following complete note refers to the *ConfigurationVariable* variable:

```
  APPLICATION.ConfigurationVariable
```

The *APPLICATION.* root can be omitted (see the next to have more informations about).

## 4.2 Local variables

Symbols declared in task units containing the code lines, before the *BEGIN* keyword, have different scopes. For example:

```
GLOBAL
  local_variable  W  ;This variable is declared in a task unit
```

```
BEGIN
  ...  ;Here are the lines code
```

the scope of this variable is limited to within the task unit where it is declared. No other task unit can access this variable in read or write. It is considered a *local* variable, since it is declared for a scope that is satisfied within that unit. It is possible for another unit to contain the declaration of a variable with the same name. In fact there is no chance of confusion because when the *local_variable* symbol is used in a task unit, it is clear that it always refers to the *local* variable and not another one in another task unit.

## 4.3 IN/OUT/INOUT variables

If certain attributes are added to the declaration of a variable an interface can be created towards the outside of the task unit containing the declaration. The attributes:

- IN
- OUT
- INOUT

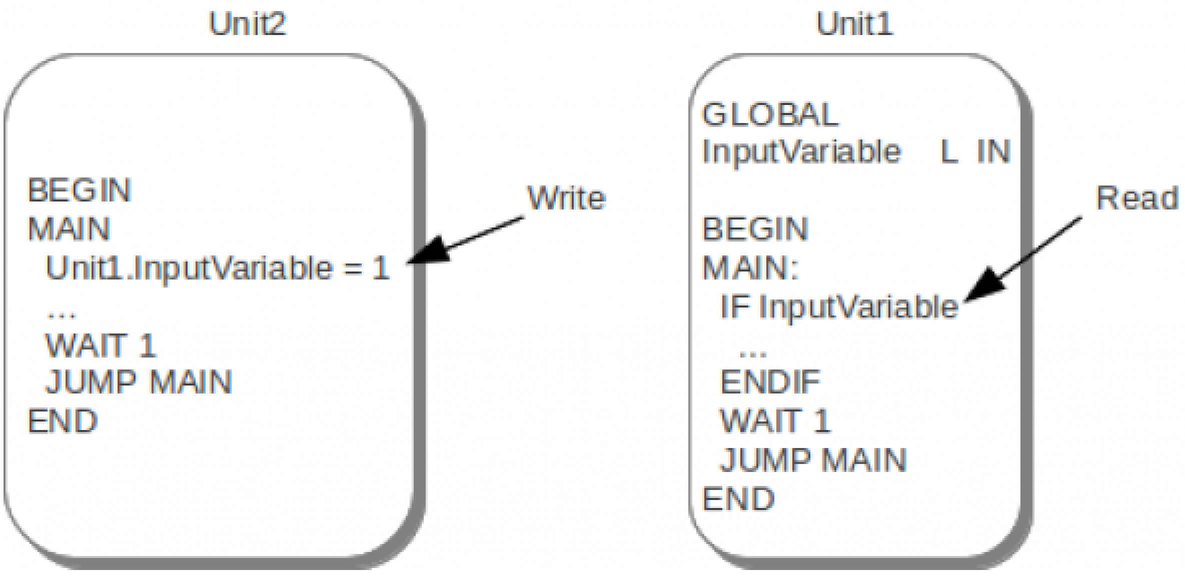
To illustrate this possibility, the previous example can be expanded:

```
GLOBAL
  InputVariable    L  IN     ;This variable is like an input
  OutputVariable   L  OUT    ;This variable is like an output
  InOutVariable    L  INOUT  ;This variable is both an input and an output

  local_variable   W         ;This variable is declared in a task unit
BEGIN
  ...  ;Here are the lines code
```

thereby obtaining a unit interface towards the other task units.

### 4.3.1 IN

The unit sees the *InputVariable* variable as an input and so it cannot be modified internally but only in its value. In pratice it is an read only variable for the task unit that declares it.



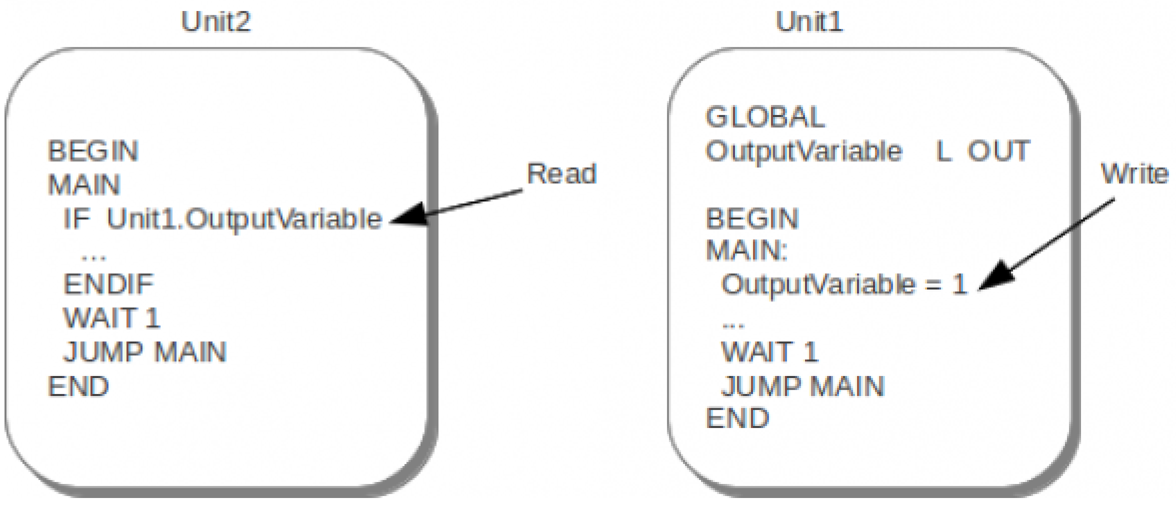
As shown in the figure, the other task units (Unit2) can access it if they specify which unit the variable belongs to:

```
  Unit1.InputVariable = 1
```

In this way several task units can have the same variables with IN properties. Confusion cannot exist because each variable belongs to units with different names.

### 4.3.2 OUT

The *OutputVariable* variable is an output for the task unit declaring it. So it is the result of processing within the unit and is used to supply this result to other task units. The variable declared with this property is only in read for all the other units.



```
BEGIN
MAIN
 IF  Unit1.OutputVariable
   ...
 ENDIF
 WAIT 1
 JUMP MAIN
END
```

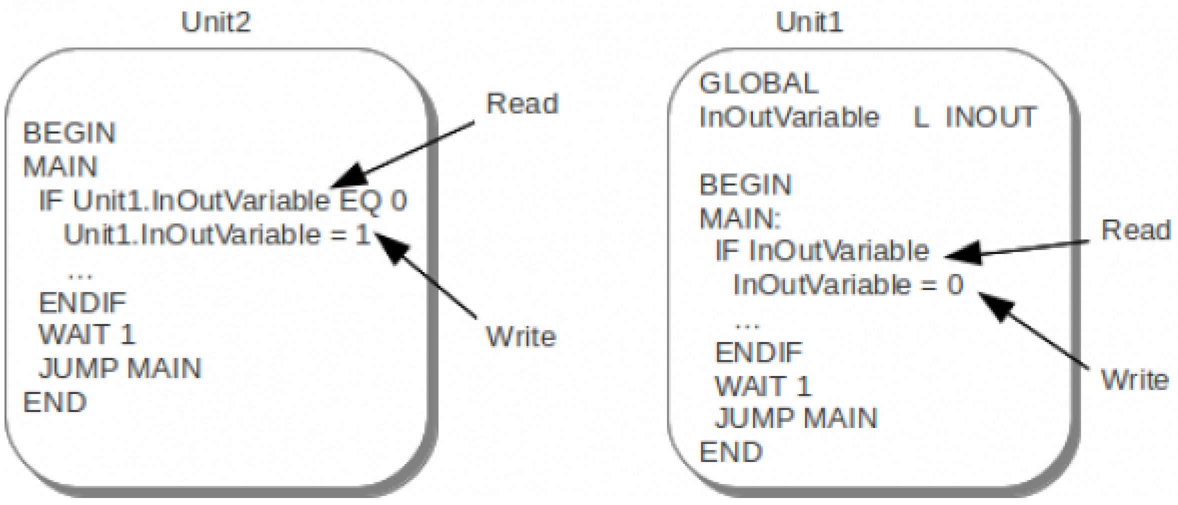
```
GLOBAL
OutputVariable    L  OUT

BEGIN
MAIN:
 OutputVariable = 1
 ...
 WAIT 1
 JUMP MAIN
END
```

As shown in the figure, the other task units (Unit2) can only use the value of the *OutputVariable* variable if they first specify the name of the unit the variable belongs to:

```
IF Unit1.OutputVariable
...
ENDIF
```

## 4.3.3 INOUT

The *InOutVariable* variable is both an IN and an OUT. This variable can be modified both in the declaring unit and in the other task units.



```
BEGIN
MAIN
 IF Unit1.InOutVariable EQ 0
   Unit1.InOutVariable = 1
   ...
 ENDIF
 WAIT 1
 JUMP MAIN
END
```

```
GLOBAL
InOutVariable    L  INOUT

BEGIN
MAIN:
 IF InOutVariable
   InOutVariable = 0
   ...
 ENDIF
 WAIT 1
 JUMP MAIN
END
```

Even in this case the other task units (Unit2) can use the Unit1 variable by specifying its name before the unit name it belongs to :

```
IF Unit1.OutputVariable EQ 0
  Unit1.OutputVariable = 1
...
ENDIF
```

## 4.4 APPLICATION

As said, this is the full name to identify variables declared in the configuration unit:

```
APPLICATION.<variable name>
```

Use the *APPLICATION* root before the variable name when there could be sources of confusion. In fact, if the *HomePosition* variable is declared in the configuration unit and another variable with the same name in Unit1, this could be a source of misunderstandings.
This is why the name

```
APPLICATION.HomePosition
```

and

```
Unit1.HomePosition
```

depend on which it refers to.

## 5. REFERENCE

A different attribute that can be assigned to a symbol in the declaration is REFERENCE. In this case the symbol declared as REFERENCE is simply a reference to a variable declared in another point (typically in the configuration unit). There are two stages in the declaration of these special symbols:

1. declaration in the task unit where it is used
2. declaration in the configuration unit (in the REFERENCES section) defining the reference variable.

Example:

In Unit1

```
GLOBAL
  MachineState  B  REFERENCE
```

In the Configuration unit

```
GLOBAL
GeneralState  B
...
REFERENCES
Unit1.MachineState  = GeneralState
```

The *GeneralState* variable is a universal symbol that is associated to the variable declared in Unit1 and called *MachineState*. The references must always be made between the same type of symbols. References can be created for all types of declaration (i.e. SYSTEM, GLOBAL, ARRSYS, ARRGBL, INPUT, OUTPUT, INTDEVICE, TIMER).

**The declaration of INTDEVICE, INPUT, OUTPUT can only be made as REFERENCE.**

## 6. COSTANTS

Constants can also be declared in the task unit containing the code. Example of this kind of declaration in Unit1:

```
CONST
  MAX_PROGRAM_NUMBER  100  ;local constant
  ...
BEGIN
  ...
```

the *MAX_PROGRAM_NUMBER* constant can only be used inside the declaring task unit. However if the following declaration is made:

```
CONST
  MAX_PROGRAM_NUMBER  100  OUT  ;Output constant
  ...
BEGIN
  ...
```

the scope of the *MAX_PROGRAM_NUMBER* constant is extended to the other task units. The rule of access by other task units is still valid:

```
  Unit1.MAX_PROGRAM_NUMBER
```

# 7. QCL Instructions

QCL (QEM Control Language) has been conceived specifically for programming the QMOVE system. The QCL salient features are simplicity (i.e. few powerful instructions), ease of use (i.e. similarity to BASIC) and its orientation towards industrial automation. with instructions especially designed for axis control.

## 7.1 QCL operators

The QCL language offers all elementary data processing operators.

### 7.1.1 Assignment

The assignment operator can be used in any expression and can act on all variables, both elementary and belonging to DataGroups or Device parameters. The typical form of an assignment instruction :

```
<variable name> = <expression>
```

If the size of the *variable name* variable is less than the result of the expression, the value is rounded down according to the type conversion rules (see appendix).

### 7.1.2 Arithmetic operators

The arithmetic operators:

- add (+),
- subtract (-),
- multiply (*),
- divide (/),
- division remainder (%).

These operators act on all data types and their combinations (including bit).
Example:

```
  sbVar01 = slVar02 + swVar03
  slVarxy = glVarA * glVarB
  sl01 = ss02 % ss03
```

### 7.1.3 Logic operators

The logic operators (i.e. AND, OR and NOT) can act on all variables except single precision type. Their function is summarised in the table below.

**Value 1 simply indicates that the variable is different to 0.**

| a b | a AND b | a OR b | NOT a |
|---|---|---|---|
| 0 0 | 0 | 0 | 1 |
| 0 1 | 0 | 1 | 1 |
| 1 0 | 0 | 1 | 0 |
| 1 1 | 1 | 1 | 0 |

Examples

```
  glBit01 = glBit02 AND gfBit03
  glBitA = NOT glBitB
```

### 7.1.4 Binary operators

The binary operators

- ANDB,
- ORB,
- NOTB,
- XORB

act on all variables including SINGLE ( after automatic conversion to whole numbers). These operators create the same t/f table that governs the equivalent logic operations, except they act on the single bits.

| $(a)_{10}$ | $(b)_{10}$ | $(a)_2$ | $(b)_2$ | a ANDB b | a ORB b | NOTB a | a XORB b |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 011 | 010 | 010 | 011 | 100 | 001 |
| 7 | 5 | 111 | 101 | 101 | 111 | 000 | 010 |

Example

```
swVar01 = gwVar02 ANDB gwVar03
```

### 7.1.5 Relational operators

Relational operators ascertain relations between values:

| | |
|---|---|
| EQ | equal to |
| NEQ | not equal to |
| LT | lower than |
| LE | lower or equal) |
| GT | greater than |
| GE | greater or equal |

Example

```
gfBit01 = glCont GE glAsse01
```

### 7.1.6 Operator Priority

Round brackets “()” modify the operator priority.
Example AA = [(A + B)/C] - (D x E)

```
gwVarAA = ((gwVarA + gwVarB) / glVarC) - (swVarD * glVarE)
```

### 7.1.7 QCL Operator Priority Levels

The priority of QCL operators is the same as all programming languages:

| **Top priority** |
|---|
| ( ) |
| Math functions and Trigonometric functions |
| NEG NOT NOTB |
| * / % |
| + - |
| LT GT LE GE |
| EQ |
| AND ANDB |
| OR ORB XORB |
| = |
| **Bottom priority** |

## 7.2 Instructions and flow control structures

All the instructions that modify the execution flow.

### 7.2.1 IF / ELSE / ENDIF

A typical conditional, so if the condition is true, i.e. boolean value 1, instruction set 1 is executed otherwise instruction set 2 is executed. An instruction set can be made up of a group of instructions or even a single instruction. What's more the ELSE set is optional. The syntax:

```
  IF <condition>
    <instruction set 1>
  [ELSE
    <instruction set 2>]
  ENDIF
```

Example:

```
MAIN:
  IF tTime                    ;If timer has expired
    tTime = TM_SECOND         ;reload the timer and
    count = count + 1         ;increase by 1 the variable count
  ENDIF
  WAIT 1
  JUMP MAIN
END
```

Example of nested IF loop.
Change the example above as follows:

```
MAIN:
  IF tSecond                      ;If the timer has expired
    tSecond = TM_SECOND           ;reload the timer.
    IF sbSeconds LT 59            ;If the sbSeconds variable is less than 59
      sbSeconds = sbSeconds + 1   ;increase the seconds counter
    ELSE                          ;else (if the seconds are greater than 59)
      sbSeconds = 0               ;reset the seconds
      IF sbMinutes LT 59          ;If the sbMinutes variable is less than 59
        sbMinutes = sbMinutes + 1 ;increase the minutes counter
      ELSE                        ;else (if the minutes are greater than 59)
        sbMinutes = 0              ;reset the minutes
        IF swHours LT 167         ;If the swHours variable is less than 167
          swHours = swHours + 1   ;increase the hours
        ELSE                      ;else (if the hours is greater than 167)
          swHours = 0             ;reset the hours, because a week has passed.
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  WAIT 1
  JUMP MAIN
END
```

This creates a system clock with three system variables: one for seconds, one for minutes and one for hours. A byte type variable is used for the seconds and minutes since they never leave the range [-128, 127], while the swHours variable uses a Word type variable since it can reach 167, remaining within the range [-32768, 32767], being the hours in a week.

## 7.2.2 FOR-NEXT

An iterative loop normally used to perform an instruction block or single instruction, one or several times. The FOR-NEXT loop can be exited, before the condition becomes false, using the BREAK instruction, which skips to the next instruction after NEXT. The syntax:

```
  FOR (<initialization>, <test>, <increment>)
    < code lines >
  NEXT
```

where:

| | |
|---|---|
| <initialization> | is an expression executed at the start of the cycle and therefore only once |
| <test> | is a conditional expression which, if false, causes the end of the FOR loop |
| <increment> | s a number value that is added to the index at the end of each execution of < code lines > |

Example:

```
MAIN:
  FOR(gbYourByte = 1, gbYourByte LE 10, 1)      ;For gbYourByte from 1 to 10
    arwMyArray[gbYourByte] = POW(3, gbYourByte) ;the element arwMyArray[gbYourByte] gets the value 3^gbYourByte,
    dswStat2[gbYourByte] = 1                    ;assign the value 1 to the static variable,
    ddbDin1[gbYourByte,1] = 2                   ;assign the value 2 to the indexed variable if the 1st step.
  NEXT
  WAIT 1
  JUMP MAIN
END
```

The FOR-NEXT loop fills the array global *arwMyArray* with the first 10 powers of 3.

## 7.2.3 WHILE-ENDWHILE

An iterative loop used to perform an instruction or instruction block until a given condition is found.
Exit the WHILE-ENDWHILE loop before it becomes false by using the BREAK instruction, which jumps to the first instruction after ENDWHILE. The syntax:

```
  WHILE (<condition>)
    < code lines >
  ENDWHILE
```

where:

| | |
|---|---|
| <condition> | is a conditional expression that has to give a boolean result. When the condition is true, the < code lines > are executed continuously. When the condition is false, the code following ENDWHILE is executed. |

Example:

**These two loops are made to set the value of the sbMinutes and swHours variables keeping two different inputs activated, the ifSetUpMin and ifSetUpHours inputs. It can be seen that tdelay, a delay time has been inserted in each WHILE instruction to slow down the increase in the variable and provide a way to disable the input when the variable reaches the required value. Of course the value assigned to minutes or hours is limited so an IF is added in the WHILE loop to control this limit. The *WAIT 1* instruction is explained below.**

```
MAIN:

  WHILE (ifSetUpHours)           ;While the input is active:
    IF tDelay                    ;if the timer has expired
      tDelay = 500               ;reload the timer,
      swHours = swHours + 1      ;increase the hours counter.
      IF swHours GE 168          ;If the hours are greater than 167 (greater or equal to 168)
        swHours = 0              ;reset the hours counter.
      ENDIF
    ENDIF
    WAIT 1
  ENDWHILE

  WHILE (ifSetUpMinutes)         ;While the input is active:
    IF tDelay                    ;if the timer is expired
      tDelay = 500               ;reload the timer,
      sbMinutes = sbMinutes + 1  ;increase the minutes counter.
      IF sbMinutes GE 60         ;If the minutes are greater than 59 (greater or equal to 60)
        sbMinutes = 0            ;reset the minutes counter.
      ENDIF
    ENDIF
    WAIT 1
  ENDWHILE
WAIT 1
  JUMP MAIN
END
```

## 7.2.4 SWITCH-ENDSWITCH

QView 6 has the SWITCH instruction, which can analyse a given expression and execute different code lines depending on the expression result. The syntax:

```
  SWITCH(<expression>)
    CASE <value 1>
      < lines code for value 1>                                            [;comments]
    CASE <value 2>..<value 3>
      < lines code for values between value 2 and value 3>                 [;comments]
    CASE <value 4>, <value 5>..<value 7>
      < lines code for value 4 and for values between value 2 and value 3>  [;comments]
    [DEFAULT
      < lines code for default>]                                           [;comments]

  ENDSWITCH
```

A typical example of a temperature control measured by a sensor that writes its value in the *Temperature* variable.

```
  SWITCH(Temperature)
    CASE 0
      CloseDoor = 1
      WarmUp = 100
      IceAlarm = 1
      OutOfRange = 0
    CASE 1..10
      CloseDoor = 1
      WarmUp = 50
      IceAlarm = 0
      OutOfRange = 0
    CASE 11..15
      CloseDoor = 1
      WarmUp = 15
      IceAlarm = 0
      OutOfRange = 0
    CASE 16..25
      CloseDoor = 0
      WarmUp = 0
      IceAlarm = 0
      OutOfRange = 0
    DEFAULT
      OutOfRange = 1
  ENDSWITCH
```

## 7.2.5 WAIT

Wait for a condition or an event. The WAIT instruction is very important when it is executed and the <condition> is false, so execution of the instructions passes onto the next task unit. However a particular must be noted at this stage: the first time a WAIT instruction is found, it always causes the passage of control to the next task, without checking the condition. When control of the instruction flow returns to the task in question, it executes the WAIT instruction and then checks the condition. For more details see the Multitasking chapter. The syntax:

```
WAIT <condition>
```

## 7.2.6 JUMP

An unconditional jump to the instruction at the specified label. The syntax:

```
JUMP <label>
```

Example
The classic example of a JUMP instruction is where it is repeated in each task to jump from the end to start of the code, i.e. from

the MAIN label.

This flow control instruction can be also used in other points of the task unit. JUMP is an unconditional jump that can be made conditional with

an IF to deviate the code flow under specific conditions.

### 7.2.7 LABELS

Labels locate given points in the sequence of instructions in an application. They are used to change the execution flow of instructions (the QCL JUMP command). When a label is inserted in the instructions it must be followed by a colon “:”, while when referring to the label the name is written without a colon.
Example of an unconditional jump

```
Label1:                 ;labeled line
  < code line >
  < code line >
  ;....................
JUMP Label1             ;Jump to the label instruction
```

Example of conditional jump

```
Label1:                 ;labeled line
  < code line >
  IF <condition>        ;If the condition is true
    JUMP Label1         ;Jump to the label instruction
  ENDIF
```

### 7.2.8 SUBROUTINE

When a part of the code is repeated several times in the same unit, it is useful to define a subroutine. The subroutine code is executed every time a subroutine call instruction is found (CALL). The subroutine is identified by a name after the SUB keyword and must end with the ENDSUB keyword. See also the CALL instruction.
To stop a subroutine before the end use the keyword RETURN.

### 7.2.9 CALL

Call to Subroutine: an unconditional jump to the first instruction in the named subroutine. At the end of the subroutine (identified by ENDSUB) the program flow continues from the instruction following the CALL.

```
CALL <subroutine_name>
```

**These code lines are placed after the END keyword, i.e. after the end of the application unit.**

A subroutine is introduced to calculate the length of a diagonal of a square, given the length of a side.

Example of subroutine call:

```
MAIN:

  glOurLong = 4             ;This is the square edge
  CALL CALC_DIAG            ;Subroutine call
  WAIT 1
  JUMP MAIN
END

;-------------This subroutine calculates a square's diagonal-----------------
SUB CALC_DIAG
  gsYourSing = SQRT(2 * POW(glOurlong,2))
ENDSUB
```

### 7.2.10 NOP

No operation: this instruction has no effect on the program and is used to introduce delays.

## 7.3 Instructions for digital outputs

Instructions for activating or deactivating a digital output. The activation and deactivation can also be executed by the assignment operator

(<output name>= 1), however dedicated instructions are faster.

### 7.3.1 SETOUT

activated. This output is then reset when the sbSeconds value is 40.^ An example can be introduced where an output status changes status according to the value of a variable:

```
MAIN:
  IF (sbSeconds EQ 20) AND ifEnable01    ;If sbSeconds is equal to 20 and the input is active
    SETOUT ofOutput1                     ;set the output
  ENDIF
  IF sbSeconds EQ 40                     ;When the sbSeconds is equal to 40
    RESOUT ofOutput1                     ;reset the output
  ENDIF

  WAIT 1
  JUMP MAIN
END
```

Activate a digital output. The syntax:

```
SETOUT <output name>
```

### 7.3.2 RESOUT

Deactivate a digital output. The syntax:

```
RESOUT <output name>
```

## 7.4 System Instructions

Instructions that allow to control unit execution (units containing QCL or LADDER codes) in a program. These instructions are best understood

after consulting Multitasking.

### 7.4.1 SUSPEND

A suspension in the execution of the specified task unit .
This instruction can be used by a unit to block the execution of another unit or to block the execution of itself. In this case, since the unit in question is suspended, it can no longer be reactivated but this must be done by one of the other units in execution. The Syntax:

```
  <unit name>.SUSPEND
```

### 7.4.2 RESUME

Restore the execution of an instruction in a suspended unit. This instruction restores execution of a suspended unit from the next instruction following the one executed at the moment of suspension. The syntax:

```
  <unit name>.RESUME
```

RESUME is only effective if the unit <unit name> was suspended.

### 7.4.3 RESTART

“Restarts” execution of a task unit from the first instruction. This instruction does not alter the suspended status of the unit in question, which means that if a suspended unit is given a RESTART, it prepares for the execution of the first instruction but remains suspended until it is activated by the RESUME instruction. The syntax:

```
  <task_name>.RESTART
```

To best understand the meaning of these system instructions see “Multitasking”.

## 7.5 System Variables

The QMOVE system provides variables related to the internal operating system and these variables have pre-set names, i.e. they cannot be changed by the user.

### 7.5.1 is_suspended

Gives the status of a unit:

- 0 = unit in execution
- 1 = unit suspended

The syntax:

```
  <unit name>.is_suspended
```

### 7.5.2 w_dog_task

*Watchdog task* indicates the execution time of that unit is over 200 ms. The same information appears in the *Monitor - CPU* panel under *Watchdog Task*. The syntax:

```
  <unit name>.w_dog_task
```

### 7.5.3 time_task_lost

Indicates that the time unit has missed an event. The same information is given in the *Monitor - CPU* panel under *Time task Lost*. A time unit must be executed at preset times. These times may not be respected because of the load of executing the code in the unit. If this happens it is known as a *Time task lost* for a given unit.
The syntax:

```
<unit name>.time_task_lost
```

### 7.5.4 battery_low

The battery charge is low and the BATT led blinks. This only refers to QMOVE models mounting a backup battery for the RAM memory.
The syntax:

```
QMOVE.battery_low
```

### 7.5.5 battery_down

The battery is flat and the BATT led is on. This only refers to QMOVE models mounting a backup battery for the RAM memory.
The syntax:

```
QMOVE.battery_down
```

Other variables depend on the CPU firmware. To access these variables use the syntax:

```
QMOVE.sys001
QMOVE.sys002
....
QMOVE.sys016
```

For details on these variables consult the hardware documentation.

## 7.6 Math Functions

### 7.6.1 Exponentiation

Exponentiation of the base. The syntax:

```
POW(<base>, <exponent>)
```

Example: gsMaxVal = $2^{gwNbit}$

```
gsMaxVal = POW(2,gwNbit)
```

### 7.6.2 Square root

Calculation of the square root of the argument. The syntax:

```
SQRT(<radicand>)
```

Example: gsIpot = v(gsL1$^2$ + gsL2$^2$)

```
gsIpot = SQRT(POW(gsL1,2)+POW(gsL2,2))
```

### 7.6.3 Natural Logarithm

Calculation of the natural logarithm of the argument. The syntax:

```
LN(<val>)
```

Example: gsValue = ln gsMaximum

```
gsValue = LN(gsMaximun)
```

### 7.6.4 Exponential

The exponentiation of the Nepero number. The syntax:

```
EXP(<exponent>)
```

Example: gsA = $e^2$

```
gsA =EXP(2)
```

### 7.6.5 Absolute value

The absolute value of the argument. The syntax:

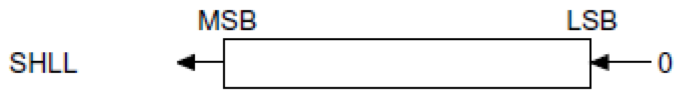
```
ABS(<variable>)
```

Example: glModule = I glValue I

```
glModule = ABS(glValue)
```

### 7.6.6 Shift logical left

Logic shifts the contents of var value to the left by *n* bits.



The syntax:

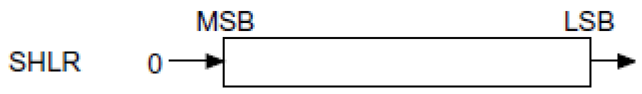
```
SHLL(<variable>,<bits>)
```

Example:

```
glValue = SHLL(glValue, 1)
```

### 7.6.7 Right logical left

Logic shifts the contents of var value to the right by *n* bits.



The syntax:

```
SHLR(<variable>,<bits>)
```

Example:

```
glValue = SHLR(glValue, 1)
```

### 7.6.8 Multiplication and division

Perform a 32 bit integer values multiplication with 64 bit result and a successive division by 32 bit integer value. Result is a 32 bit value.
**N.B.** If a division by zero occur, CPU state will set to “Division by zero error”.
The syntax:

```
MULDIV(<factor1>,<factor2>,<divisor>)
```

Example: glValue = a * b / c

```
glValue = MULDIV(a, b, c)
```

### 7.6.9 Remainder of multiplication and division

Perform a 32 bit integer values multiplication with 64 bit result and a successive division by 32 bit integer value. Result is the division remainder.
**N.B.** If a division by zero occur, CPU state will set to “Division by zero error”.
The syntax:

```
RMULDIV(<factor1>,<factor2>,<divisor>)
```

Example: glValue = a * b % c

```
glValue = RMULDIV(a, b, c)
```

### 7.6.10 Nearest integer rounding

| | |
|---|---|
| ROUND | Rounds to nearest integer, rounding away from zero in halfway cases. |
| TRUNC | Rounds to nearest integer not greater in magnitude than the given value. |
| FLOOR | Computes largest integer not greater than the given value. |
| CEIL | Computes smallest integer not less than the given value. |

The syntax:

```
ROUND(<variable>)
TRUNC(<variable>)
FLOOR(<variable>)
CEIL(<variable>)
```

Example:

```
gsValue = ROUND(2.7)     ;risults 3.0
gsValue = ROUND(-2.7)    ;risults -3.0
gsValue = TRUNC(2.7)     ;risults 2.0
gsValue = TRUNC(-2.7)    ;risults -2.0
gsValue = FLOOR(2.7)     ;risults 2.0
gsValue = FLOOR(-2.7)    ;risults -3.0
gsValue = CEIL(2.7)      ;risults 3.0
gsValue = CEIL(-2.7)     ;risults -2.0
```

### 7.6.11 Classification operators

| | |
|---|---|
| ISFINITE | Checks if the given number has finite value. |
| ISINF | Checks if the given number is infinite. |
| ISNAN | Checks if the given number is NaN. |
| ISNORMAL | Checks if the given number is normal. |

Syntax:

```
ISFINITE(<variable>)
ISINF(<variable>)
ISNAN(<variable>)
ISNORMAL(<variable>)
```

Example:

```
gfValue = ISFINITE(gsValue)     ;returns 1 if the number is finite otherwise 0
gfValue = ISINF(gsValue)        ;returns 1 if the number is infinite otherwise 0
gfValue = ISNAN(gsValue)        ;returns 1 if the number is NaN (Not a Number) otherwise 0
gfValue = ISNORMAL(gsValue)     ;returns 1 if the number is normal (not zero, not infinite, not NaN) otherwise 0
```

## 7.7 Trigonometry Functions

### 7.7.1 Sine

Calculation of the sine of an angle expressed in radians. The syntax:

```
SIN(<angle>)
```

Example: gsYPos = ssRadius x SIN gsalpha

```
gsYPos = ssRadius * SIN(gsalpha)
```

### 7.7.2 Cosine

Calculation of the cosine of an angle expressed in radians. The syntax:

```
COS(<angle>)
```

Example: gsXPos = ssRadius x COS gsalpha

```
gsXPos = ssRadius * COS(gsalpha)
```

### 7.7.3 Tangent

Calculation of the tangent of an angle expressed in radians. The syntax:

```
  TAN(<angle>)
```

Example: gsMyVal = TAN gsalpha

```
  gsMyVal = TAN(gsalpha)
```

### 7.7.4 Cotangent

Calculation of the cotangent of an angle expressed in radians. The syntax:

```
  COT(<angle>)
```

Example: gsMyVal = COT gsalpha

```
  gsMyVal = COTG(gsalpha)
```

### 7.7.5 Arcsine

Calculation of an angle, expressed in radians, where the sine is equal to the argument. The syntax:

```
  ASIN(<arc>)
```

Example: gsAngle = ASIN Arc

```
 gsAngle = ASIN(Arc)
```

### 7.7.6 Arcosine

Calculation of an angle, expressed in radians, where the cosine is equal to the argument. The syntax:

```
  ACOS(<arc>)
```

Example: gsAngle = ACOS Arc

```
  gsAngle = ACOS(Arc)
```

### 7.7.7 Arctangent

Calculation of an angle, expressed in radians, where the tangent is equal to the argument. The syntax:

```
  ATAN(arc)
```

Example: gsMyVal = ATAN Arc

```
  gsMyVal = ATAN(Arc)
```

### 7.7.8 Caution

In trigonometry functions the angles are normally expressed in radians. Since p is an irrational number (not finite) and since it is represented with a precision of 7 decimal points, this introduces an approximation in trigonometric calculations.
Another limit is with trigonometric calculations that give the result as an infinite number, which cannot be represented by a single precision floating point number (7 digit). For example the result of the tangent of p/2 is an enormous number in negative, which is not infinite and so incorrect.

Example: tang p/2 = tang 1.570796371 = -22877332 ;
arctg -22877332 = -1.570796371 = - p/2 (!!)

The tangent of p/2 cannot be calculated by the equation:
tang p/2 = (sin p/2) / (cos p/2)
since it has a division by zero.

# 8. Demo Application

The development of this application implements the information illustrated in this manual.
The project includes:

- a unit for the machine state management
- a unit to create a sequencer
- a unit to detect digital inputs and activate digital outputs.

## 8.1 New Project

Create a new project with

- File - New Project

enter the *Project Name* as *MyApplication*.

## 8.2 Add the configuration unit

Select

- File - Add Unit - Configuration Unit

and name it CONFIG. The unit will be precompiled with all the keywords that distinguish the various variable groups. At this stage it is essential

to declare the hardware that is going to be used in the configuration unit.
The demo uses

```
...
BUS
  1  1P51F  30
  2  .      .
  3  1MG8F  .
```

Nothing else is declared in the configuration unit.

## 8.3 Add the QCL task units

### 8.3.1 MANAGER Unit

Select

- File - Add Unit - Qcl Unit

and name it *MANAGER*. Leave the ***Task Mode*** property as *Normal*. The new unit has the following default code lines:

```
MAIN:

  WAIT 1
  JUMP MAIN
END
```

The *MANAGER* unit manages the machine states shown in the diagram below:

The *MANAGER* unit code lines will manage the state transitions and supply information (e.g. the current state).
Declare the following variables in the *MANAGER* unit:

```
CONST
  ALARM_STATE      0  OUT
  MANUAL_STATE     1  OUT
  AUTOMATIC_STATE  2  OUT

GLOBAL
  NewState  B  IN    ;New state requested
  State     B  OUT   ;Actual state

BEGIN
MAIN:

  WAIT 1
  JUMP MAIN
END
```

Now add the following code:

```
CONST
  ALARM_STATE      0  OUT
  MANUAL_STATE     1  OUT
  AUTOMATIC_STATE  2  OUT

GLOBAL
  NewState  B  IN    ;New state requested
  State     B  OUT   ;Actual state

BEGIN
MAIN:

  SWITCH State
  CASE ALARM_STATE
    ;Put the code to do when the state is ALARM
  CASE MANUAL_STATE
    ;Put here the code to do when the state is MANUAL
  CASE AUTOMATIC_STATE
    ;Put here the code to do when the state is AUTOMATIC
  ENDSWITCH

  WAIT 1
  JUMP MAIN
END
```

The *State* variable is considered an OUT for this unit. If another unit wants to change machine state it cannot act directly on *State* because

only the *MANAGER* unit can change it. So a variabile *NewState* variable is introduced that is used by *MANAGER* as an IN as follows

```
CONST
  ALARM_STATE      0  OUT
  MANUAL_STATE     1  OUT
  AUTOMATIC_STATE  2  OUT

GLOBAL
  NewState  B  IN    ;New state requested
  State     B  OUT   ;Actual state

BEGIN
MAIN:

  SWITCH State
  CASE ALARM_STATE
    ;Put here the code to do when the state is ALARM
  CASE MANUAL_STATE
    ;Put here the code to do when the state is MANUAL
  CASE AUTOMATIC_STATE
    ;Put here the code to do when the state is AUTOMATIC
  ENDSWITCH

  ;Check if there is a state transition request
  IF State NEQ NewState
    SWITCH NewState
    CASE ALARM_STATE

      SWITCH State
      CASE MANUAL_STATE
        ;Put here the code for MANUAL to ALARM trasition
      CASE AUTOMATIC_STATE
        ;Put here the code for AUTO to ALARM transition
      ENDSWITCH

    CASE MANUAL_STATE

      SWITCH State
      CASE ALARM_STATE
        ;Put here the code for ALARM to MANUAL transition
      CASE AUTOMATIC_STATE
        ;Put here the code for AUTO to MANUAL transition
      ENDSWITCH

    CASE AUTOMATIC_STATE

      SWITCH State
      CASE MANUAL_STATE
        ;Put here the code for MANUAL to AUTO transition
      ENDSWITCH

    ENDSWITCH

    State = NewState  ;The transition is done

  ENDIF

  WAIT 1
  JUMP MAIN

END
```

It can be seen that, when *NewState* is different to *State*, one of the codes associated to one of the 5 possible transitions is executed. At

the end of the transition, the machine state is conclusively set to the new status.
So any unit that wants to use the MANAGER unit must use this kind of code:

```
  MANAGER.NewState = MANAGER.AUTOMATIC_STATE
  WAIT MANAGER.State EQ MANAGER.NewState
```

To command the state transition.

### 8.3.2 SEQUENC Unit

Add another unit with the same properties as *MANAGER* and name it *SEQUENC*. To obtain a sequence write the following code:

```
GLOBAL
  Start           F  IN   ;Start sequence input
  Sequence        W  OUT  ;This is the step of the sequence

  start_rise_up   F  ;Local variable to check the rise edge of Start input

BEGIN
MAIN:

  IF Start
    IF start_rise_up
      start_rise_up = 0
      Sequence = 10        ;Start the sequence
    ENDIF
  ELSE
    start_rise_up = 1
    Sequence = 0           ;Reset the sequence
  ENDIF

  WAIT 1
  JUMP MAIN
END
```

It can be seen that, when the *Start* IN variable is 0 (FALSE), the *Sequence* variable is zero-set (this is to prevent the sequence from working). When the *Start* variable is 1 (TRUE), the *Sequence* variable is set to 10 only the first time, because the *start_rise_up* variable is set to 0.
Now add the code lines to use the *Sequence* variable.

```
CONST
  OFF  0
  ON   1

GLOBAL
  Start           F  IN   ;Start sequence input
  Force           F  IN   ;Force the signal output to ON
  Sequence        W  OUT  ;This is the step of the sequence
  Signal          F  OUT  ;This is an output signal

  start_rise_up   F  ;Local variable to check the rise edge of Start input

TIMER
  time_on
  time_off

BEGIN
MAIN:

  IF Start
    IF start_rise_up
      start_rise_up = 0
      Sequence = 10        ;Start the sequence
    ENDIF
  ELSE
    start_rise_up = 1
    Sequence = 0           ;Reset the sequence
    Signal = Force         ;Force the signal
  ENDIF

  ;This is the sequence management
  IF Sequence EQ 10        ;First step
    Signal = OFF           ;Reset signal
    time_off = 1000        ;1000 ms = 1 s
    Sequence = 20
  ENDIF
  IF Sequence EQ 20        ;Second step
    IF time_off            ;Wait time expired
      Sequence = 30
    ENDIF
  ENDIF
  IF Sequence EQ 30        ;3rd step
    Signal = ON            ;Set signal
    time_on = 500          ;500 ms = 0.5 s
    Sequence = 40
  ENDIF
  IF Sequence EQ 40        ;4th step
    IF time_on             ;wait time expired
      Sequence = 50
    ENDIF
  ENDIF
  IF Sequence EQ 50        ;5th step
    Sequence = 10
  ENDIF

  WAIT 1
  JUMP MAIN
END
```

When the *Sequence* variable is set to 10, the following code is executed:

```
  IF Sequence EQ 10
    ...
```

that changes it to 20 and so forth executing the sequence step by step. The sequence sets the *Signal* variable to OFF for 1s and then to ON for

0.5s and the sequence is repeated infinitely, so long as *Start* is TRUE).

### 8.3.3 COMMAND Unit

Add the COMMAND unit and enter the following code:

```
GLOBAL
```

```
  ForceOutput    F  OUT

INPUT
  Selector       F  REFERENCE
  Enable         F  REFERENCE
  ManualButton   F  REFERENCE

BEGIN
MAIN:

  WAIT 1
  JUMP MAIN
END
```

Three *REFERENCE*'s are declared to three digital inputs. See below how to assign these *REFERENCE*'s to three digital inputs that will be

added to the configuration unit. Now add the following code:

```
GLOBAL
  ForceOutput    F  OUT

INPUT
  Selector       F  REFERENCE
  Enable         F  REFERENCE
  ManualButton   F  REFERENCE

BEGIN
MAIN:

  IF Enable  ;If enable input is on
    IF Selector  ;If selector in on automatic position
      ;Go to automatic state
      MANAGER.NewState = MANAGER.AUTOMATIC_STATE
      WAIT MANAGER.State EQ MANAGER.AUTOMATIC_STATE
    ELSE
      ;Go to manual state
      MANAGER.NewState = MANAGER.MANUAL_STATE
      WAIT MANAGER.State EQ MANAGER.MANUAL_STATE
    ENDIF
  ELSE
    ;Go to alarm state
    MANAGER.NewState = MANAGER.ALARM_STATE
    WAIT MANAGER.State EQ MANAGER.ALARM_STATE
  ENDIF

  ForceOutput = ManualButton  ;Replay  the input value to an output variable

  WAIT 1
  JUMP MAIN
END
```

This code is used to decide when to execution the state transitions. Now the code written in *MANAGER* has to be completed as follows:

```
CONST
  ALARM_STATE      0  OUT
  MANUAL_STATE     1  OUT
  AUTOMATIC_STATE  2  OUT

GLOBAL
  NewState  B  IN    ;New state requested
  State     B  OUT   ;Actual state

BEGIN
MAIN:

  SWITCH State
  CASE ALARM_STATE
    ;Put here the code to do when the state is ALARM
    SEQUENC.Start = SEQUENC.OFF
    SEQUENC.Force = SEQUENC.OFF
  CASE MANUAL_STATE
    ;Put here the code to do when the state is MANUAL
    SEQUENC.Force = COMMAND.ForceOutput
  CASE AUTOMATIC_STATE
    ;Put here the code to do when the state is AUTOMATIC
    SEQUENC.Start = SEQUENC.ON
  ENDSWITCH

  ;Check if there is a state transition request
  IF State NEQ NewState
    SWITCH NewState
    CASE ALARM_STATE

      SWITCH State
      CASE MANUAL_STATE
        ;Put here the code for MANUAL to ALARM trasition
        SEQUENC.Start = SEQUENC.OFF
        SEQUENC.Force = SEQUENC.OFF
      CASE AUTOMATIC_STATE
        ;Put here the code for AUTO to ALARM trasition
        SEQUENC.Start = SEQUENC.OFF
        SEQUENC.Force = SEQUENC.OFF
      ENDSWITCH

    CASE MANUAL_STATE

      SWITCH State
      CASE ALARM_STATE
        ;Put here the code for ALARM to MANUAL trasition
        SEQUENC.Start = SEQUENC.OFF
        SEQUENC.Force = SEQUENC.OFF
      CASE AUTOMATIC_STATE
        ;Put here the code for AUTO to MANUAL trasition
        SEQUENC.Start = SEQUENC.OFF
        SEQUENC.Force = SEQUENC.OFF
      ENDSWITCH

    CASE AUTOMATIC_STATE

      SWITCH State
      CASE MANUAL_STATE
        ;Put here the code for MANUAL to AUTO trasition
        SEQUENC.Start = SEQUENC.OFF
        SEQUENC.Force = SEQUENC.OFF
      ENDSWITCH
    ENDSWITCH

    State = NewState  ;The trasition is done

  ENDIF

  WAIT 1
  JUMP MAIN

END
```

This simple examle shows how all the operations are executed during the state transitions are the same and block any operation on the *Signal*

variable. It can be noted that when in the manual state, activation of *Signal* is assigned to *COMMAND.ForceOutput* that in turn has the same

value as a digital input. In the automatic state the sequence managed in the *SEQUENC* unit is activated.
The last addition to the *CONFIG* configuration unit is made as follows:

```
INPUT
  ManAuto   F    3.INP01
  PowerOn   F    3.INP02
  Push          F   3.INP03

REFERENCES
  COMMAND.Selector     = ManAuto
  COMMAND.Enable       = PowerOn
  COMMAND.ManualButton = Push
```

In this way the symbols of the *COMMAND* unit: *Selector*, *Enable*, *ManualButton* are references associated to the inputs declared as:

*ManAuto*, *PowerOn*, *Push* respectively. It is evident that in this example the REFERENCE's functionalities are of little use. In effect, instead

of the *Selector* symbol, *ManAuto* could have been used directly. The REFERENCE's potential can be appreciated when reusing the code

written for the *COMMAND* unit. A new unit could be added, named *COMMAN2* that contains exactly the same code as *COMMAND* but with

REFERENCE's targetting other inputs. This would obtain a perfectly useful code.

# 9. The QCL function library

A QCL function is part of a code that allows to solve particular situations, perform data processing or provide a QMOVE project with specific functionalities. To use a QCL function in a unit code it just has to be called and its arguments completed, as described below.
This programming method offers several advantages:

- They are parts of code ready written and tested, which offer optimised solutions to the most common situations faced when drawing up a project.
- They offer the possibility of using the same function several times without having to write it every time.
- The library can be updated any time new functions are added, maintaining full compatibility with existing applications.

The list of QCL functions and their use is available in the ***Help > Functions info*** menu, which can be opened anywhere in the development environment.

To recall any library function indicated by *FuncQCL01*, use the following syntax :

```
  FuncQcl01(<arguments list>)
```

Since functions cannot return values, they cannot be used to the right of an assignment or as part of an expression in an IF, FOR or WHILE instruction.

```
slType01 = FuncQCL01(...)                       !COMPILING ERROR!
IF (FuncQCl01(...) AND ...)                     !COMPILING ERROR!
FOR (sInt01 = 1, sInt01 LT FuncQCL01(...), ...) !COMPILING ERROR!
WHILE (FuncQCL01(...) LT 24)                    !COMPILING ERROR!
```

Even if a QCL function does not return a value, a QCL function can set values to past parameters as an argument and so in practice it has the effect of returning several results.

Some simple examples:

```
  AC10AvergArr (MyArray, average_value, ok_calc)
```

The function calculates the average value of an array supplied as argument. In this case *MyArray* is an input parameter and the function calculates its average value.
*average_value* and *ok_calc* are output parameters and are the average value calculated by the function and a flag indicating that calculation is complete. When *ok_calc* takes on value 1 the function has finished calculating and therefore the value can be recovered by reading the *average_value* variable.

In what part of the unit a function must be called is indicated. In fact there are functions, because of the actions they have to perform, that have to be called in a cyclic part of the project, while others can also be called in a non-cyclic part of the project. Cyclic is intended as part of the QCL code that is executed at every program execution.

Non-cyclic is intended as part of the code that is not executed at every program execution because they lack certain conditions (e.g. if the code is contained in an IF instruction with a false condition).
Indications on “where” to recall a function are provided in the description of each function (***Help > Functions info***).
Another characteristic to remember for other functions is that, when actions require long execution times, the function executes a WAIT instruction every 180 milliseconds. This prevents unit executions being “blocked” by particularly heavy actions. This kind of function normally has a flag type argument that can be monitored to verify when the function has completed its action.
The function help clearly indicates the types of argument that the function expects. If the programmer does not respect these types an error is generated during the project compilation.

### 9.0.1 Practical example of QCL functions

The use of the arithmetic average calculation function on the values in an array (AC10AvergArr):

```
GLOBAL
  glAverage L           ; This is used as the result variable
  gfDone F              ; This flag is used to result variable
ARRSYS
  aslMyArray 10 L       ; Array System

BEGIN

MAIN:
  IF NOT gfDone
    AC10AvergArr(aslMyArray, glAverage, gfDone)
  ENDIF
  WAIT 1
  JUMP MAIN
END
```

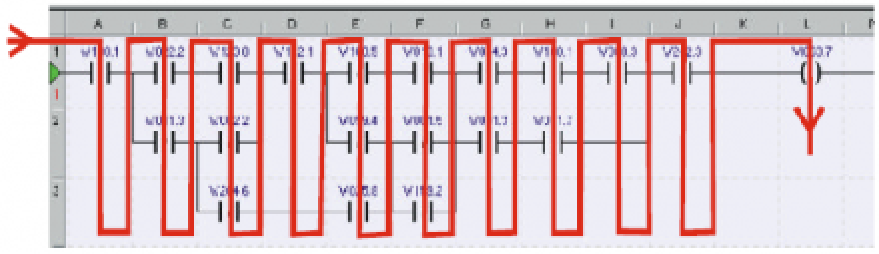
# 10. Editor Ladder

Ladder (contact logic) is a language conceived for PLC programming, where it offers very good performance in managing inputs, outputs and short sequences.
It is presumed that the user knows the IEC1131 Ladder language and if this is not the case it is advisable to integrate this manual with a general course in PLC.

## 10.1 Execution of the rung

In LADDER it must be remembered that the various elements inside a rung are analysed starting from the top left-hand corner and moving up to down and left to right (Figure 1).

Figure 1: execution of the rung.

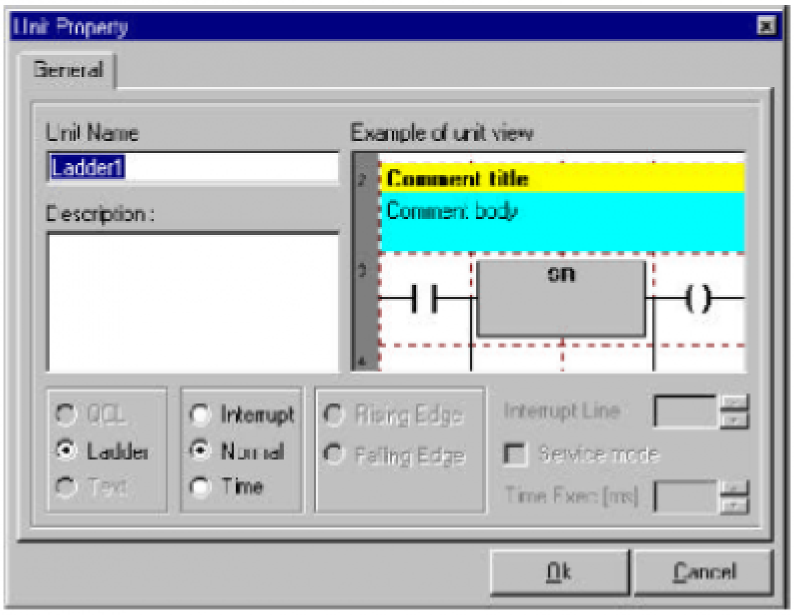


If an element that takes up several cells (e.g. a counter) is inserted, the top left-hand cell is referred to when executing the rung.
The name of the variables for the Ladder program are declared in the configuration unit and the same as the QCL section. The same variables can be used in Ladder and QCL, even simultaneously.
An element is inserted in the LADDER grid using the menu functions is described below.

## 10.2 How to insert an element

First of all a new LADDER unit has to be inserted by selecting ***File > Add unit > Ladder unit***. The window in Figure 2 will open.

Figure 2: inserting a new LADDER unit.



Unit Property

General

Unit Name: Ladder1

Description:

Example of unit view

( ) QCL [✓ Ladder] ( ) Text

( ) Interrupt [✓ Normal] ( ) Time

( ) Rising Edge ( ) Falling Edge

Interrupt Line

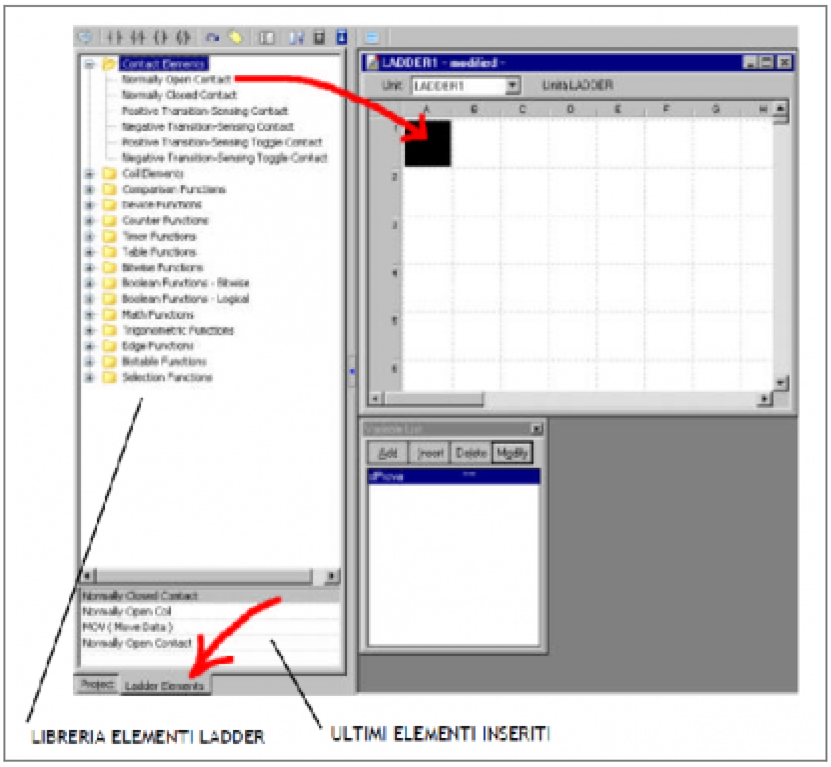
[ ] Service mode

Time Exec [ms]

Ok Cancel

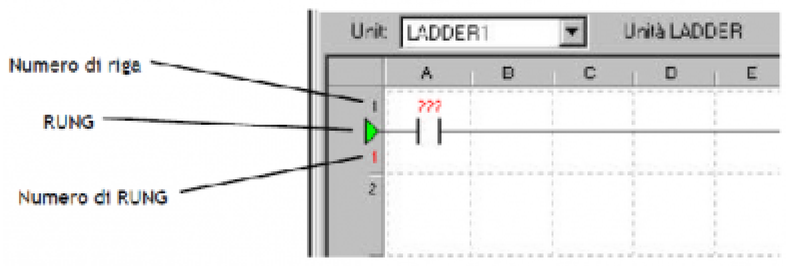
Confirm with *Ok* and a new unit is added in the **UNIT MANAGER** window. Double click on the new LADDER unit to open the LADDER editor. Now the first element of this LADDER grid can be inserted using the “LADDER library” as shown in figure 3.

Figure 3: LADDER Library



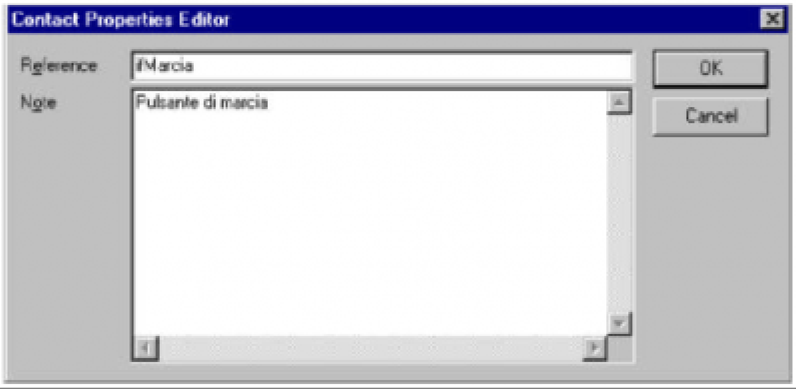
At this stage the various elements available can be selected and placed on the editor grid. The element can be placed by dragging it to its position, or by double clicking on the element, which positions the element in the highlighted editor box.
The ladder element can also be selected from the list of “latest elements inserted”.
The ladder element must be inserted in an existing grid. If the grid does not exist, it has to be created by inserting a new rung.
To insert a new “rung” select menu ***Edit > New Rung***.
For instance, to insert a normally open contact, select the Normally Open Contact and the symbol of the contact will appear on the editor (Figure 4).

Figure 4: normally open contact in a LADDER grid.



The LADDER library has a series of categories dividing the elements used in a LADDER grid, which will be viewed briefly below. Remember that most of the elements are standard (IEC1131) and all have an in-line help that can be opened in Qview by selecting an element and pressing F1.
The element inserted in figure 4 is not associated to a variable. To associate a variable highlight the element and select the menu ***Edit > Element Properties***. A window will open as in figure 5.

Figure 5: block properties.

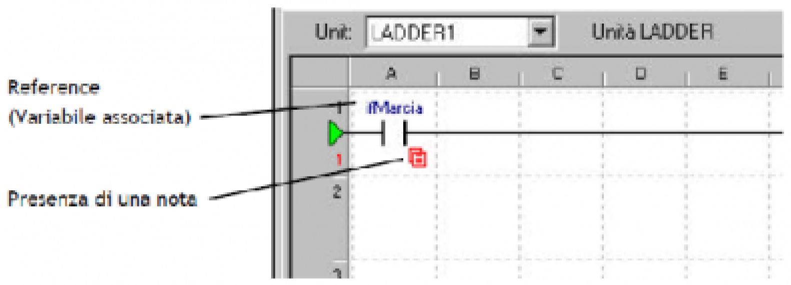

Contact Properties Editor

Reference: iMarcia

Note: Pulsante di marcia

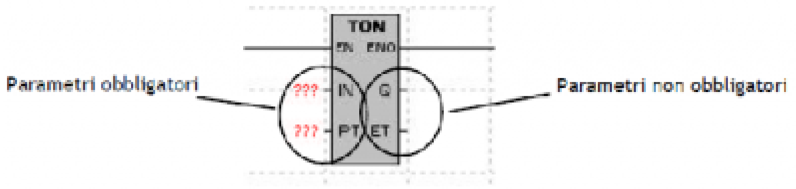
The “Note” field is used to associate useful comments about the element for the programmer. When the window has been filled in as in figure 5, the LADDER grid will appear as in figure 6.

Figure 6: block properties.



The element properties window is used to enter all the variables required by the element. However some variables are optional. For example. figure 7 shows that some parameters of the **TON (On Delay)** element are not mandatory and this is indicated by the lack of “???” (i.e. 3 question marks).

Figure 7: TON: mandatory and optional parameters.



## 10.3 Element Categories

The LADDER element categories for the LADDER grid:

| |
|---|
| Contact Elements |
| Coil Elements |
| Comparison Function |
| Device Functions |
| Counter Functions |
| Timer Functions |
| Table functions |
| Bitwise Functions |
| Boolean Functions - Bitwise |
| Boolean Functions - Logical |
| Math Functions |
| Trigonometric Functions |
| Edge Functions |
| Bistable Functions |
| Selection Functions |

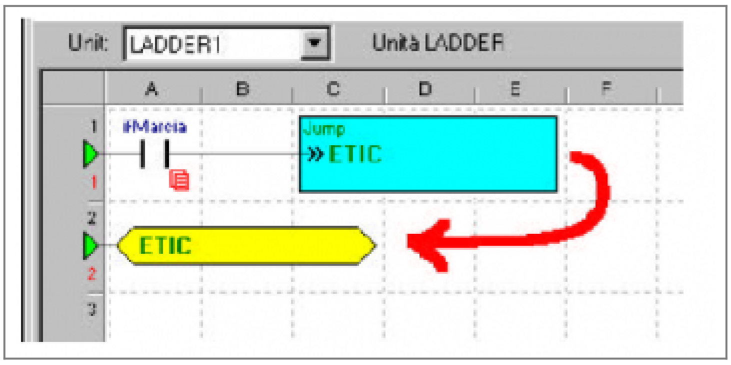
## 10.4 Comments

The LADDER grid can accept a row of comment boxes. To enter a comment select the menu ***Edit > New Element > Comment***.
Each comment can have a title and a text.

## 10.5 Jump and Label

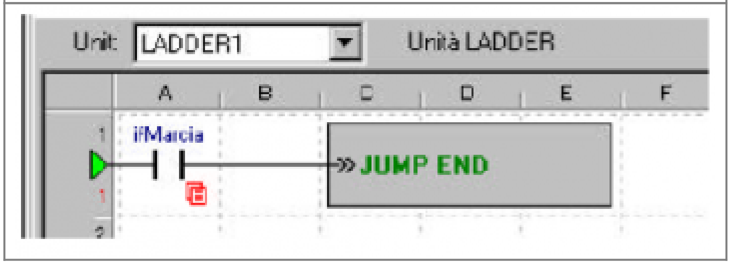
The Ladder structure provides the possibility to jump to a label so the program can skip heavy parts of the program when there is no need to execute them. To insert a jump element select the menu ***Edit > New Element > Jump***. In the properties of this element specify the label associated to the destination rung. To enter the target label select the menu ***Edit > New Element > Label***. Two labels cannot have same name in a unit (Figure 8).

Figure 8: (JUMP) to (LABEL) ETIC.



“Jump end” element can be inserted to simple jump directly to the end of the unit without a label (Figure 9).

Figure 9: JUMP END.



## 10.6 Moving LADDER rows

The Ladder structure can be changed by move parts of a program up or down to add new program lines. To move a row select a rung and select the menu ***Edit > Move Rows Up*** or ***Edit > Move Rows Down***.
When the ladder unit has ended, the program lines can be compacted to remove any empty spaces by the command ***Edit > Compact Rows***.

## 10.7 Drag & Drop

One or more ladder elements can be selected and then dragged by mouse and then dropped in another position. When the elements are selected keep the left mouse button pressed on the selection, which changes colour and the cursor changes shape, so that it can be moved. When dragging the selection the bottom status bar shows the message:: “Placing mode for drag and drop action”. As with the *Paste* command, the following situations can be found:

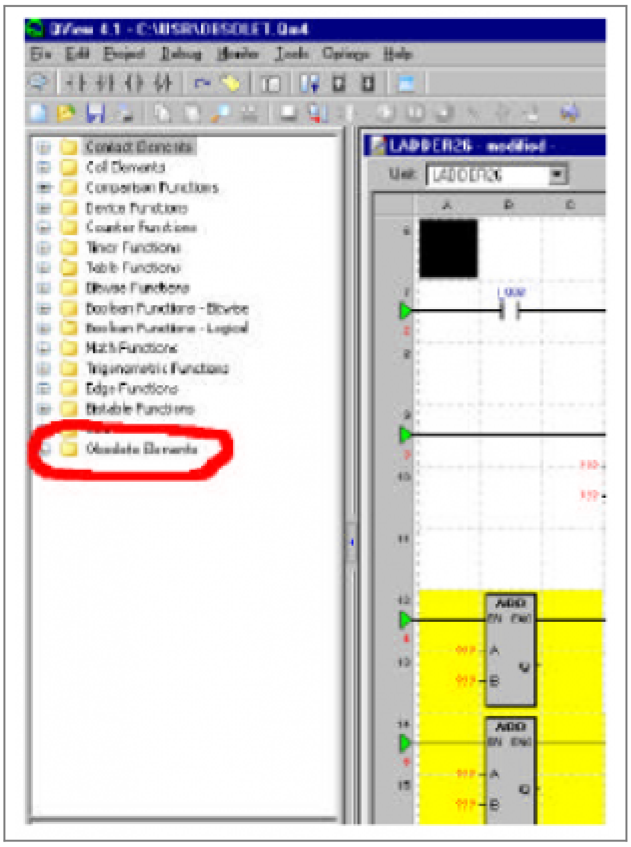
- Drag & Drop in an empty area with no other ladder elements:
Release the mouse button to place it in the box, the coloured box disappears and the selected elements are inserted in the editor.

- Drag & Drop over or in an area with other ladder elements:
as above, when the box is placed and the mouse button release, the coloured box disappears and a message box asks to confirm or cancel the operation. Confirm the operation to insert the elements in the box below, automatically creating all the connections needed for a correct compilation of the end result.

## 10.8 Obsolete LADDER elements

**Obsolete elements**, are old ladder elements that Qem has then made obsolete during Ladder library updates and maintenance. A ladder element is usually marked obsolete if it has been substituted by an updated version, which normally increases efficiency of the internal code. The obsolete element can always be replaced by an equivalent element in the ladder library in use. A project using obsolete ladder elements highlights them by giving them a coloured background (yellow by default). The obsolete elements in a program are easily found by ***Edit > New Elements > Obsolete Elements*** and in the “Obsolete Elements” folder of the element library (Figure 10).

Figure 10: Obsolete Elements folder



In ***Options*** **>** ***Program Setup…*** **>** ***Ladder Editor*** modify the default settings for obsolete ladder elements.

## 10.9 Substitution of obsolete elements

The obsolete element substitution function (**Substitute Obsolete element**) has been provided to replace an obsolete element in the ladder grid with its corresponding updated element.
Go to Menu > ***Edit*** **>** ***Substitute Obsolete Element***. This function is activated if the ladder editor cursor is positioned over a replaceable obsolete element.
When the obsolete elements are substituted they change background colour to the standard background for valid elements.
Use Menu > ***Edit*** **>** ***Substitute All Obsolete Elements*** to replace all the ladder elements in a project by one passage. When this procedure is started it first finds all the obsolete ladder elements and gives the message:

**Searches for obsolete elements in progress…**

This message is followed by:

**No obsolete elements to substitute**

if there are no obsolete elements, or

**There are obsolete ladder elements…**

At this stage either select to start substitute all or quit the procedure. The complete substitution is signalled by the message:

**All obsolete elements are substituted!**

# 11. Multitasking

Multitasking is the capacity of a system to manage simultaneous execution of several tasks.
A QCL and Ladder project is made up of a series of task units. The multitasking implemented in the QMOVE system is a so-called *cooperative* type, in other words the execution of instructions passes from one task unit to another only when, in the task being executed, a specific instruction is found

```
WAIT < condition >.
```

This is true for task units written in QCL, while tasks developped in ladder pass onto the next task after the last instruction. The CPU stops executing the code of a task unit when the task is put in standby by the WAIT instruction.
A flow chart of the so-called *task cycle* is shown in figure 1. So each task of a project has to be able to *cooperate* with the other tasks by entering a wait status when a condition is met and so, in the meantime pass onto executing the other task units. For instance, a task that has to carry out an axis positioning has to wait for the axis to reach the target position before continuing with the other operations. So meanwhile the CPU can execute other instructions programmed in other task units.
On startup the execution order of the units is the same as the order declared in the UNIT MANAGER window in QVIEW, except that the first task executed is the second in the list and then contining down the last and the first in list is the last task.

Figure 1: task execution (initial condition).

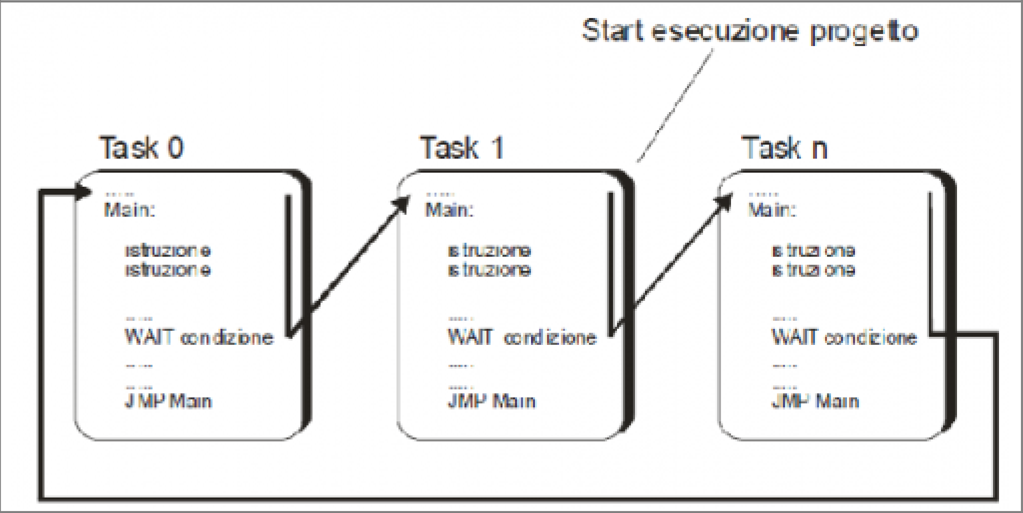
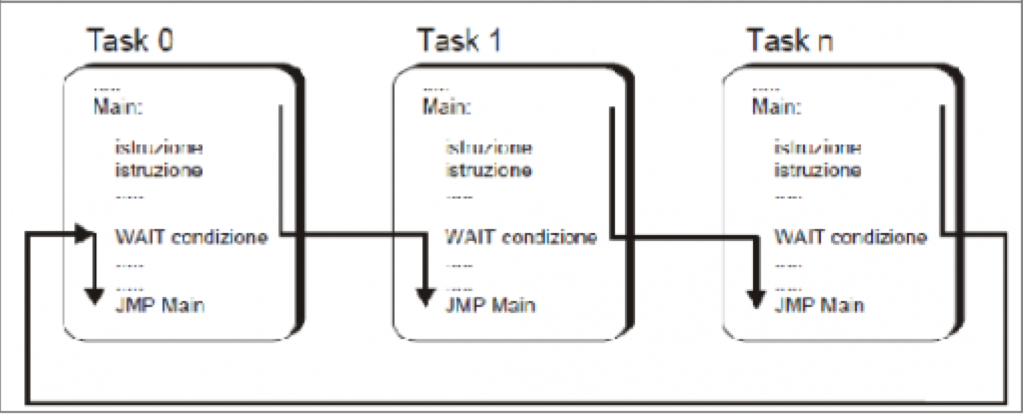


Figure 2: task execution (flow chart).

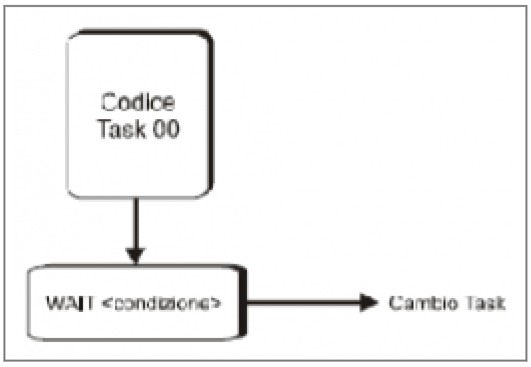


In addition to executing the code lines, the CPU also simultaneously executes the devices. Devices are explained in their specific chapter, however they can be generally described as tools that provide functionalities called by QCL or Ladder code lines, to achieve the most common applications met in industrial automation.

## 11.1 The Standard structure of a QCL task

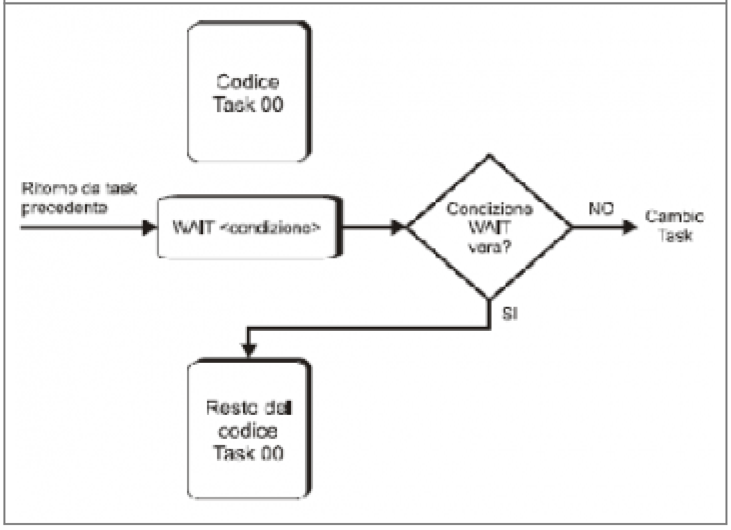
All tasks start their execution from the first code line. Control then passes onto the next task when a WAIT instruction is met (Figure 3).

Figure 3: program flow (initial condition).



When control returns to the task in question, it restores execution of the code from the point where it was interrupted only if the condition to the right of the WAIT keyword is true, otherwise it executes a new task change, as represented in figure 4.

Figure 4: program flow (change task condition).



So if a task is in WAIT and its condition is never met, the code line following the WAIT is never executed.
The QCL code is executed through to the end in every unit. To ensure that the code is repeated continuously, an unconditioned jump from end or start has to be inserted. Normally every task has the following minimum QCL code:

```
[< declaration code >
BEGIN]
[< initialization code >]
MAIN:
  < operative code >
  WAIT 1
  JUMP MAIN
END
```

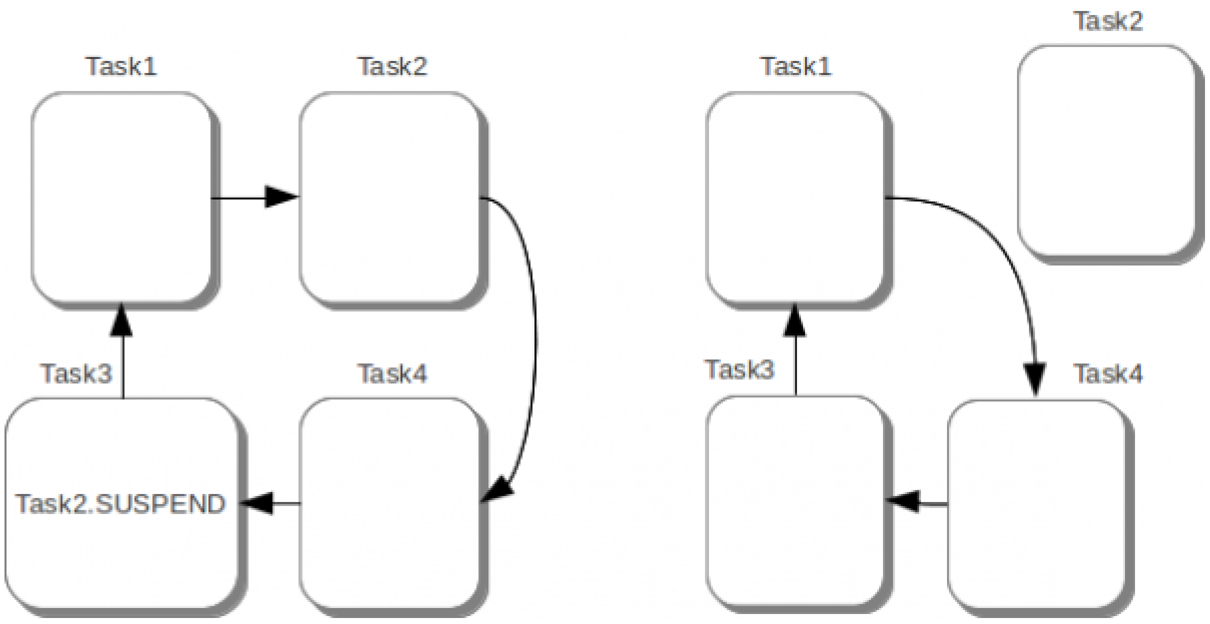
The initialisation code before the MAIN is only executed once during startup, while everything between MAIN and JUMP MAIN is executed cyclically . If a given task does not have the JUMP MAIN instruction,the task would reach the - END - and thereby paralyse the whole multitasking cycle. The CPU indicates that the cycle is perpetually blocked in a task by the *Watchdog Active* signal (in QVIEW ***Monitor > CPU***).
The WAIT 1 instruction is needed to ensure the task change.

### 11.1.1 Example of SUSPEND

If this instruction is inserted in the task3 unit

```
Task2.SUSPEND
```

task2 is suspended and will no longer be executed.

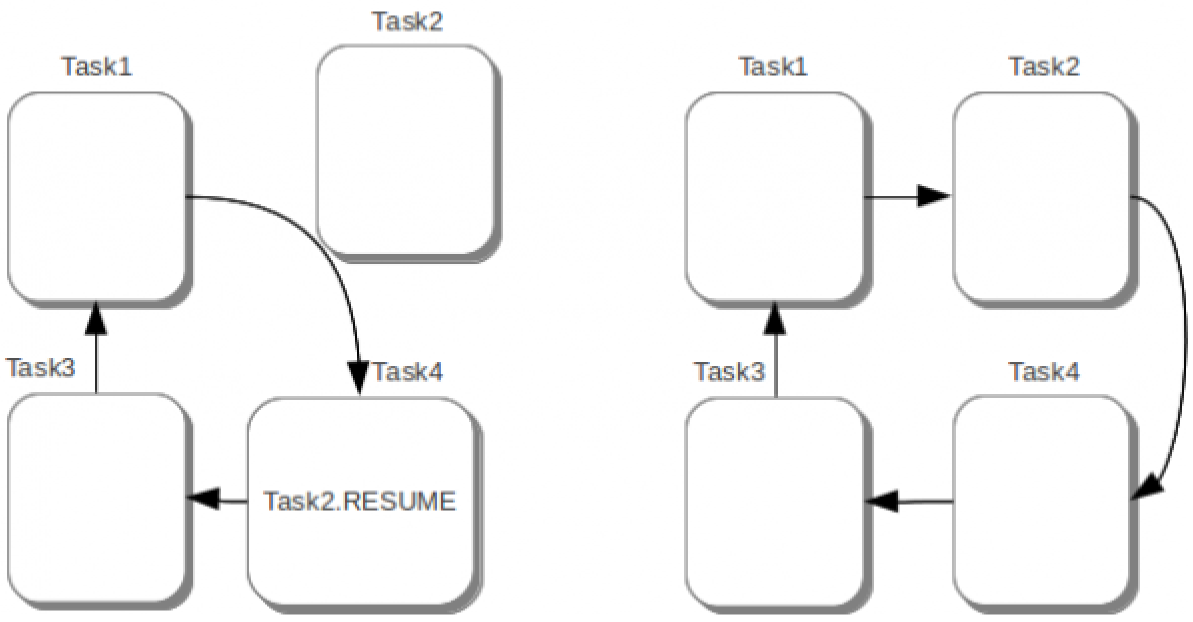


### 11.1.2 Example of RESUME

If the CPU executes this instruction in task4

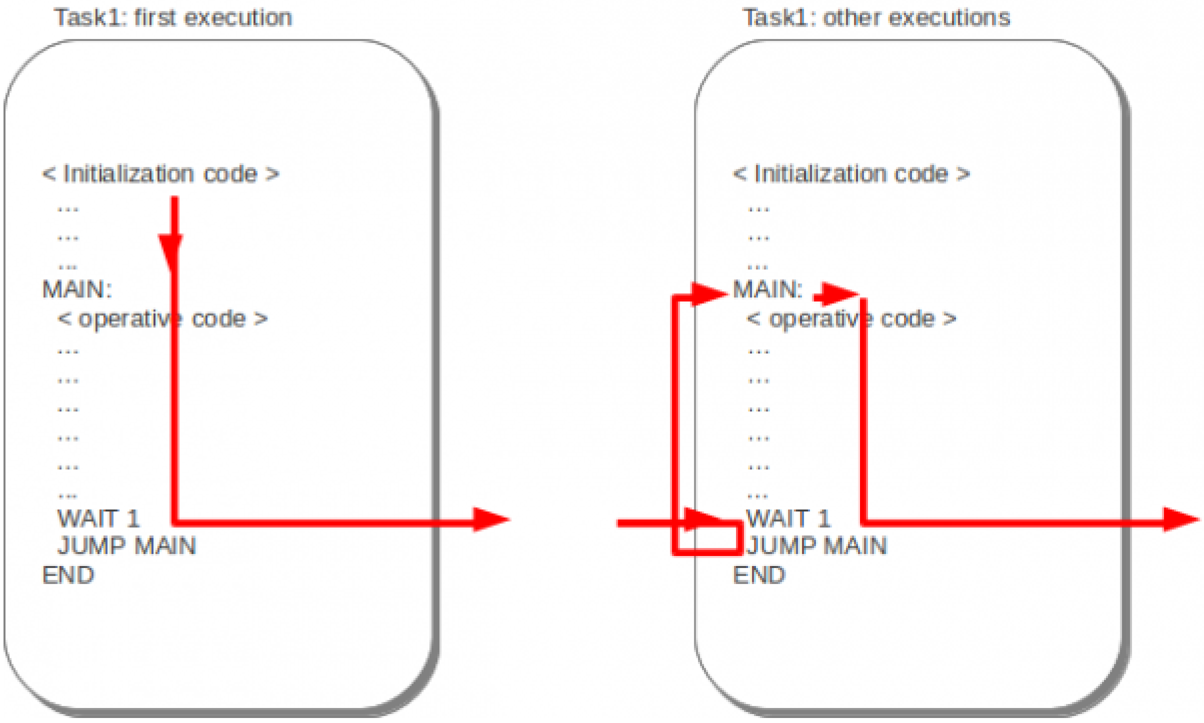
```
task2.RESUME
```

task2 unit enters the multitasking cycle and the code it contains is executed again.



### 11.1.3 Example of RESTART

The RESTART instruction applied to a task unit restarts the execution of a code from the first instruction in the unit. For example the order of execution of instructions contained in a task are represented in the figure below:



If, in the same or another task of the project, the CPU executes the instruction

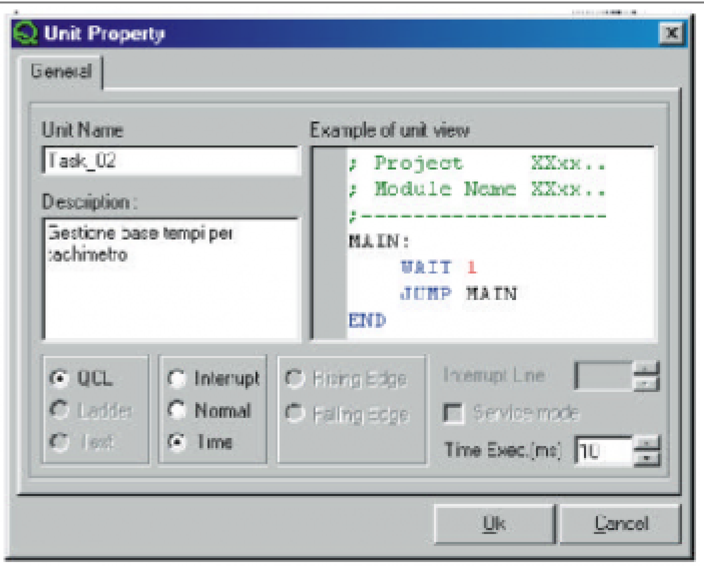
```
task1.RESTART
```

The next execution of a code in task1 restarts as if it were the first execution.

## 11.2 Time task units

When writing a program it may be necessary to manage events at fixed, repeatable times. Since the cooperative multitasking system does not have a repeatable cycle time, time units have been introduced, which can be activated by selecting the specific box in the unit properties and declaring the time base to use.

Figure 1: declaration of a time unit


Unit Property

General

Unit Name: Task_02

Description: Gestione base tempi per tachimetro

Example of unit view

```
; Project      XXxx..
; Module Name  XXxx..
;--------------------
MAIN:
      WAIT 1
      JUMP MAIN
END
```

[✓ QCL] Ladder Text

Interrupt Normal [✓ Time]

Rising Edge Falling Edge

Interrupt Line

Service mode

Time Exec.(ms) 10

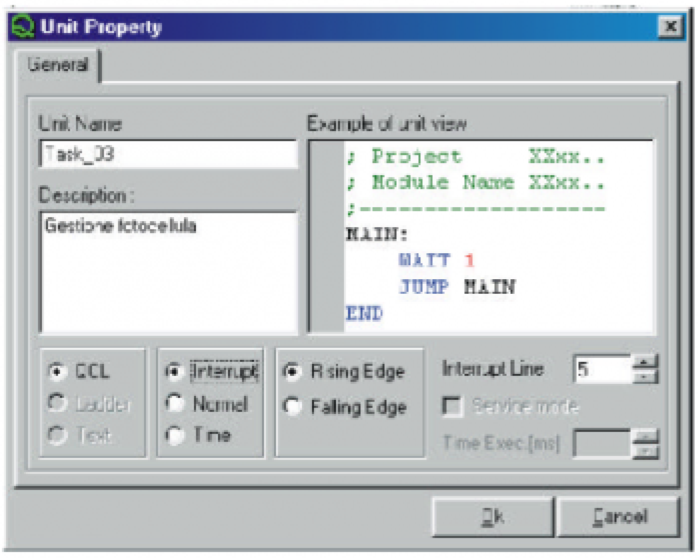
Time units have priority over normal task units. Make sure that the code lines in a time unit (both QCL and

LADDER) do not require more time for execution than the set repeat time.
If this happens some code repetitions are skipped or lost and the CPU signals this event with a “Time task lost” status. This can be seen in the CPU Monitor window. Moreover, type of task unit cannot accept WAIT instructions, so the relevant code lines must be executed in a single cycle.
**A maximum of 7 time units can exist in a project.**

## 11.3 Interrupt task units

When writing a program it may be necessary to manage events created by an interrupt input (e.g. photocells or proximities). Interrupt task units have been introduced to satisfy this need, which can be activated by selecting the specific box in the unit properties and declaring the interrupt line and activation front.

Figure 2: declaration of an interrupt based task



Unit Name: Task_03

Description: Gestione fotocellula

Example of unit view

```
; Project      XXxx..
; Module Name  XXxx..
;-------------------
MAIN:
    WAIT 1
    JUMP MAIN
END
```

[✓ QCL] Ladder Text

[✓ Interrupt] Normal Time

[✓ Rising Edge] Falling Edge

Interrupt Line: 5

Service mode

Time Exec.[ms]

The interrupt units are priority over normal task units and time units. Moreover, type of unit cannot accept WAIT instructions, so the relevant code lines must be executed in a single cycle. To avoid overloading the CPU with interrupts, there is a control to check that until execution of an interrupt code has not been completed, the system does not enable the interrupt line, so it is impossible to build up several interrupts from the same line.
The interrupt units can be written in both QCL and Ladder.
A maximum of 2 interrupt units are allowed for a project.

# 12. Devices

Devices are a category of software tools that perform more or less complex support or control operations that simplify operations and procedures in industrial automation.
For instance, a typical device can manage a CNC type positioner with +/-10V analog output, offering all the functions, commands and parameters needed to manage the positioning correctly (e.g. homing, manual positioning, speed parameters, inversion time, etc.).
Devices have their own functions, variables, commands and parameters. They can be configured and inserted in a project so that one or more devices become an integral part of the application program.

The parameters are variables used to configure the device execution (e.g. positioning speed, count , etc.):

```
  <device name>.<parameter device>
```

The commands make the device perform specific functions (e.g. START, STOP, etc.):

```
  <device name>.<command name>
```

The devices available for any given QMOVE model are listed in the hardware installation and maintenance manuals.

## 12.1 Declaring a Device

**The syntax for declaring a device is given in the specific device documentation.**

The use of a device must be declared in the configuration unit or in the declaration section of each unit. The declaration syntax changes according to the device.

### 12.1.1 Device Declaration

The device declaration in the configuration unit must include the INTDEVICE keyword.
For more details see the specific documentation on each Device.

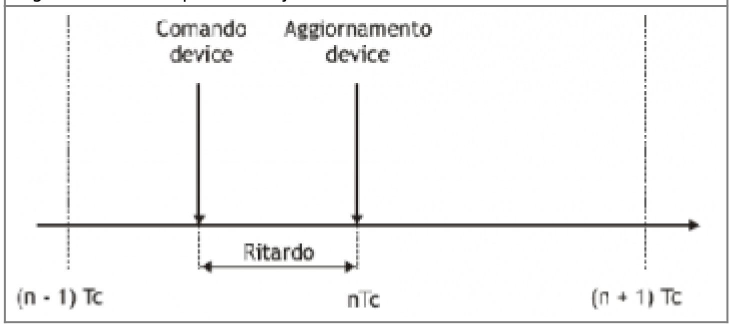
## 12.2 Using devices

This chapter will give a preliminary introduction for the correct use of devices. The information below must be completed with the specific documentation on each device.

### 12.2.1 Sampling time

As explained in Multitasking, one of the special characteristics of devices is the sampling time (Tc) that establishes how often the device is managed by the CPU.
The general criteria in choosing sampling times is that a short time normally executes the device at a high rate so it therefore reacts rapidly to external actions (e.g. better control of an axis). When executing the QCL code project, a write access to the device or a command are processed after a maximum time, know as the sampling time. This means that a lower Tc, gives a shorter delay in executing the device. The diagram in Figure 1 gives an example.

Figure 1: device update delay.

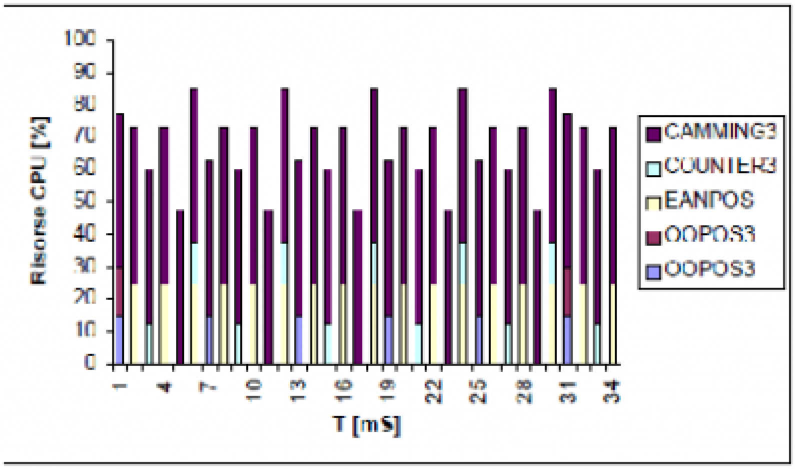


The choice of device sampling times used in a project must account for certain considerations. A natural question is, “Why not select the minimum sampling time for each device?”. In effect this would give maximum performance, however this choice is not always possible.
The CPU allocates a part of its processing resources to managing the devices in a project. Each sampling of each device occupies a part of these resources. To determine and estimate the use of resources, consider that the CPU provides a resource of 100% every millisecond. The firmware documentation gives the percentage resource that each device uses in a sampling. In the same sampling the CPU can manage several devices and the overall resource allocation is the sum of the percentages for

each device. When choosing the times, make sure that the overall resource allocation for sampling is over 100%. The CPU automatically staggers device executions to avoid overshooting this maximum processing resource.
The diagram in figure 2 gives an example of this concept using the following devices:

- an OOPOS3 device with tc=6msec.,
- an OOPOS3 device with tc=30msec.,
- an EANPOS device con tc=2msec.
- a COUNTER3 device with tc=3msec
- a CAMMING3 device with tc=1msec.

Figure 2: CPU resource requirement for device management.



## 12.2.2 Consecutive commands and priorities

A command addressing a device is not processed immediately by the CPU, but at the next sampling time, without interrupting the QCL instructions. This is why the device may have to process several commands in the same sampling instant, and management of the device does not account for the sequence these commands are given but processes them according to an internal priority. This priority is specified in the documentation of each device.
To ensure the sequence of commands given in the QCL code, introduce WAIT instructions conditioned by a device status. In this way the WAIT instruction waits for the execution of the command before executing the next instruction.

## 12.2.3 Complementary commands

Some commands are complementary with others, i.e. they each produce the opposite effect of the other. If a device has to execute complementary commands in a given sampling time, only the last in the sequence has effect. To ensure execution of the two commands see “Consecutive commands and priorities” above.

## 12.3 Device groups

A group of devices and be defined in the INTDEVICE section by the *DEVGROUP* and *ENDDEVGROUP* keywords. Example:

```
DEVGROUP Axes
  X ANPOS2 4 3.CNT01 1 3.INP19 X.X 3.AN01
  Y ANPOS2 4 3.CNT02 2 3.INP18 X.X 3.AN02
ENDDEVGROUP
```

The devices in a group have two restrictions:

1. they must have the same sampling time
2. the sum of their execution times must not be more than the maximum execution time of a sampling.

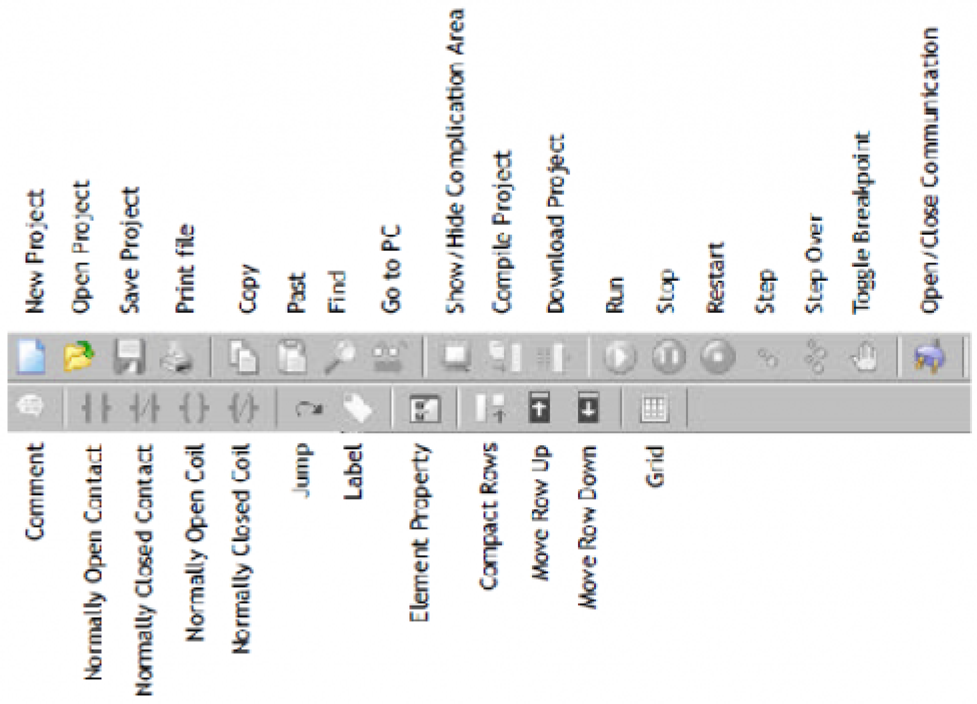
Definition of a group ensures execution of all its devices at the same sampling time. This function is dedicated to particular motion control applications.

# 13. The Qview environment

This chapter describes the menus and commands in QVIEW. For details on some Windows functions (e.g. Open, Save, etc.) consult the operating system instructions.

## 13.1 Tool Bar

The Tool Bar has the icons of the most common command functions.



## 13.2 Status Bar

The Status Bar has four separate sections.
1 left - shows the value of the selected variable. Click the right mouse button Click on a variable or device parameter to see its value in this section (the serial connection must be active). 2 - shows the cursor position (row and column) and the write mode (INS = insert text and OVR = overwrite text) (Figure 1).
3 - shows the serial port communication status between PC and CPU: port status (*No connection* or *Connected with:...*), communication protocol and transmission speed (Figure 2).
4 - shows the “Match OK” or “No Match” messages when the serial communication is active and the projects on PC and CPU are the same or different.

Figure 1: section 2 from left: cursor position.

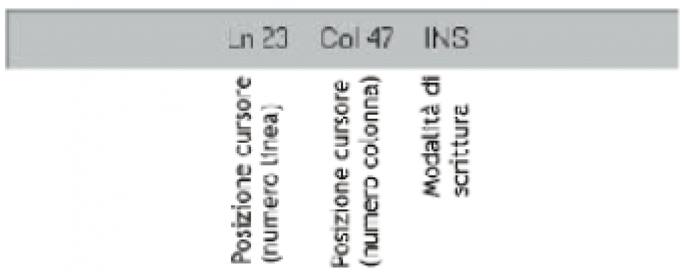
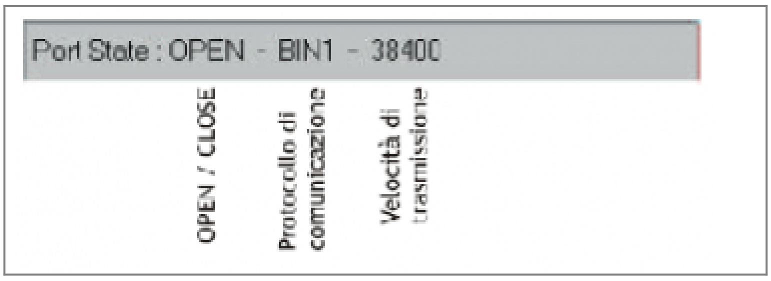

Ln 23
Col 47
INS
Posizione cursore (numero linea)
Posizione cursore (numero colonna)
Modalità di scrittura

Figure 2: section 3 from left: serial port communication parameters and status .


Port State : OPEN - BIN1 - 38400

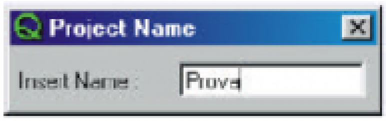
## 13.3 Menu > File

The commands for managing the project and related files.

### 13.3.1 New Project

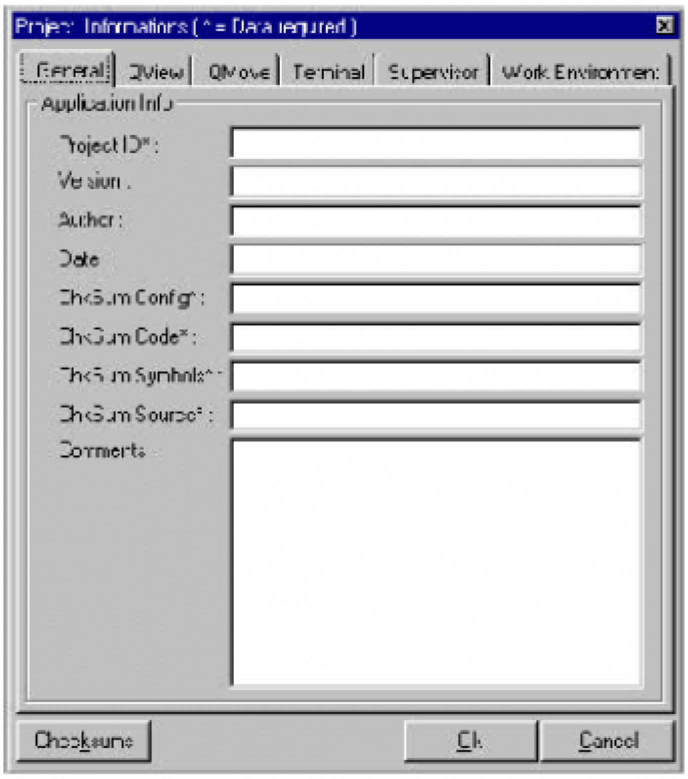
Create a new project. A name must be given to the project (Figure 1).
The Project Information window opens (Figure 2) to enter the project data. This form is for information about the project and is optional so it can be completed later.

Figure 1: entering the project name.



Insert Name : Prova

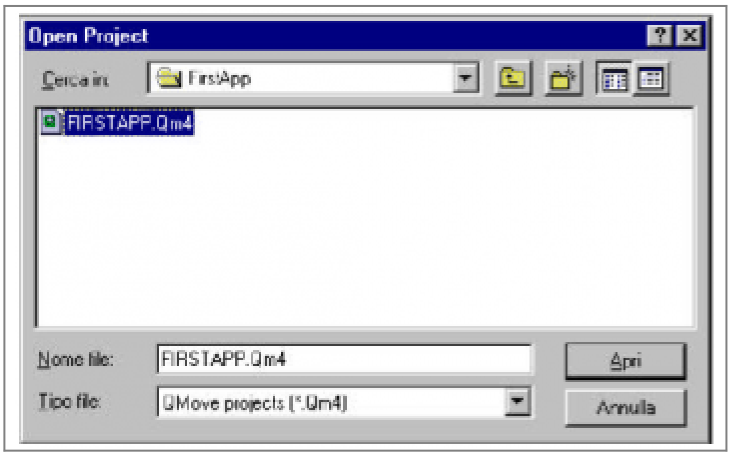
Figure 2: project information window.



### 13.3.2 Open Project

Open an existing project (Figure 3).

Figure 3: opening a project.



A project created with a previous version of Qview can be opened by selecting the different extensions in “File Type”.

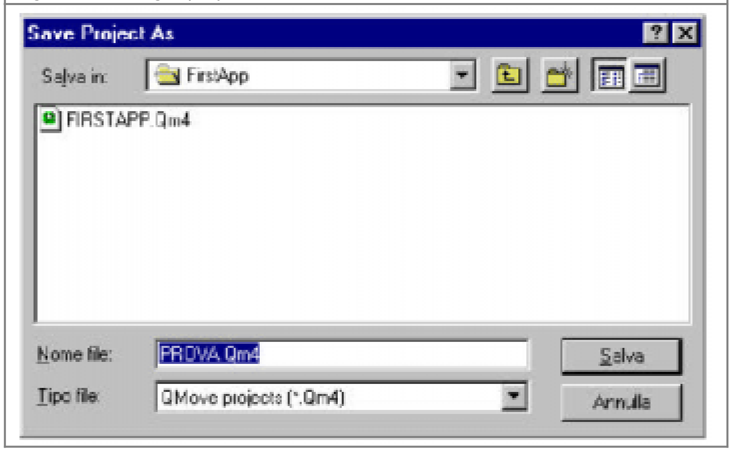
### 13.3.3 Save Project

Save any changes made to the project.

### 13.3.4 Save Project As

Save a copy of the project under a different name, including any changes (Figure 4).

Figure 4: saving a project under a different name.



==== - Close Project ===== - Close the project.

### 13.3.5 Add Unit

Add a new unit to the project.
New unit options:

- Configuration Unit: enter the configuration unit. This is not enabled if a configuration unit already exists in the project.
- QCL Unit: add a new task unit in QCL language
- Ladder Unit: add a new task unit in Ladder language
- Document Unit: add a new document unit (text document to group comments, spec's, notes …).

The new unit is added to the bottom of the list of existing units.

### 13.3.6 Insert Unit

Insert a new unit in the project.

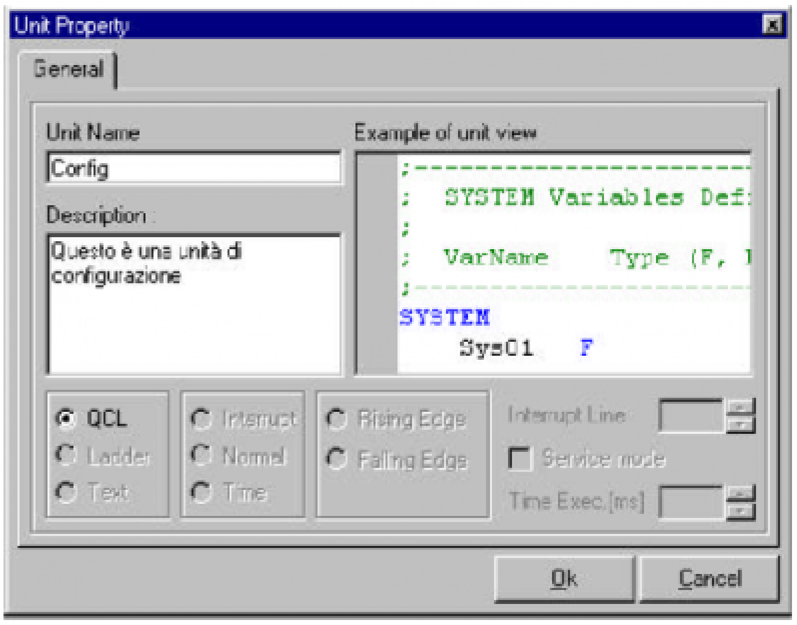
See *Add Unit* above for the new unit options.

The new unit is inserted above the selected unit in the list.

### 13.3.7 Create a configuration unit

When a new configuration unit is created with the “Add unit” or “Insert unit” commands, the “Unit property” window opens (Figure 5).

Figure 5: Configuration unit title.



Unit Name: Config

Description: Questo è una unità di configurazione

Example of unit view:

```
;------------------------
;  SYSTEM Variables Def:
;
;  VarName     Type  (F, I
;----------------------
SYSTEM
    Sys01    F
```

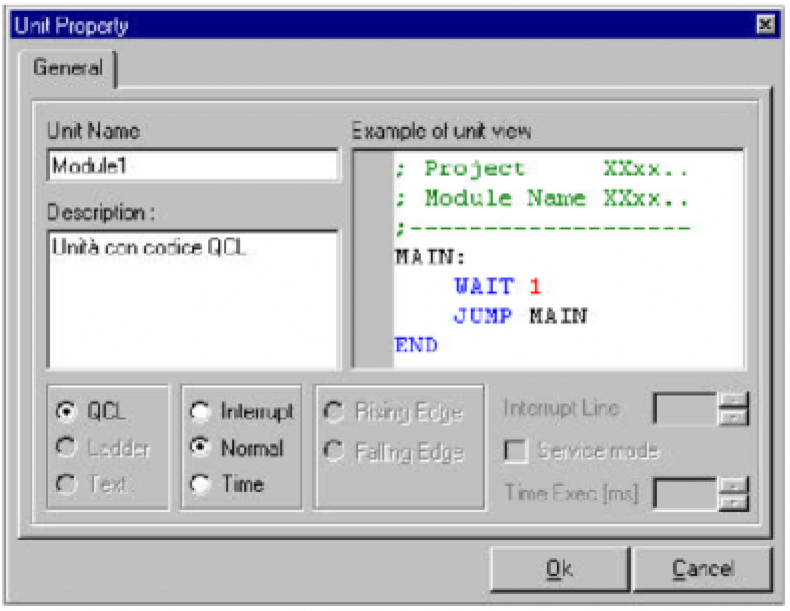
(● QCL) Ladder, Text

This window assigns a unit name and brief description of the unit. Confirm with ***Ok*** to add the configuration unit to the unit list and the editor window is opened to begin editing it.
Only one configuration unit can exist in a project.

### 13.3.8 Create a QCL or LADDER unit

Whenever a new QCL or LADDER unit is created by the “Add unit” or “Insert unit” commands, a “Unit property” window opens (Figures 6, 7).

Figure 6: QCL unit title.



Unit Name: Module1

Description: Unità con codice QCL

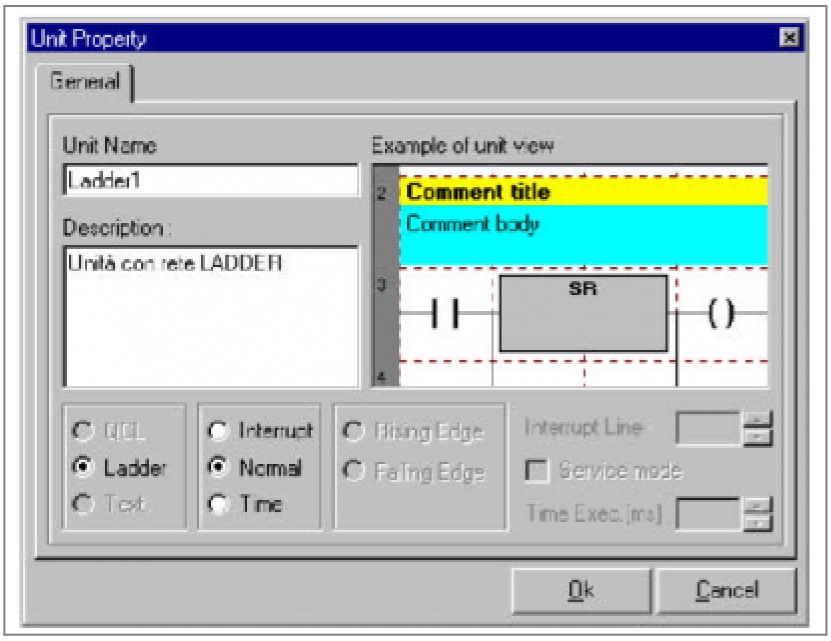
Example of unit view:

```
; Project     XXxx..
; Module Name XXxx..
;------------------
MAIN:
    WAIT 1
    JUMP MAIN
END
```

(● QCL) Ladder, Text

Interrupt (● Normal) Time

Figure 7: LADDER unit title.



Unit Name: Ladder1

Description: Unità con rete LADDER

( ) QCL [✓ Ladder] ( ) Text

( ) Interrupt [✓ Normal] ( ) Time

( ) Rising Edge ( ) Falling Edge

Give a name to the unit and a brief description.
Press ***Ok*** to confirm and the QCL or LADDER unit is added to the list of units and is opened in an editor window for modification.
The “Unit property”window specifies the unit properties (QCL or LADDER)

- **normal**
- **interrupt**: specify the pulse front (i.e. **Rising Edge**, **Falling Edge**) that starts the interrupt and the dedicated interrupt line

(**Interrupt Line**);

- **time**: specify the repeat time for the unit code (**Time Exec.**).

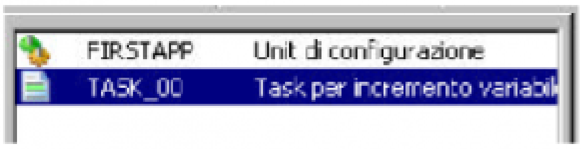
### 13.3.9 Create document unit

Document units are used text only units to write comments, notes and operating details.

### 13.3.10 Remove Unit

Remove the selected unit.

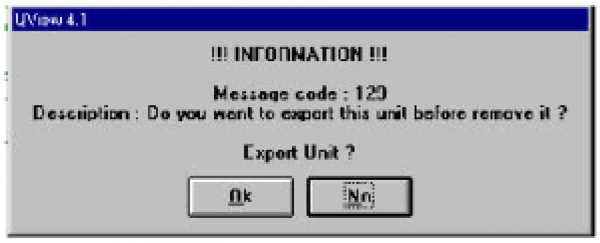
Figure 8: selecting a unit for removal.



FIRSTAPP Unit di configurazione
TASK_00 Task per incremento variabil

Before deleting the unit Qview asks if the unit has to be exporting before removal, otherwise the unit is lost.

Figure 9: export unit request.



QView 4.1

!!! INFORMATION !!!

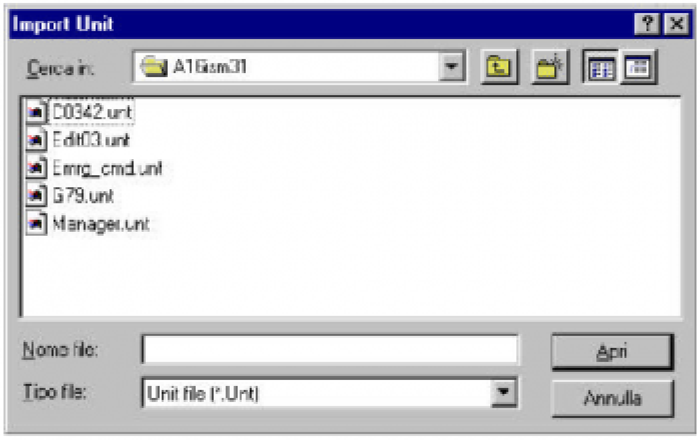
Message code : 120
Description : Do you want to export this unit before remove it ?

Export Unit ?

### 13.3.11 Import Unit

Import a unit previously exported from another project (Figure 10).

Figure 10: Importing a unit.

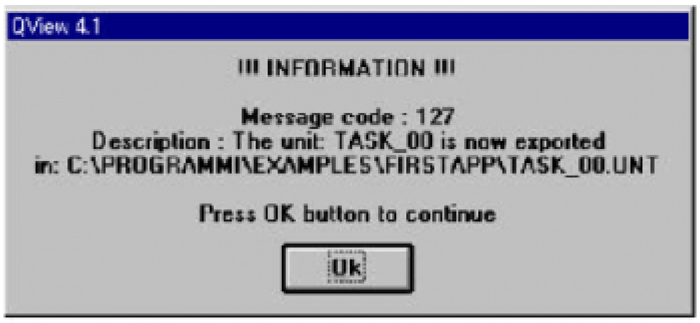


The two alternatives are “Add unit...” to import a unit at the end of the list or “Insert unit...” to import a unit above the selected unit.

## 13.3.12 Export Unit

Export a copy of the selected unit for transfer to another project. The original remains in the project and the unit is exported to the same directory as the project. The unit is exported with a *.unt file extension.

Figure 11: Confirm export.


QView 4.1

!!! INFORMATION !!!

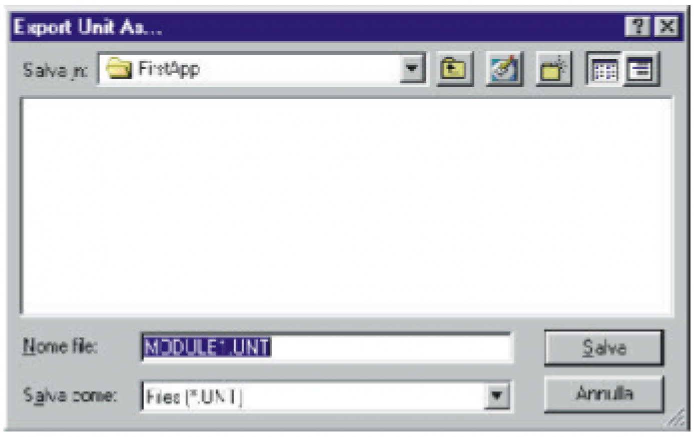
Message code : 127
Description : The unit: TASK_00 is now exported
in: C:\PROGRAMMI\EXAMPLES\FIRSTAPP\TASK_00.UNT

Press OK button to continue

## 13.3.13 Export Unit As

Export a copy of the selected unit for transfer to another project with another unit name. The original remains in the project.

Figure 12: exporting a unit with another name.



## 13.3.14 Unit Property

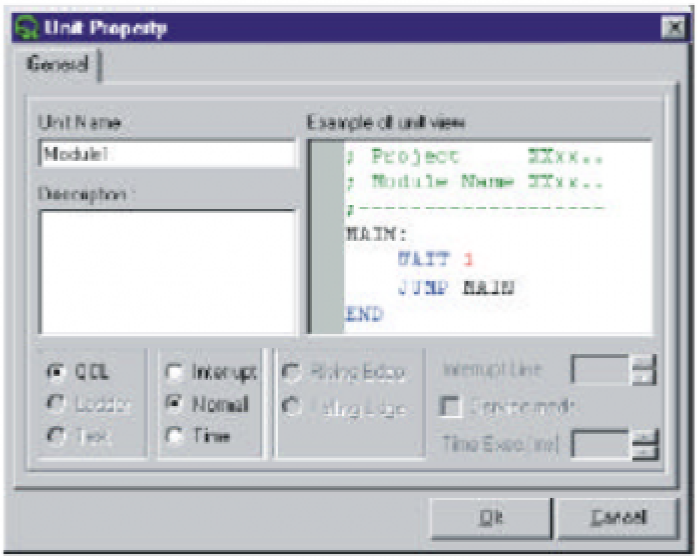
Modify the unit properties and especially the unit Runtime settings, i.e. the unit execution mode when it is downloaded onto the Qmove CPU. The units can be given the following settings:

**Normal** Maximum 65535 units in a project.
The unit is given a Runtime loop (see Multitasking chapter).

**Interrupt** Maximum three units in a project.
The unit is given a Runtime behaviour determined by external events (i.e. interrupt hardware lines). Unit execution can be determined to activate on the rising or drop front of the incoming signal and the associated hardware line number, with settings from 1 to 8.

**Time** Maximum seven units in a project.
The unit is given a Runtime behaviour at a pre-set time rate in the hardware.
The rate can be adjusted from 1 to 999 milliseconds. A new time unit is given a default time of 100ms.

Figure 13: changing unit properties.



Unit Name: Module1

Description:

Example of unit view

```
; Project     XXxx..
; Module Name XXxx..
;-------------------
MAIN:
    WAIT 1
    JUMP MAIN
END
```

[✓ QCL] Ladder Text

Interrupt [✓ Normal] Time

Rising Edge  Falling Edge

Interrupt Line

Service mode

Time Exec (ms)

The “Unit property” window (Figure 13) can be used to give a new Unit Name to an existing unit.

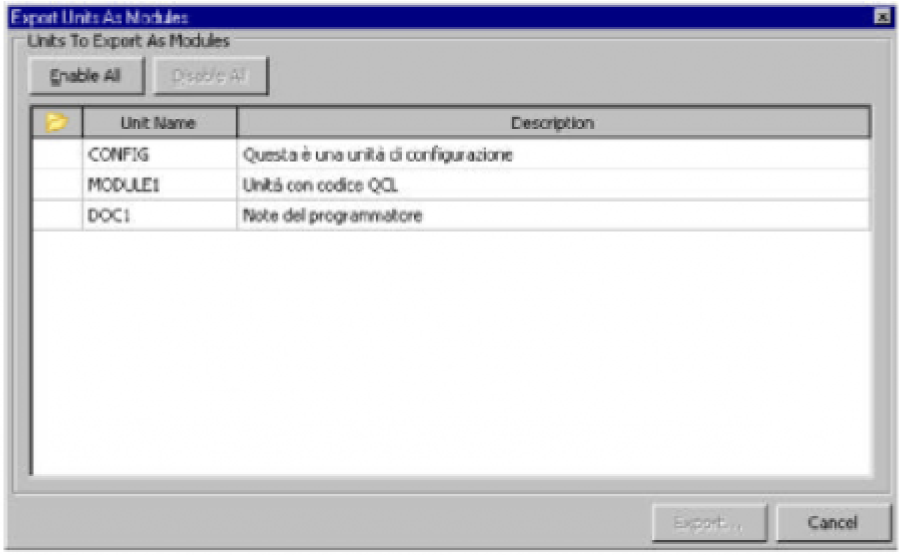
## 13.3.15 Import Module As Unit

The old versions of Qview used the term module instead of unit. Import a module created in an old version of Qview (*.mod). The unit can be added at the end of the list (Add Module as unit) or inserted above a selected unit (Insert Module as Unit).

## 13.3.16 Export Unit as Module

Export units in a compatible format for old versions of Qview (i.e. Qview 2.x & Qview 3.x Modules). The window in Figure 14 opens, where units can be selected for export (only units in QCL not LADDER).

Figure 14: exporting units as modules.



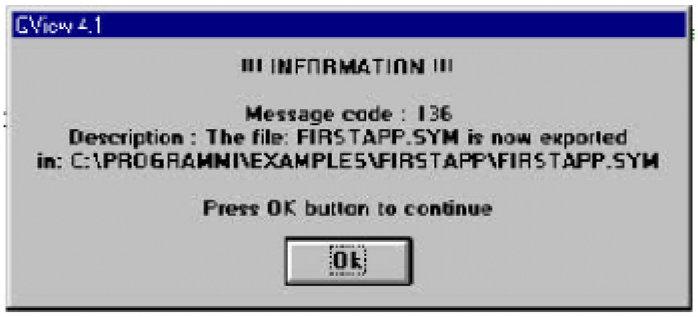
|  | Unit Name | Description |
|---|---|---|
|  | CONFIG | Questa è una unità di configurazione |
|  | MODULE1 | Unità con codice QCL |
|  | DOC1 | Note del programmatore |

Double click a unit to select.

### 13.3.17 Export Symbols File

Export symbols from a project to realign the symbols in the terminal.

Figure 15: exporting symbols.



QView 4.1

!!! INFORMATION !!!

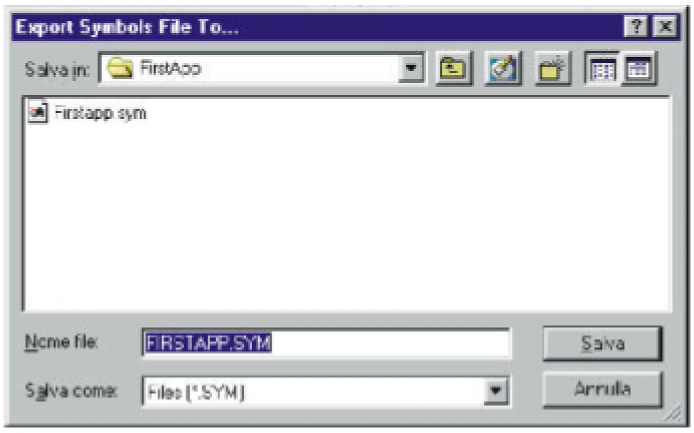
Message code : 136
Description : The file: FIRSTAPP.SYM is now exported
in: C:\PROGRAMMI\EXAMPLES\FIRSTAPP\FIRSTAPP.SYM

Press OK button to continue

### 13.3.18 Export Symbols File As ...

Export and rename a symbol file from the project to realign the symbols in the terminal.

Figure 16: exporting and renaming symbols files.



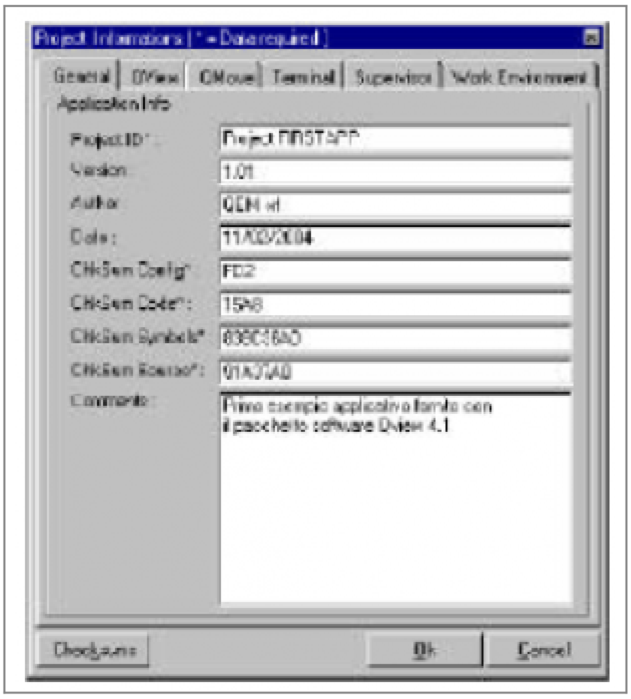
### 13.3.19 Export Binary File & Export Binary File As ...

Function only available if the project compilation is successful. Export the binary file (i.e. compilation result) to download the application on the CPU without using the serial communication (e.g. transferring it to the CPU by Multi Media Card). Binary files can be exported with the same project name (**Export Binary File**) or changing the name (**Export Binary File As...**).

### 13.3.20 Project Information ...

Open the Project Information window (Figure 20) to enter specific information about the project.

Figure 17: project information window.



Application Info

Project ID* : Project FIRSTAPP
Version : 1.01
Author : QEM srl
Date : 11/02/2004
ChkSum Config* : FD2
ChkSum Code* : 15A8
ChkSum Symbols* : 838C16AD
ChkSum Source* : 91A37A0
Comments : Primo esempio applicativo fornito con il pacchetto software Qview 4.1

This window is made up of folders that divide the project information by topic. Mandatory information is marked by an asterisk. If any mandatory information is not entered, messages will periodically appear to signal the missing information (to disable the messages see ***Menu > Options > Program Setup***). The window has a “Checksums” button that automatically inserts the project checksum codes.

### 13.3.21 Print

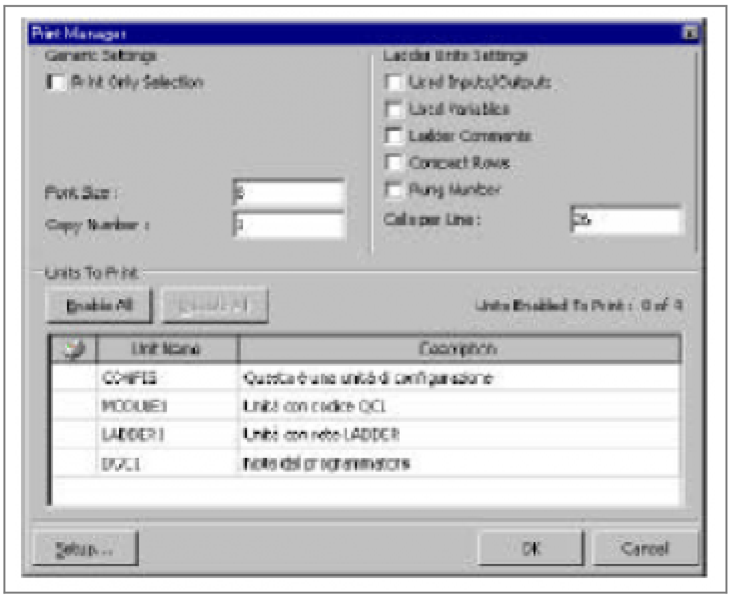
**Command only available after selecting an editor window.**

Open the “Print Manager” window to print part or all the project with various print options. Figure 18 shows the window where the bottom box gives the list of units in the project, which can be selected for printing. The top left-hand box has the usual font size and n. copies settings.
The top right-hand box configures the LADDER task printout, specifying:

- print the list of inputs and outputs in the LADDER task
- print the variables in the LADDER task
- print the comments in the LADDER task
- print compacting the LADDER rows deleting all empty spaces
- print the number of rungs;
- print a number of cells per row.

In the top left of the window select to print only the selected part of the task unit.

Figure 18: selecting the print mode.


Print Manager

Generic Settings

Print Only Selection

Font Size : 8

Copy Number : 1

Ladder Units Settings

Local Inputs/Outputs

Local Variables

Ladder Comments

Compact Rows

Rung Number

Cells per Line : 26

Units To Print

Enable All | Disable All

Units Enabled To Print : 0 of 4

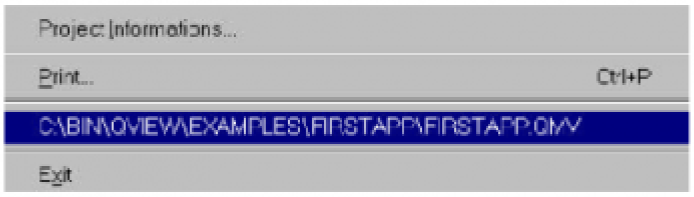
| | Unit Name | Description |
|---|---|---|
| | CONFIG | Questa è una unità di configurazione |
| | MODULE1 | Unità con codice QCL |
| | LADDER1 | Unità con rete LADDER |
| | DOC1 | Note del programmatore |

Setup... | OK | Cancel

### 13.3.22 Recent projects

The last 4 files opened can be directly recalled (Figure 19).

Figure 19: opening recent projects.



### 13.3.23 Exit

Close QVIEW and confirm to save any changes made to the project.

## 13.4 Menu > Edit

**Commands only available when a unit is open in the editor.**

### 13.4.1 New Element

Insert a new element in the Ladder grid (a LADDER unit must be open).

### 13.4.2 New Rung

Insert a new rung in the Ladder grid (a LADDER unit must be open).

### 13.4.3 Delete Rung

Delete the selected rung in the Ladder grid (a LADDER unit must be open).

### 13.4.4 Toggle Link

Connect two Ladder elements vertically. The link is always made downwards and to the left of the selected cell (a LADDER unit must be open).

### 13.4.5 Substitute Obsolete Element & Substitute All Obsolete Elements...

Substitute obsolete elements. For more details see “LADDER editor - Obsolete LADDER elements” (a LADDER unit must be open).

### 13.4.6 Element Properties...

Change the variables in a LADDER element (a LADDER unit must be open).

### 13.4.7 Undo

Cancel the last change.

### 13.4.8 Redo

Cancel the Undo command and restore the last change.

### 13.4.9 Cut - Copy - Paste - Delete

Cut, copy, paste and delete texts or elements.

### 13.4.10 Select All

Select all the text or elements in an editor window.

### 13.4.11 Compact Rows

Compact the Ladder code, eliminating any empty rows (a LADDER unit must be open).

### 13.4.12 Move Rows Up

Move a Ladder code line up (a LADDER unit must be open).

### 13.4.13 Move Rows Down

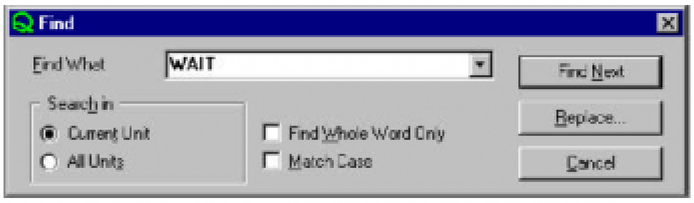
Move a Ladder code line down (a LADDER unit must be open).

### 13.4.14 Find

Find for a word by selected search criteria (Figure 20).
Search criteria

- Search in Current Unit: search only selected unit.
- Search in All Units: search all project units.
- Find Whole Word Only: find only whole words
- Match Case: search words as entered, with upper and lower case.

Figure 20: search criteria window.


Find
Find What: WAIT
Search in: (● Current Unit) ○ All Units
☐ Find Whole Word Only
☐ Match Case

### 13.4.15 Find Next

When a term is found by Find, the search for the same term can continue in the rest of the project with the same search criteria.

### 13.4.16 Replace

Find a term (Find What) with the set search criteria and replace it with a new term (Replace With)(Figure 21).

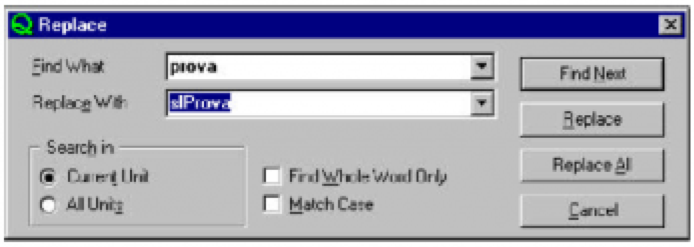
**Search criteria**

- Search in Current Unit: search only the selected unit
- Search in All Units: search all project units
- Find Whole Word Only: find only whole words
- Match Case: find words as entered, with upper and lower case.

**Replace criteria**

- Find Next: no replacement, just a word search ( Find What) in the rest of the project;
- Replace: the search finds the word ( Find What) and replaces it with a new word (Replace With);
- Replace All: all of the words are found ( Find What ) and automatically replaced in all the project.

Figure 21: search and replace criteria window.



Find What: prova
Replace With: slProva
Search in: [✓ Current Unit] All Units
☐ Find Whole Word Only
☐ Match Case

## 13.4.17 Go to ...

The cursor goes to the set line number (Figure 22).

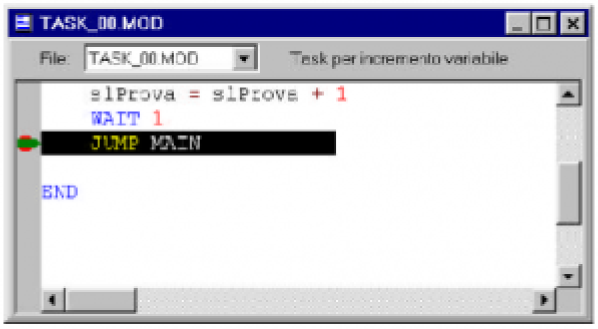
Figure 22: going to a set line number.

7

## 13.4.18 Go to PC

When execution of the project is interrupted (e.g. by a breakpoint), this command shows the code line causing the interruption. In Figure 23 the Go to PC command shows an interruption by a breakpoint.

Figure 23: program execution interrupted by a breakpoint.



File: TASK_00.MOD    Task per incremento variabile

```
    slProva = slProva + 1
    WAIT 1
    JUMP MAIN

END
```

## 13.4.19 Next Unit / Previous Unit

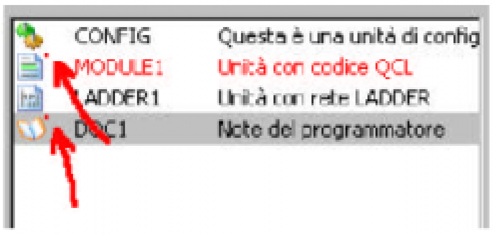
Go to the next or previous unit in the project.

## 13.4.20 Next Selected Unit / Previous Selected Unit

Go to the next or previous selected units, skipping the others. Select the units one by one and press the space bar. A red spot beside

the unit marks the selection (Figure 24).
The Next and Previous Selected Unit commands can scroll forward and back only between the selected units.
To remove the marking, select the unit, press the space bar again and the red spot will disappear.

Figure 24: selecting units for an analysis.



| | |
|---|---|
| CONFIG | Questa è una unità di config |
| MODULE1 | Unità con codice QCL |
| LADDER1 | Unità con rete LADDER |
| DOC1 | Note del programmatore |

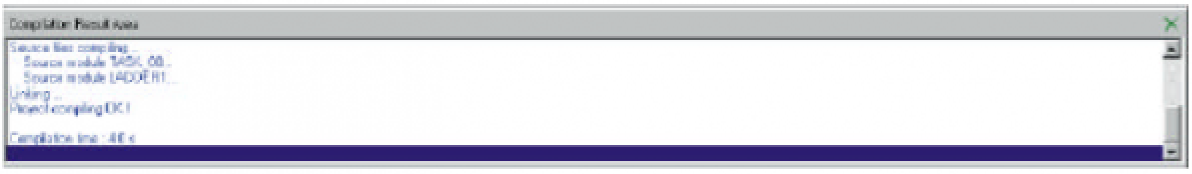
## 13.5 Menu > Project

The project and CPU data management commands.

### 13.5.1 Compile

Conversion of the project into a format for interpretation by the CPU. The project can only be downloaded onto the CPU if it has been compiled without errors (Figure 25).

Figure 25: project compilation window.



Compilation Result Area

```
Source files compiling...
   Source module TASK_00...
   Source module LADDER1...
Linking...
Project compiling OK !

Compilation time : 4.6 s
```

If the project compilation gives errors, double click the message in the compilation window for the editor to show the code line containing the error.

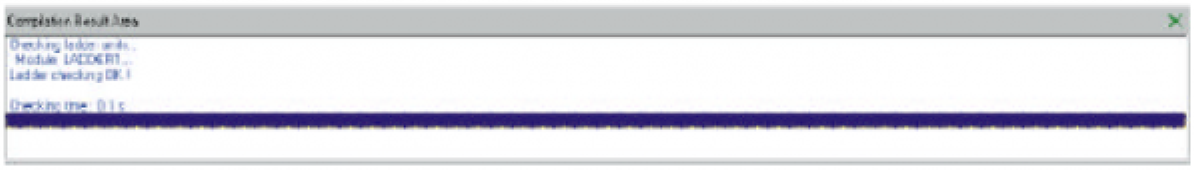
### 13.5.2 Force Compile

Compilation of the whole project is forced regardless of its compiled condition.

### 13.5.3 Ladder Network checking

A check on the Ladder net. A window opens to give the compilation result (Figure 26).

Figure 26: Ladder net compilation window.



Compilation Result Area

```
Checking ladder units...
  Module LADDER1...
Ladder checking OK !

Checking time: 0.1 s
```

### 13.5.4 View compilation result

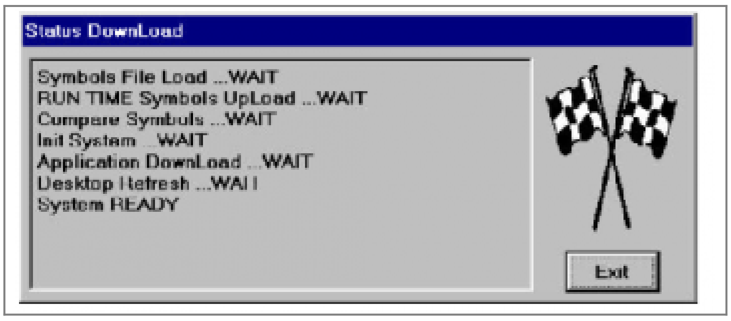
View or hide the result window for the last compilation (Figure 24.

### 13.5.5 Download

**Command only available with the PC - QMOVE serial communication activated.**

The compiled project can be downloaded onto the CPU. The stages in the download are shown in a window (Figure 27).

Figure 27: download outcome.



Status DownLoad

```
Symbols File Load ...WAIT
RUN TIME Symbols UpLoad ...WAIT
Compare Symbols ...WAIT
Init System ...WAIT
Application DownLoad ...WAIT
Desktop Refresh ...WAIT
System READY
```

## 13.5.6 Backup data

**Command only available after an application download.**

The application downloaded onto CPU contains code and data information.

**Unchangeable Application** Application information that is not changed, e.g. QCL instructions, symbols, application title, etc

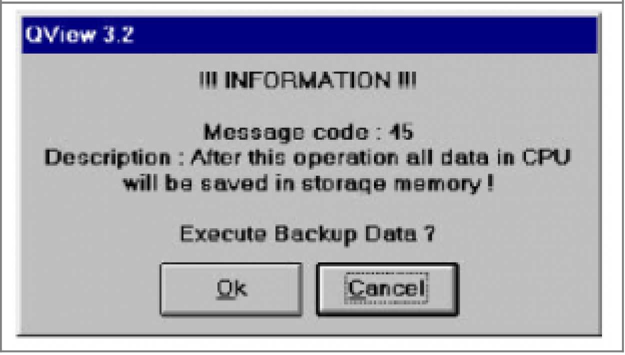
**Changeable Data** Data indicating the application operating conditions, e.g. device data areas, variable values, array systems, datagroup contents and data that modifies during operation (all the application data).

Backup creates a safety copy of all data, recording it on the internal non-volatile mass storage. This facility is useful because it offers the possibility to restore all existing parameter settings at a given point. The values are copied on non-volatile mass to give maximum security (Figure 28).
Execution conditions of the backup command:

- CPU in READY status.
- The application does not use more RAM space than the backup limit.
- The total space taken up by retentive and non-retentive data must not be more than the non-volatile memory space available.

If the backup is executed from the terminal and takes longer than the terminal timeout, a timeout error will trip in the serial communication between CPU and terminal.

Figure 28: confirm request for backup.



QView 3.2

!!! INFORMATION !!!

Message code : 45
Description : After this operation all data in CPU will be saved in storage memory !

Execute Backup Data ?

## 13.5.7 Restore data

**Command only available after downloading the application.**

The application recalled from the CPU contains application and data information.

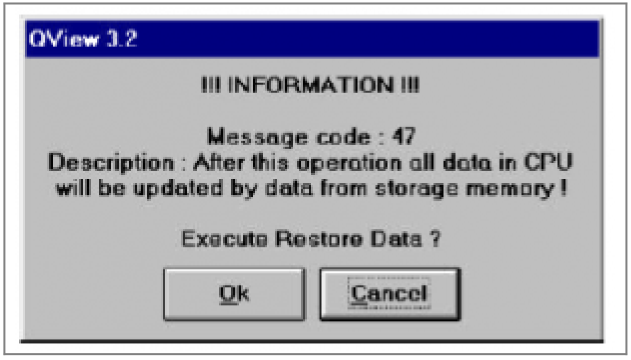
**Unchangeable Application** The application that is not modified, e.g. QCL instructions, symbols, application title, etc.

**Changeable data** Data indicating the operating conditions, e.g. device data areas, variable values, array systems, datagroup contents and any data that changes value during operation ( all application data).

The restore command recovers all the data with the backup data (Figure 29).
The backup contents are deleted during the download procedure, since there is no reason to copy the values of an application's variable data to another application. Execution conditions of the restore command:

- A previous backup must have been made
- CPU is in READY or ERROR status.

Figure 29: confirm restore procedure.



QView 3.2

!!! INFORMATION !!!

Message code : 47
Description : After this operation all data in CPU will be updated by data from storage memory !

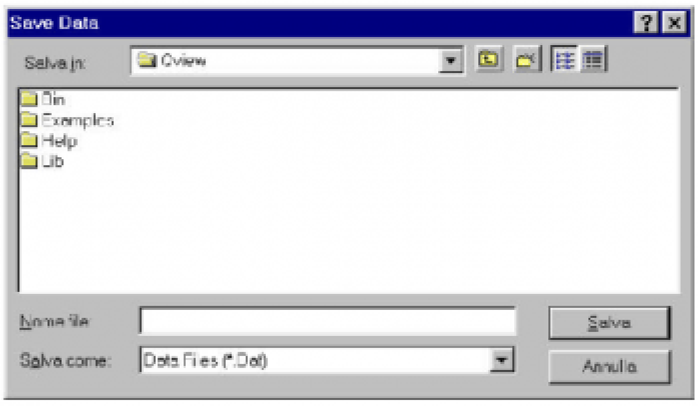
Execute Restore Data ?

### 13.5.8 Save Data...

**Command only available after downloading the application.**

The CPU data (i.e. retentive variable values) are saved in a .DAT file, defining also the destination directory (Figure 30).

Figure 30: saving CPU data on file.



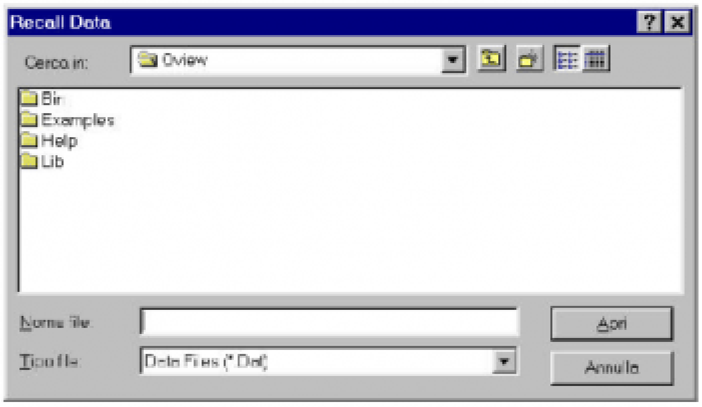
### 13.5.9 Recall Data...

**Command only available after downloading the application.**

Replace the retentive variable values on the CPU with the values of the same variables stored in a .DAT file(Figure 31).

The replacement is only made to variables that are common to the application and the .DAT file, so, for example, if variables are added on the CPU after the last data backup, they are not replaced.

Figure 31: replacing CPU data by .DAT file

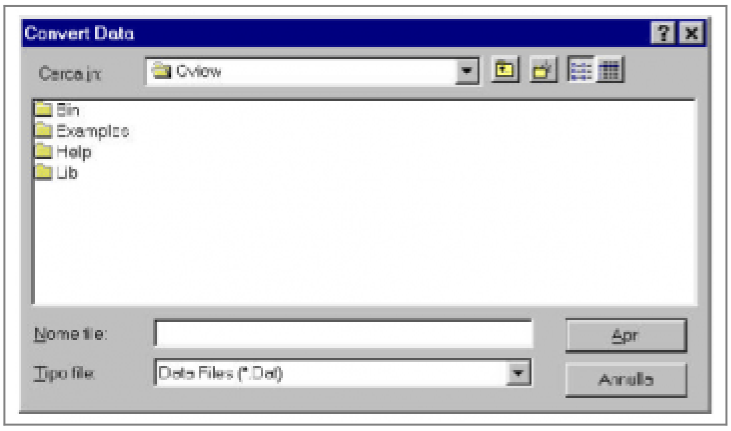


### 13.5.10 Convert Data...

**Command always available.**

Convert the .DAT file into a .TXT text file containing the variable names and values. The text file is generated in the containing the .DAT file and is given the same name as the .DAT file (Figure 32).

Figure 32: data file conversion.



Convert Data

Cerca in: Qview

Bin
Examples
Help
Lib

Nome file:

Tipo file: Data Files (*.Dat)

### 13.5.11 Checksum View

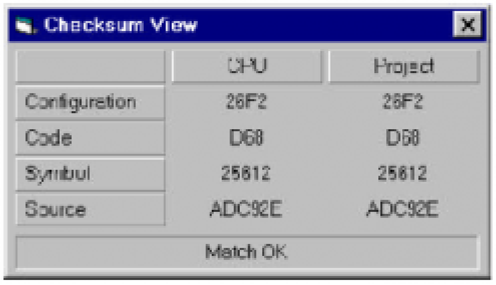
**Command available after opening a project.**

Compare project checksums with the application downloaded on the CPU (Figure 33), if they are the same the CPU and PC have the same project. Any differences are signalled with the checksum values in red. The following data types are compared:

- Configuration: memory use configuration.
- Code: QCL code generated by the compilation.
- Symbol: symbols of the variables, which depends on the list of variables and their types.
- Source: unit contents

CAUTION! Compilation of the same project by two different Qview versions or builds guarantees the same functions, but does not guarantee that the checksum codes are kept the same.

Figure 33: checksum summary and comparison.



Checksum View

| | CPU | Project |
|---|---|---|
| Configuration | 26F2 | 26F2 |
| Code | D68 | D68 |
| Symbol | 25812 | 25812 |
| Source | ADC92E | ADC92E |

Match OK

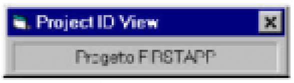
The CPU column gives the values in the CPU and its project signature.
The Project column is updated when the project is opened (if compiled) and after each compilation.

### 13.5.12 View Project ID

**Command available only with the PC - QMOVE serial communication open.**

View the project name. The name was defined in the Project Information window. When downloading, the Project ID is downloaded on the CPU with the application and is then visible by the Project ID View command (Figure 34).

Figure 34: Viewing the project name string.



Project ID View

Progetto FIRSTAPP

## 13.6 Menu > Debug

Commands for executing the application downloaded onto the CPU.

### 13.6.1 Run

**Command available after a download.**

Execution of the application downloaded onto the CPU..

### 13.6.2 Stop

**Command available after a run.**

Stop execution of the CPU application (the devices still continue to operate).

### 13.6.3 Restart

**Command available after a start or stop.**

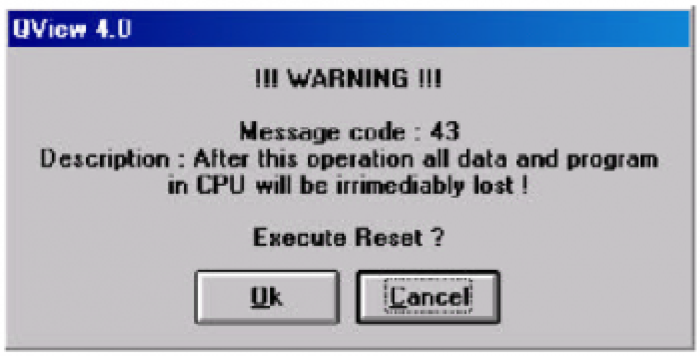
Execution of the application is stopped and at the next RUN the project starts from the beginning.

### 13.6.4 Reset

**Command available only with the PC - QMOVE serial communication open.**

Delete the application in the CPU. The data is deleted definitively so a safety confirmation is requested during the reset procedure (see Figure 35).

Figure 35: confirm reset project procedure.



QView 4.0

!!! WARNING !!!

Message code : 43
Description : After this operation all data and program in CPU will be irrimediably lost !

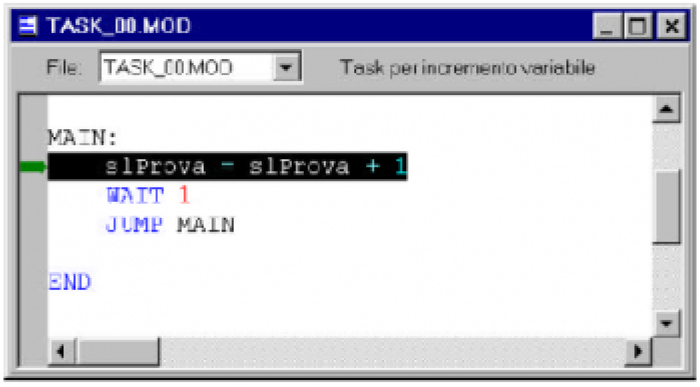
Execute Reset ?

### 13.6.5 Step

**Command available only after a download and selecting the editor window.**

Execute the project step by step and at each step the program execution flow advances by a step. When a WAIT instruction is found, the editor window editor shows the next unit so that the program can execute all the units at one step at a time.
A green arrow to the left of the editor window indicates the code line that has not been executed (Figure 36). LADDER units execute a rung in each step.

Figure 36: Program execution in Step mode.



```
MAIN:
    slProva = slProva + 1
    WAIT 1
    JUMP MAIN

END
```

### 13.6.6 Step Over

**Command only available after a download and selecting the text editor window.**

Execute the project one instruction at a time and at each step the project execution advances by an instruction. When a WAIT instruction is found the code execution continues normally through the other units until it returns to the WAIT instruction returning to step by step execution.
A green arrow to the left of the editor window indicates the code line that will be executed (Figure 37). LADDER units execute a rung in each step.

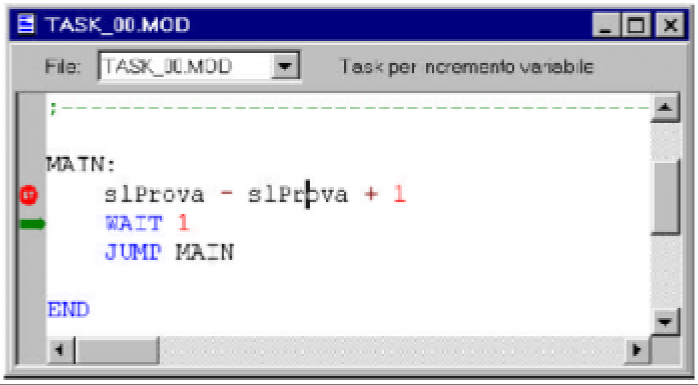
### 13.6.7 Toggle Breakpoint

**Command only available after a download and selecting the editor window.**
**Up to 5 breakpoints can be inserted, which are reset when the CPU is shut off and when the project is downloaded.**

Insert a breakpoint or in other words mark a code line where the program execution stops. Several breakpoints can be defined by

selecting a a code line for each breakpoint, the *Toggle breakpoint* command cancels the breakpoint. The breakpoints are indicated on the left of the editor window by a red mark with *ST* (Figure 37). Breakpoints can be inserted at rungs in LADDER units.

Figure 37: inserting a breakpoint.



```
;-----------------------------------------

MAIN:
    slProva = slProva + 1
    WAIT 1
    JUMP MAIN

END
```

### 13.6.8 Clear All

**Command only available after a download.**

Cancel all breakpoints in all the task units.

### 13.6.9 Watchpoint

**Command only available after a download.**
**Up to 5 watchpoints can be inserted, which are reset when the CPU is shut off and when the project is downloaded.**

Interrupt the project execution when a specific condition is satisfied, like a conditional breakpoint. The watchpoint command opens the window in Figure 38 to show the set stop conditions.

- Add: add a stop condition (Figure 39).
- Delete: delete the selected stop condition.
- Reset: delete all set stop conditions.

Figure 38: watchpoint list.

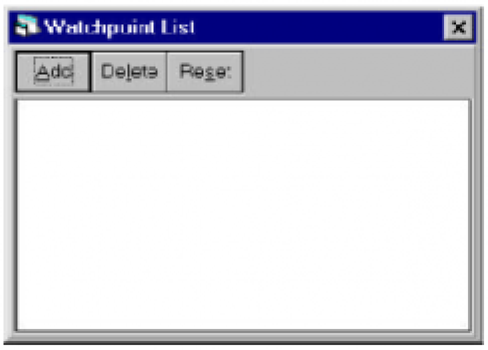
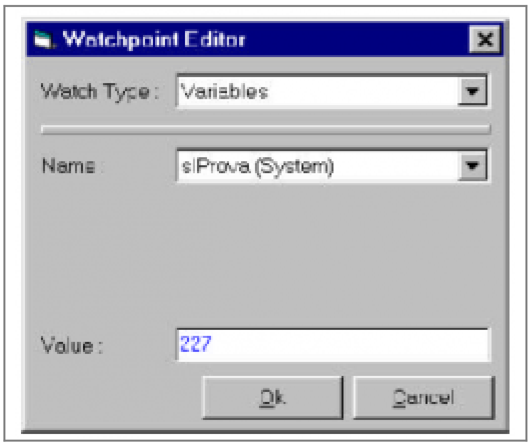


Figure 39: watchpoint parameter settings.


Watchpoint Editor

Watch Type : Variables

Name : slProva (System)

Value : 227

Click on ADD (Figure 38) to open the window in Figure 39 and set the program stop condition.

- Watch Type: select by variable type (i.e. Variables, I/O, Arrays, Data Groups, Devices).
- Name: select a variable belonging to the group defined in the Watch Type drop list.
- Value: enter a variable value that stops the project.

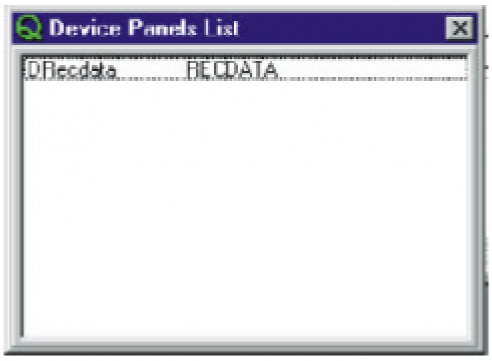
## 13.7 Menu > Monitor

The diagnostic commands for the QMOVE system and the project execution.

### 13.7.1 Devices Panels

In addition to the command parameter status list, Qview provides a user-friendly panel with the device name and type (Figure 51).

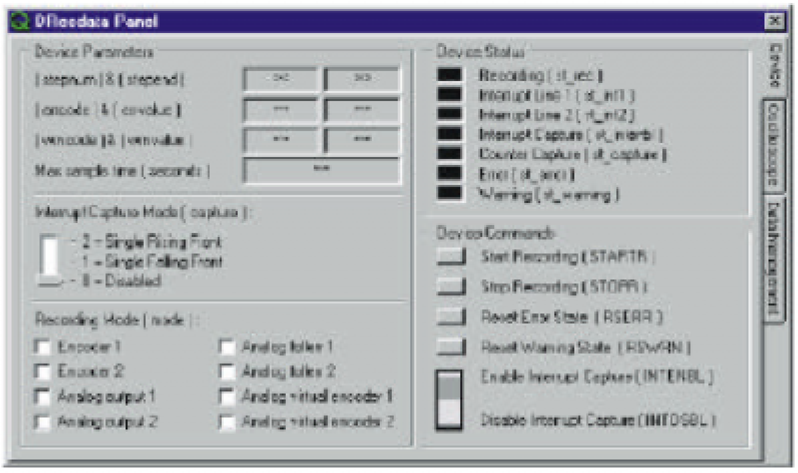
Figure 51: the device panels .


Device Panels List

DRecdata RECDATA

Select a device (by ENTER or double click) to open the window in Figure 52 that shows the device data. For more details on the devices panels see the help guide.

Figure 52: a typical device panel.



### 13.7.2 CPU MONITOR

The *CPU MONITOR* window gives information about the CPU status and the project execution times. The CPU Monitor Panel has three sections:

- GENERAL: general info on the CPU status and execution times of the whole project.
- TASK INFO: information on each unit.
- CPU’s oscilloscope: tool to trace variable value changes in time.

### 13.7.3 General

CPU status info

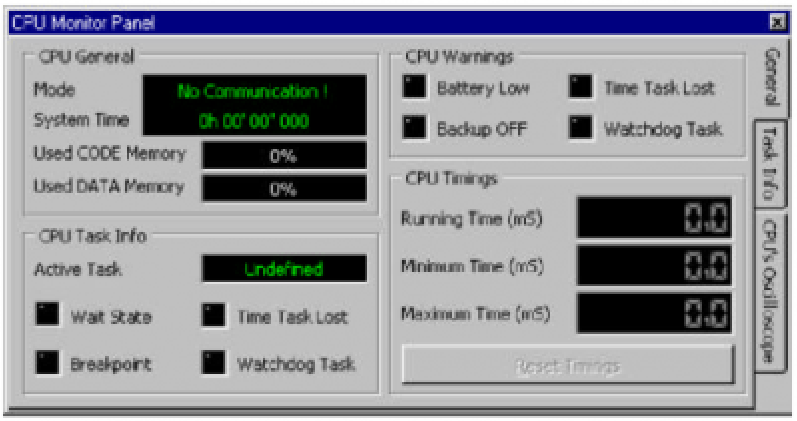
- Mode: the CPU mode(RUN, STOP, … Err). If Err it also shows the type of error.
- System Time: the application real run time from the CPU startup or restart.
- Used CODE Memory: the percentage memory space occupied by the code.
- Used DATA Memory: the percentage memory space occupied by the data.
- Battery low: low backup battery signal.
- Watchdog Task: signal when a task unit execution requires over 200ms. The signal persists until the CPU is shut-off. The project

continues to run normally, even though the signal indicates an error in the QCL programming.

- Backup OFF: signals the backup cannot be run (e.g. a high percentage of Data Memory space is occupied).
- Time Task Lost: signals when executing the program execution, a time task has not been executed one or more times.
- Running time in ms: task execution time.
- Active Task: task in use.
- Time Task Lost: signals that a time task unit has not been executed during the program execution.
- Wait State: signals a wait instruction is found in the task in use.
- Breakpoint: signals the task flow has found a breakpoint and has been interrupted.
- Watchdog Active: signals the active task has caused a Watchdog signal.

Moreover, MINIMUM TIME and MAXIMUM TIME show the times to execute the project.
There is also a *Reset Timing* button that zero-sets the two times and the *Running time*, enabling the processing time counter.

Figure 56: CPU general panel.


Mode: No Communication !
System Time: 0h 00' 00" 000
Used CODE Memory: 0%
Used DATA Memory: 0%
Active Task: Undefined
Running Time (mS): 0.0
Minimum Time (mS): 0.0
Maximum Time (mS): 0.0

#### 13.7.3.1 Task info

This menu can control all the units processed by the project and various types of information can be obtained. All the units can give information of the minimum, maximum and actual execution time.

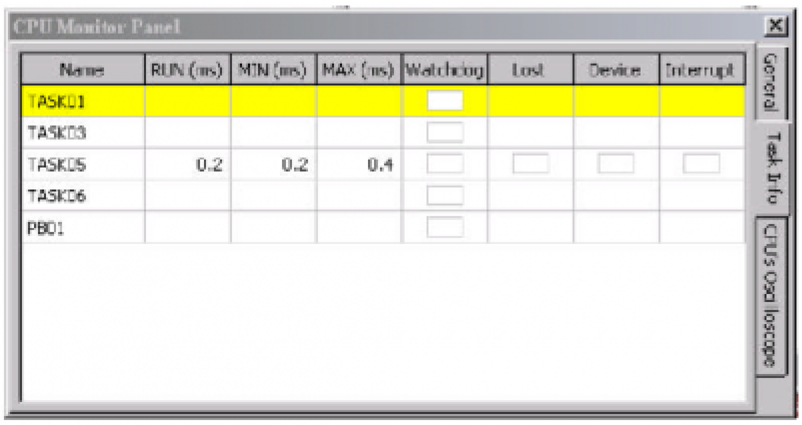
Task units also give information about the *WATCHDOG*. When a *WATCHDOG* is signalled for a unit, the CPU has taken over 200ms to execute its code without executing any other task (this is a fault that must be solved).

All other task units can have additional information:

- **TASK LOST:** Indication that the time task unit has lost an event.

- **DEVICE ACCESS:** Indication that the task unit has executed an access to a device that is not updated.
- **INTERRUPT:** Indication that a time task unit has been interrupted by an interrupt task unit (Figure 57).

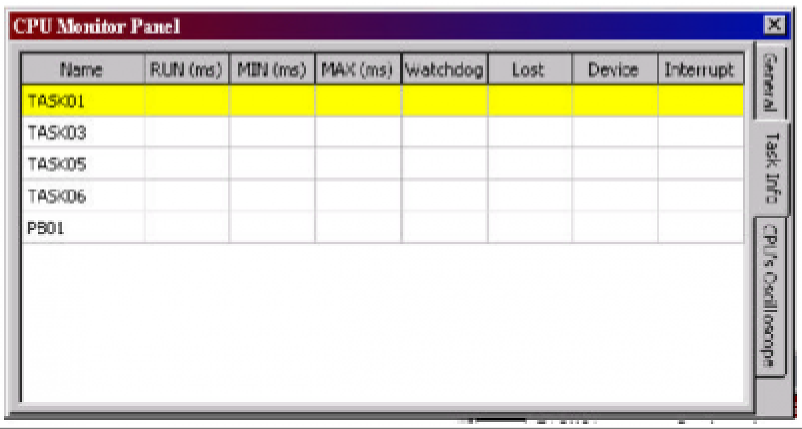
Figure 57: CPU general panel.


CPU Monitor Panel

| Name | RUN (ms) | MIN (ms) | MAX (ms) | Watchdog | Lost | Device | Interrupt |
|---|---|---|---|---|---|---|---|
| TASK01 | | | | □ | | | |
| TASK03 | | | | □ | | | |
| TASK05 | 0.2 | 0.2 | 0.4 | □ | □ | □ | □ |
| TASK06 | | | | □ | | | |
| PB01 | | | | □ | | | |

All the information in Figure 57 can only be shown if the CPU and PC check-sums match, otherwise the Task Info only shows the program units without any other information (Figure 58).

Figure 58: CPU Task info.


CPU Monitor Panel

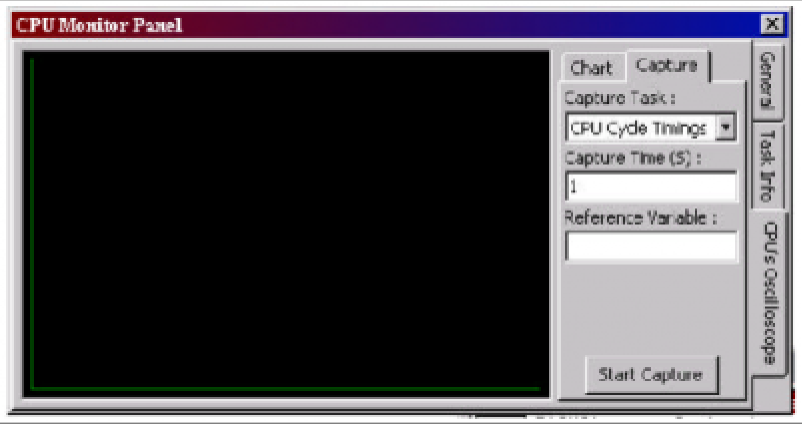
| Name | RUN (ms) | MIN (ms) | MAX (ms) | Watchdog | Lost | Device | Interrupt |
|---|---|---|---|---|---|---|---|
| TASK01 | | | | | | | |
| TASK03 | | | | | | | |
| TASK05 | | | | | | | |
| TASK06 | | | | | | | |
| PB01 | | | | | | | |

Errors are marked by changing the boxes to red, instead of the normal white.

### 13.7.3.2 CPU Oscilloscope

The oscilloscope shows the CPU status changes over time (Figure 59).

Figure 59: CPU Oscilloscope.


CPU Monitor Panel

Capture Task : CPU Cycle Timings
Capture Time (S) : 1
Reference Variable :

The acquisition speed is dependent on two factors :
- The serial speed. - How much data is controlled.

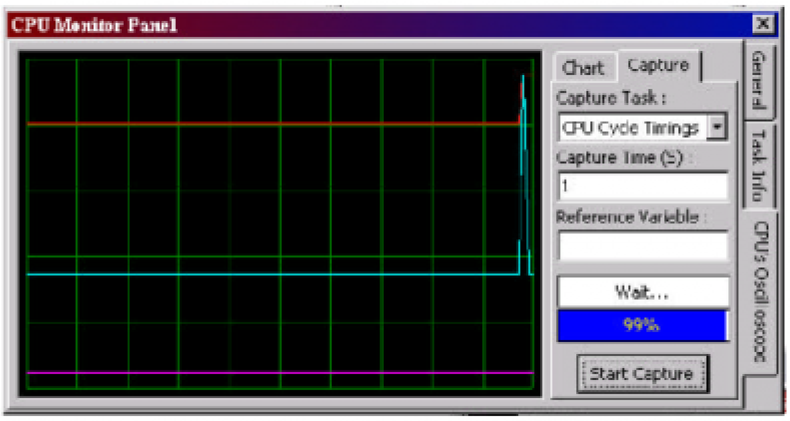
For an acquisition, insert the *Capture Time (S)* and the scan starts, continuing for the set time that cannot be over 9999 sec. When running an acquisition all the rest of Qview is blocked to give all resources to this function.

The acquisition can be associated to any variable, except a datagroup, and the acquisition time can also be enabled with the machine running without influencing its operation since the control does not burden the process.
Acquisition can be stopped at any time by pressing the Backspace key.

Save settings:

CAPTURE TASK: the measurement parameter settings:
- All task units, if set as CPU CYCLE TIMINGS to the cycle time of the whole program (Min. Max. Run). - On a single task unit, if a time or interrupt task, to see the cycle time of the selected task (Min. Max. Run). Naturally, the scan time for a normal task is for the whole program, however this particular case is also highlighted by a message (Figure 60).

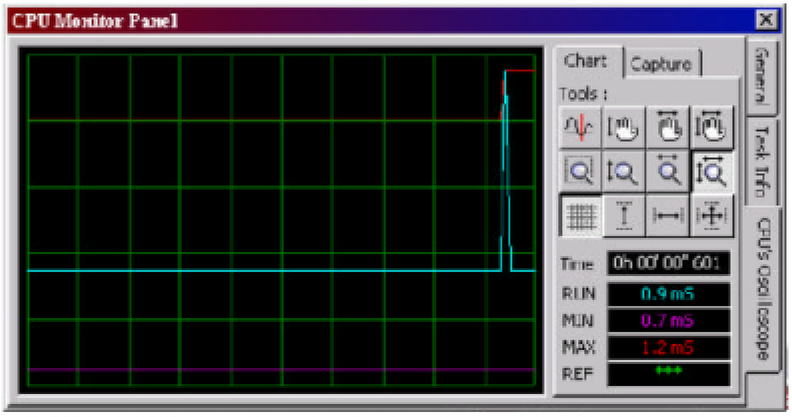
Figure 60: CPU Oscilloscope Capture Task.



Capture Task : CPU Cycle Timings
Capture Time (s) : 1
Reference Variable :
Wait...
99%

CHART: This window sets the graphic options to show characteristics and references related to the measurements (Figure 61).

Figure 61: CPU Oscilloscope Chart.



Time 0h 00' 00" 60:
RUN 0.9 mS
MIN 0.7 mS
MAX 1.2 mS
REF ---

To know the meaning of a button, pass the cursor over the it and a description is given.
In addition to task unit times, the time of a reference variable can be monitored by specifying it in the “Reference variable” of the Capture section.
If the variable does not exist a signal message is shown, the recoding is made but without the variable.

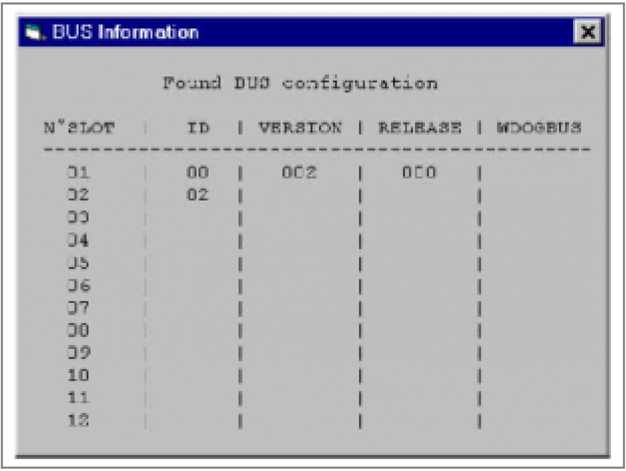
## 13.7.4 BUS

**Command available after opening an existing project and the PC - QMOVE serial communication port.**

The BUS Information window (Figure 63) shows the hardware construction and firmware versions/releases. This information is acquired by the CPU by the BUS and this is why it is only available if the communication is open.

- N° SLOT: the slot position.
- ID: identification of the card type.
- VERSION: the firmware version installed on the card.
- RELEASE: the firmware release installed on the card.
- WDOGBUS: any card faults.

Figure 63: bus composition.



```
          Found BUS configuration

N°SLOT  |   ID   | VERSION | RELEASE | WDOGBUS
-----------------------------------------------
  01    |   00   |   002   |   000   |
  02    |   02   |         |         |
  03    |        |         |         |
  04    |        |         |         |
  05    |        |         |         |
  06    |        |         |         |
  07    |        |         |         |
  08    |        |         |         |
  09    |        |         |         |
  10    |        |         |         |
  11    |        |         |         |
  12    |        |         |         |
```

## 13.8 Menu > Tools

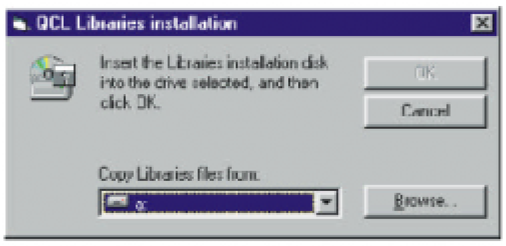
### 13.8.1 Upgrade QCL Card Library

**This command is always available.**

The Upgrade QCL Card Library command updates the QCL language libraries (Figure 64). The libraries contain the following information:

- QMOVE models.
- QMOVE cards.
- QMOVE devices.

Figure 64: library update.



Insert the Libraries installation disk into the drive selected, and then click OK.

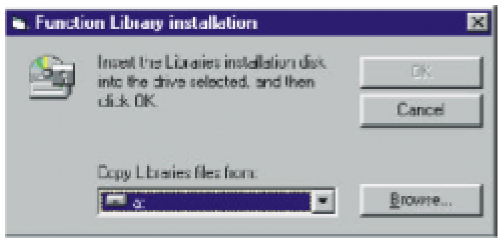
Copy Libraries files from:

### 13.8.2 Upgrade QCL Functions Library

**This command is always available.**

The Upgrade Functions Library command will update the QCL function libraries (Figure 65).

Figure 65: updating the QCL function libraries.



Insert the Libraries installation disk into the drive selected, and then click OK.

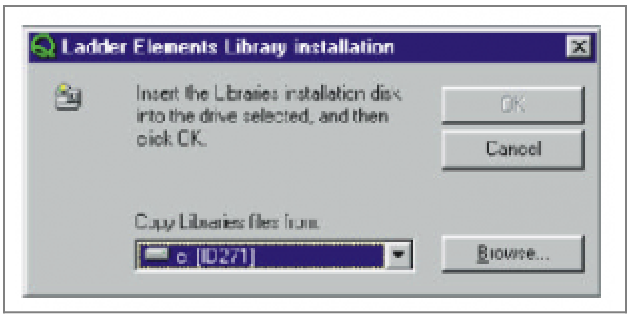
Copy Libraries files from:

### 13.8.3 Upgrade Ladder Elements Library

**This command is always available.**

The Upgrade Ladder Elements command updates the ladder element libraries (Figure 67).

Figure 67: Updating the Ladder Library.


Ladder Elements Library installation

Insert the Libraries installation disk into the drive selected, and then click OK.

Copy Libraries files from:

c: [ID271]

## 13.9 Menu > Options

Commands to personalise Qview functions.

### 13.9.1 Open COM / Close COM

Open and close the PC - QMOVE serial communication port.

### 13.9.2 Program Setup ...

**This command is always available.**

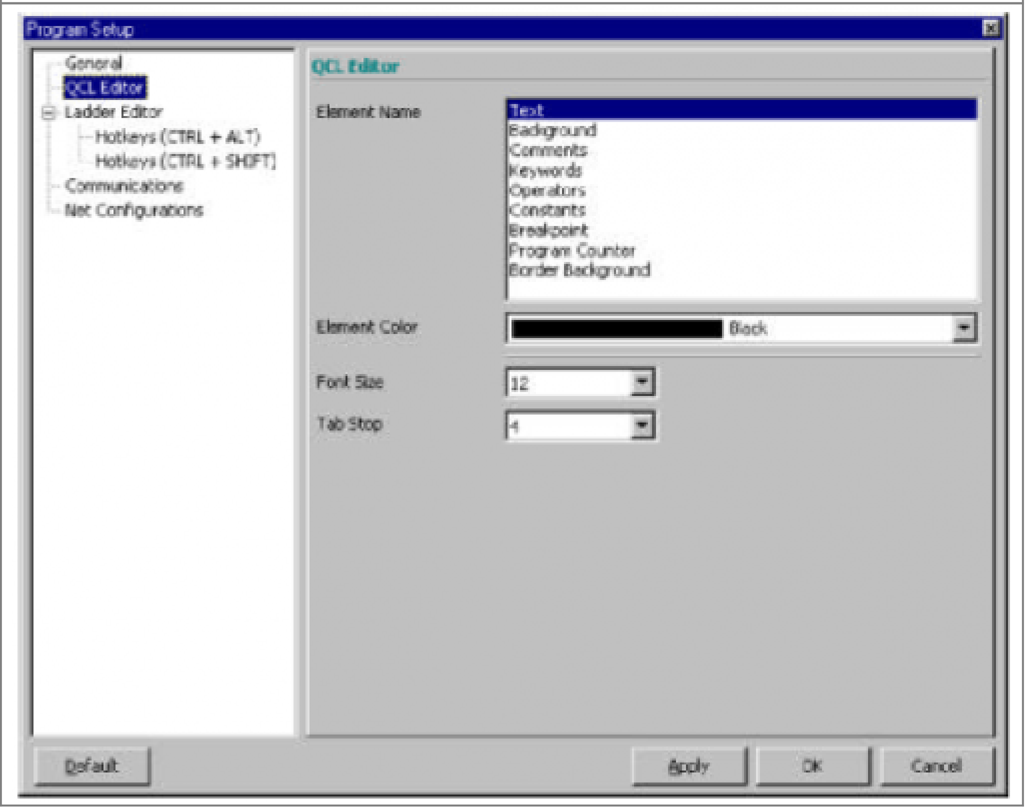
A window is opened containing six folders.

#### 13.9.2.1 QCL Editor Folder

**This command is always available.**

Custom QVIEW graphics settings to create personal options, assigning different sizes and colours to the QCL text(Figure 68).

Figure 68: custom QCL editor settings.


Program Setup

QCL Editor

Element Name: Text

Element Color: Black

Font Size: 12

Tab Stop: 4

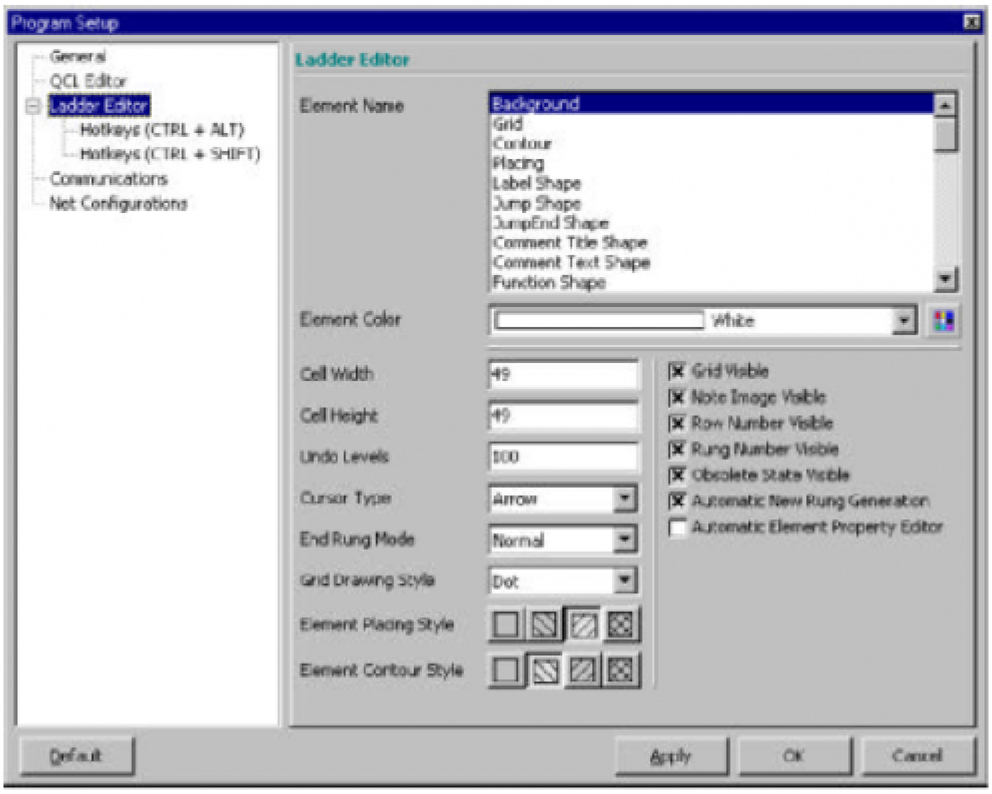
- Font Size.
- Tab Stop.
- Text Color.
- Element Color.
- Element Name.
- Text.

- Background.
- Comments.
- Keywords.
- Operators.
- Constants.
- Breakpoint.
- Program Counter.
- Border Background.

### 13.9.2.2 Ladder Editor Folder

Personalise the QVIEW interface by selecting the size and colour of the LADDER editor elements (Figure 69).

Figure 69: custom Ladder editor settings.



Ladder Editor

Element Name: Background, Grid, Contour, Placing, Label Shape, Jump Shape, JumpEnd Shape, Comment Title Shape, Comment Text Shape, Function Shape

Element Color: White

Cell Width: 49

Cell Height: 49

Undo Levels: 100

Cursor Type: Arrow

End Rung Mode: Normal

Grid Drawing Style: Dot

Element Placing Style

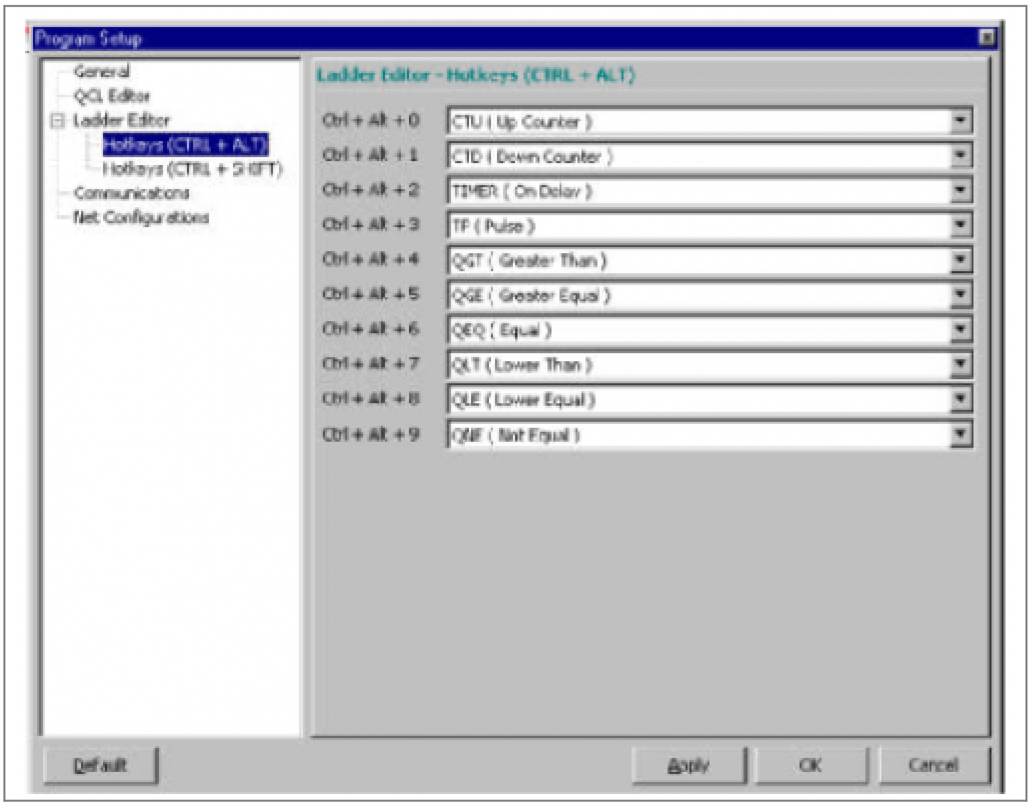
Element Contour Style

[✓] Grid Visible
[✓] Note Image Visible
[✓] Row Number Visible
[✓] Rung Number Visible
[✓] Obsolete State Visible
[✓] Automatic New Rung Generation
[ ] Automatic Element Property Editor

- Cell Width.
- Cell Height.
- Undo Levels: select max number of UNDO's.
- Cursor type: select type of cursor.
- End Rung Mode: select type of end rung.
- Grid Drawing Style.
- Element Placing Style.
- Element Contour Style.
- Grid Visible.
- Note image Visible.
- Row Number visible.
- Rung Number visible.
- Obsolete State Visible: select to indicate the obsolete LADDER elements by a different background color.
- Automatic New Rung Generation: select to automatically generate rungs when inserting a new LADDER element.
- Automatic Element Property Editor: select to automatically open the LADDER element property window it is when inserted.
- Color: the background colour of the editor window.
- Element Color: select the colour of the selected element.
- Element Name: select the editor elements to change colour from the list in the window.

“Program Setup” also offers the possibility to associate a series of HotKeys for the most common LADDER elements. Figure 70 shows how 10 LADDER elements are associated to key combinations of CTRL + ALT + 0...9 and another 10 to CTRL + SHIFT + 0...9. Default associations are already provided, but these can be changed.

Figure 70: hotkey associations.



Ladder Editor - Hotkeys (CTRL + ALT)

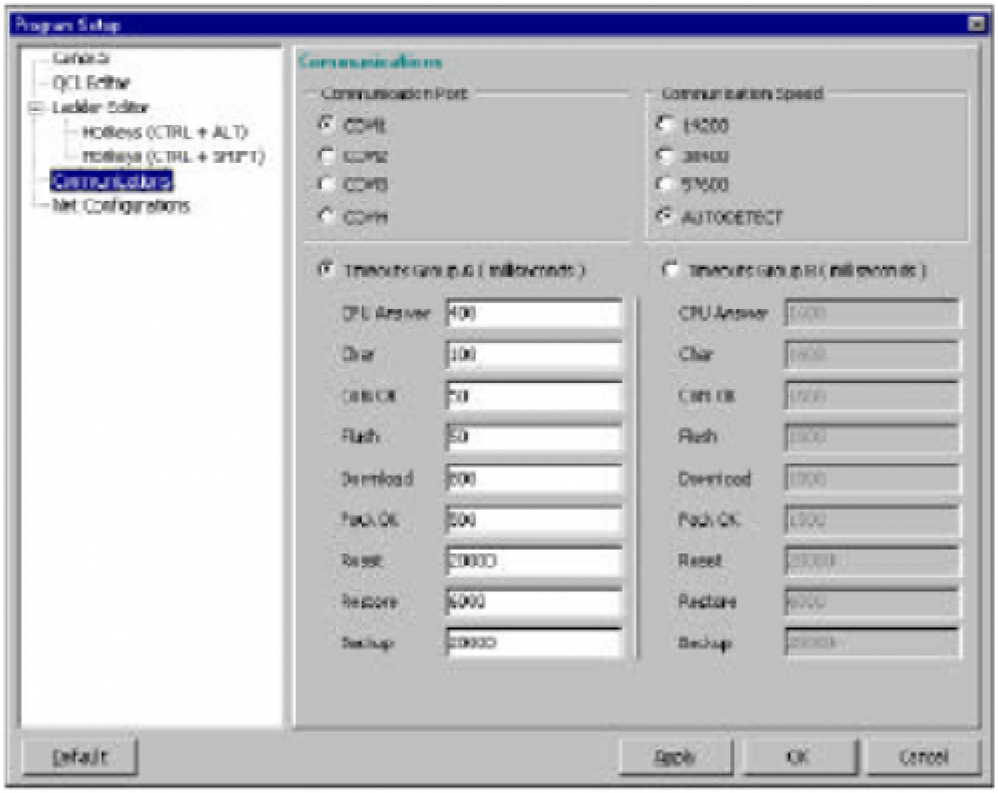
| | |
|---|---|
| Ctrl + Alt + 0 | CTU ( Up Counter ) |
| Ctrl + Alt + 1 | CTD ( Down Counter ) |
| Ctrl + Alt + 2 | TIMER ( On Delay ) |
| Ctrl + Alt + 3 | TP ( Pulse ) |
| Ctrl + Alt + 4 | QGT ( Greater Than ) |
| Ctrl + Alt + 5 | QGE ( Greater Equal ) |
| Ctrl + Alt + 6 | QEQ ( Equal ) |
| Ctrl + Alt + 7 | QLT ( Lower Than ) |
| Ctrl + Alt + 8 | QLE ( Lower Equal ) |
| Ctrl + Alt + 9 | QNE ( Not Equal ) |

=== - Communication Folder ==== -

**To use the COM3 and COM4 ports different IRQ's have to be use than those for other hardware (e.g. mouse, COM1, COM2, …).**

Configuration of the serial communication port (Figure 71).

Figure 71: setting the serial communication port parameter.



Communications

Communication Port: [✓ COM1] COM2 COM3 COM4

Communication Speed: 19200 38400 57600 [✓ AUTODETECT]

[✓ Timeouts Group A ( milliseconds )] Timeouts Group B ( milliseconds )

| | Timeouts Group A | Timeouts Group B |
|---|---|---|
| CPU Answer | 400 | [illegible] |
| Char | 100 | [illegible] |
| Cmd OK | 50 | [illegible] |
| Flash | 50 | [illegible] |
| Download | 500 | [illegible] |
| Pack OK | 500 | [illegible] |
| Reset | 20000 | 20000 |
| Restore | 6000 | 6000 |
| Backup | 20000 | 20000 |

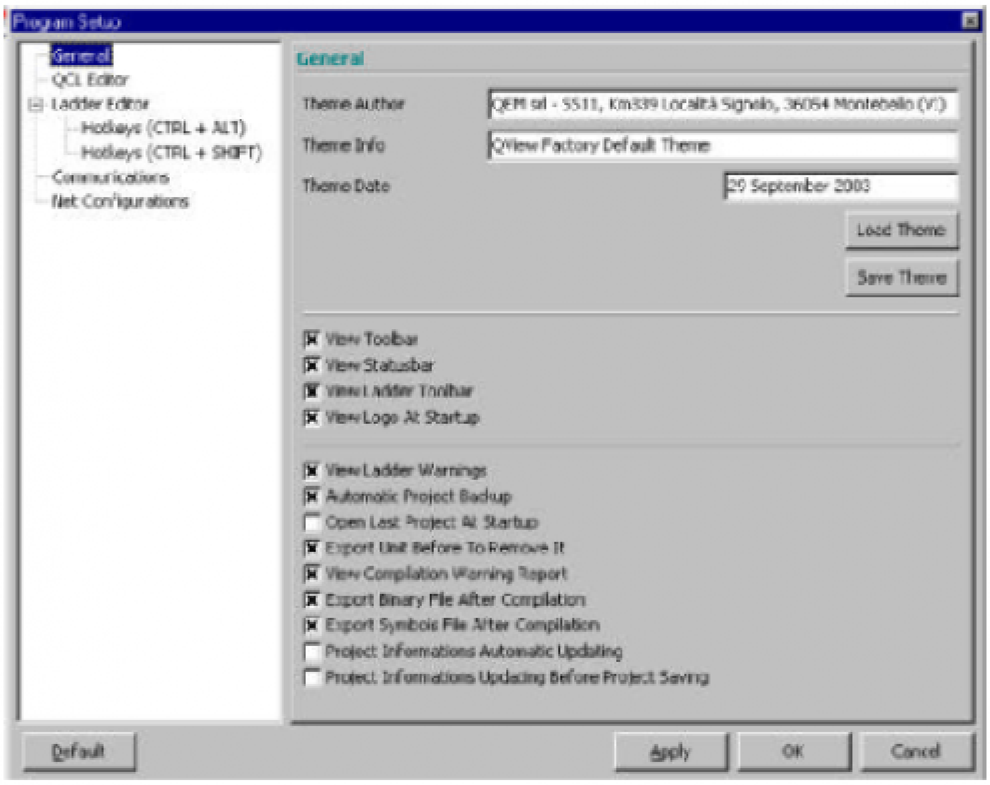
- COM Port: select the serial communication port - Communication Speed: define the transmission speed of the serial communication, by selecting the autodetect option, Qview automatically detects the transmission speed.

The serial communication times have two setting groups (Timeouts Group A and Timeouts Group B).

### 13.9.2.3 General

Development environment custom settings (Figure 72).

Figure 72: general settings.



Program Setup

- General
- QCL Editor
- Ladder Editor
  - Hotkeys (CTRL + ALT)
  - Hotkeys (CTRL + SHIFT)
- Communications
- Net Configurations

General

Theme Author: QEM srl - SS11, Km339 Località Signolo, 36054 Montebello (VI)
Theme Info: QView Factory Default Theme
Theme Date: 29 September 2003

- [x] View Toolbar
- [x] View Statusbar
- [x] View Ladder Toolbar
- [x] View Logo At Startup

- [x] View Ladder Warnings
- [x] Automatic Project Backup
- [ ] Open Last Project At Startup
- [x] Export Unit Before To Remove It
- [x] View Compilation Warning Report
- [x] Export Binary File After Compilation
- [x] Export Symbols File After Compilation
- [ ] Project Informations Automatic Updating
- [ ] Project Informations Updating Before Project Saving

- View Toolbar: show the project toolbar.
- View Status Bar: show the status bar.
- View Ladder Toolbar: show the ladder toolbar.
- View Logo At Startup: show the Qview logo at startup.
- View Ladder Warnings: show warning messages in the compilation result window.
- Automatic Project Backup: execute a backup copy of the project each time it is saved.
- Open the last project at startup: open the last project saved when opening Qview.
- Export unit before remove it: request the unit export before deleting.
- View Compilation Warning report: show warning messages during compilation.
- Export Binary file After Compilation: the binary file result of a compilation is automatically exported.
- Export symbol file after compiling: the symbols file is automatically exported after a complication. The symbols file is exported to the same directory and with the same name as the project file. A directory request window opens for new projects that have not been

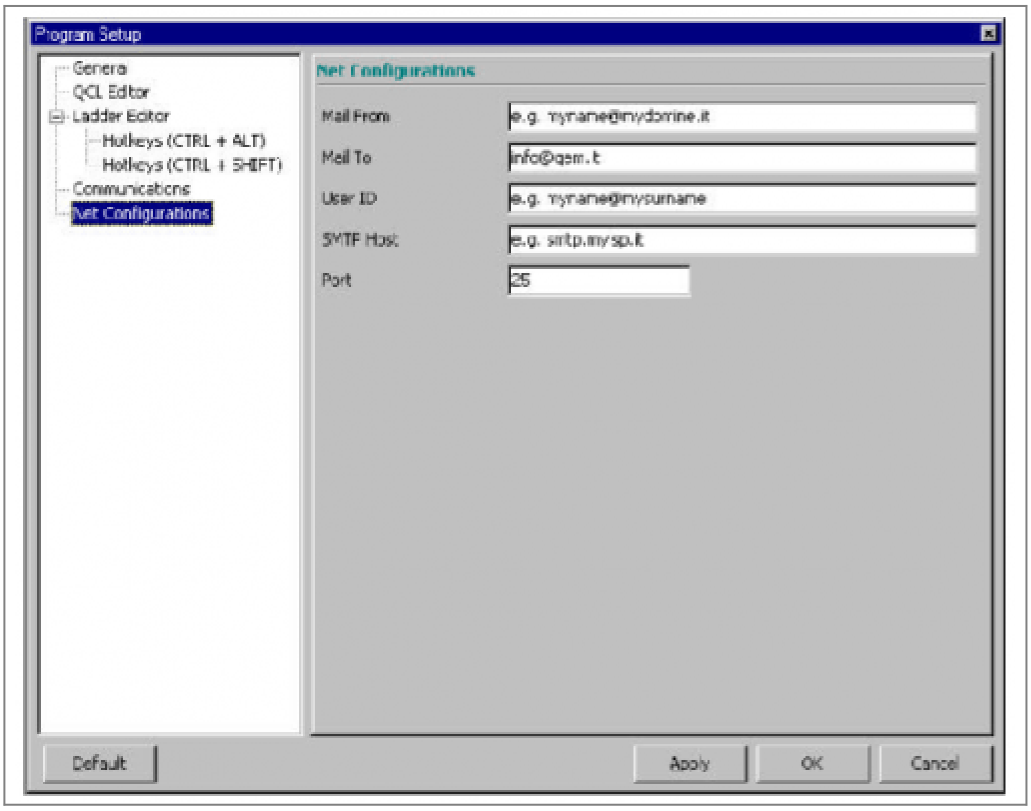
saved.
- Automatic Project Information update: show a message asking if the project details in “Project information” have been updated.
This message is shown if the application has not been saved for at least an hour.
- Project Information update before project saving: every time the project is saved a message warning that mandatory data in “Project information” is missing.

This window can save file settings and specify the author details.

### 13.9.2.4 Net Configurations

Data settings to send emails directly from Qview without using an external client mail (Figure 73).

See Figure 73: general setup.



Net Configurations

Mail From: e.g. myname@mydomine.it
Mail To: info@qem.it
User ID: e.g. myname@mysurname
SMTP Host: e.g. smtp.mysp.it
Port: 25

- Mail From...: set the sender address. This can be an existing personal email address.
- Mail To...: the destination address. Usually support@qem.it, but any others are possible.
- User ID: User ID for connection to the internet service provider(ISP).
- SMTP Host: the electronic mail server address .
- Port: the IP communication port. In general 25 is correct in most cases.

## 13.10 Menu Help

**This command is always available.**

The help menu lists the Qview help guides.

### 13.10.1 Contents

In-line help guide on the programming language.

### 13.10.2 QCL Language guide

In-line QCL programming help guide.

### 13.10.3 Ladder Functions info

In-line LADDER element help guide.

### 13.10.4 Functions info

In-line help guide on the QCL functions.

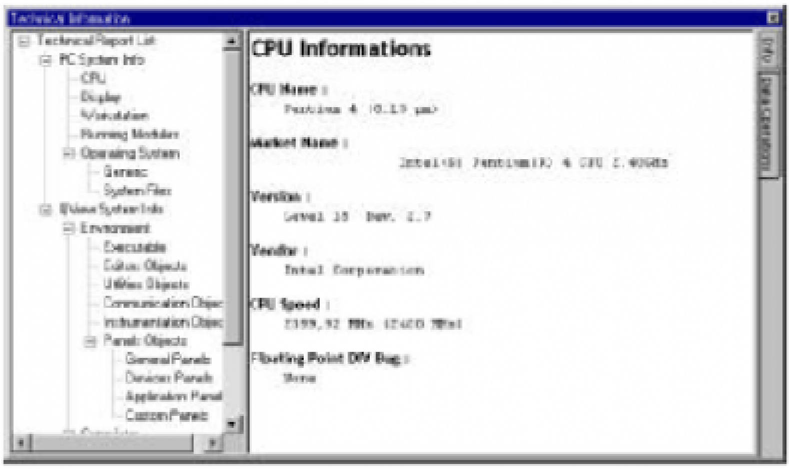
### 13.10.5 User Functions info

In-line help guide on the user QCL functions.

### 13.10.6 Technical Info

To access the Technical Info data select: TECHNICAL INFO to open the window in Figure 74 that summarises all the information about the PC used for Qview, complete with settings and advanced setup. Moreover the DLL files are recorded to obtain the most information possible.
When all this information has been obtained it can be automatically transferred to a Word document, ready for sending by email (Figure 76), or this information can be sent automatically by email directly from Qview (Data Operations).

Figure 74: information about the installed hardware and software.



CPU Informations

CPU Name :
Pentium 4 (0.13 µm)

Market Name :
Intel(R) Pentium(R) 4 CPU 2.40GHz

Version :
Level 15 Rev. 2.7

Vendor :
Intel Corporation

CPU Speed :
2399,92 MHz (2400 MHz)

Floating Point DIV Bug :
None

The information is divided in three categories.
- PC System Info: a summary of all information about the Computer Hardware and Software.
- Qview System Info: a summary of all information about the Qview program.
- Project Information Info: a summary of all information about the Qview project.

## 13.10.7 About Qview

The QVIEW version and QEM srl contact details.

# 14. Debug

As mentioned at the beginning of this manual, QVIEW is an essential support for programming in QCL and LADDER, for writing and compiling the codes, and for debugging the final project.
What is debugging? The time needed to develop a project can normally be divided evenly between the programming and the time for correcting errors. Debugging is the series of all activities that allow to detect any errors that cause unwanted behaviour. The following debugging tools are provided by QCL and LADDER :

- Execution step-by-step of the project (Step).
- Execution step-by-step of an individual module (Step over).
- Breakpoints.
- Watchpoint.

### 14.0.1 Step by step Execution

After the project has been compiled and downloaded, the application can be executed by selecting the ***Run*** command from the ***Debug*** menu. Each instruction can also be executed one at a time by selecting the ***Step*** command from the ***Debug*** command. At each ***Step*** command, the flow of an instruction can be observed, with the editor marking the code line in execution by an arrow along the left edge of the window. While scanning the code, the effective flow of the instructions can be observed with the values of the variables.
The ***Step*** command executes the exact sequence of QCL and LADDER instructions and, the editor passes from unit to unit when it finds a WAIT instruction. When units are written in LADDER, the step execution performs a single rung in each step.

### 14.0.2 Step Execution of a single unit

The ***Debug*** **>** ***Step Over*** command is the same as the ***Step*** command, only with the sequence of instructions limited to the unit open in the editor. This command is useful to scan the code contained in a single unit of the project.

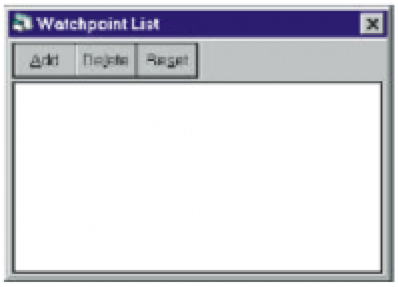
### 14.0.3 Breakpoints

A breakpoint is a precise point in the program where execution of the project is interrupted. To set a breakpoint, place the cursor at the point in editor where the program has to be stopped and select the ***Debug*** **>** ***Toggle Breakpoint*** command. The breakpoint is indicated by the symbol ST. The RUN command executes the program until it reaches the instruction marked by a breakpoint and it is stopped before the instruction. The CPU will go into STOP status. The Breakpoint LED lights up in the CPU window to signal a breakpoint has been found. The editor marks the line targeted by the breakpoint with the symbol . A new RUN will execute the program again until it finds it again. This function is very useful for observing the status of variables at a given point in the code that would be impossible to control during normal execution. What's more the breakpoint system can be used to check if any part of code is executed or not. A maximum of 7 breakpoint can be set at any time in the code. If there are several breakpoints, to see the point in the code where the program has been interrupted use the icon  (Go To PC).

### 14.0.4 Watchpoint

Watchpoint is a breakpoint conditioned by the value of a variable, digital input or output, an array, Data Group or the parameter of a device.
The watchpoint asks at what point in the program a parameter or variable acquires the set value.
Select ***Debug*** **>** ***WatchPoint*** to open the ***Watchpoint List*** window (Figure 1). This window contains the parameter setting for watchpoint and at the value it is "triggered".

Figure 1:watchpoint list



Use the ***Add*** button to select the name and parameter type for watchpoint.
Then enter the value that triggers watchpoint and stops execution of the program.

# 15. The QCL Libraries

The QCL libraries are an essential part of the QCL compiler. During compilation they provide a series of information concerning the devices and hardware. This information is necessary to know which parameters a device uses, their characteristics and memory location, which commands are available for any given device and its configuration syntax.

So when creating a new project check that the Qview version installed contains the libraries needed to manage the devices and hardware being used (refer to the hardware technical data sheets). Select *Technical Info* in the *Help* menu to open the window in figure 1.

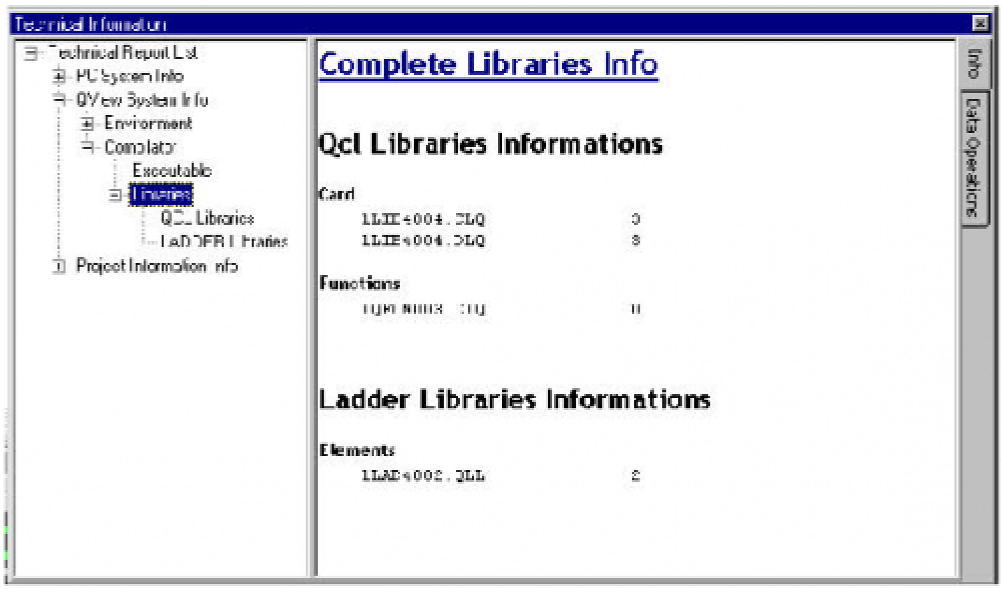


Figure 1: installed software components

Complete Libraries Info

Qcl Libraries Informations

Card

```
    1LIE4004.CLQ            0
    1LIE4004.DLQ            8
```

Functions

```
    [illegible]             [illegible]
```

Ladder Libraries Informations

Elements

```
    1LAD4002.JLL            2
```

The LIBRARY row specifies the library identifier in use (e.g. 1LIB3004).
If there are not the required libraries, upgrade them by selecting *Upgrade Library…* in the *Tools* menu and follow the instructions provided.

# 16. Appendix A: QCL Restrictions

### 16.0.1 Maximum Number of labels

When compiling certain QCL instructions the compiler generates internal labels that are used for later processes. A maximum of 999 labels can be generated for each task unit, over this the source file cannot be compiled. To know the number of internal labels generated it must be known that IF, ELSE, CALL, SUB, instructions or labels (e.g. MAIN) generate a label, while FOR and WHILE instructions generate two. The only solution is to eliminate some of these instructions from the source file to fall under this limit.

### 16.0.2 Maximum number of sets of terms

When compiling expressions, there can only be a maximum of 6 sets of terms.
For example, this kind of instruction is allowed:
Variable = 1+(1+(1+(1+(1+1))))
while this causes an error:
Variable = 1+(1+(1+(1+(1+(1+1)))))

### 16.0.3 Maximum number of elements

A Datagroup can have a maximum of 65534 elements and steps. Over this the compiler gives an error.

### 16.0.4 Maximum dimension of an Array

An Array (both ARRSYS and ARRGBL) can have a maximum of 65535 elements. Over this the compiler gives an error.

### 16.0.5 FOR cyle

A FOR cycle must have a numerical increase step, not variables or expressions that cannot be used.

### 16.0.6 Datagroup

A Datagroup declaration must have a DATAPROGRAM subsection.

# 17. Appendix B: Type conversion and promotion

## 17.1 Type conversions

**It is important to remember that the conversion from a whole type (i.e. Flag, Byte Word or Long) to Single type does not increase precision, it merely changes the format of the value.**

An Expression is a series of operators, constants and variables whose result defines a number value. QCL provides the assignment

operator “=” in its general form:

variable = expression
Multiple assignments type *variable = variable = expression* are not supported
In the assignment instruction, the conversion rule is simple: QCL converts the value to the right of the number sign to be the same as the type of the data on the left.

For example:
Consider the following variables:

```
  SYSTEM
    sfFlag   F
    sbByte   B
    swWord   W
    slLong   L
    ssSingle S
```

sbByte = swWord
The first byte of the swWord variable is eliminated assigning only the significant byte to sbByte . If the value of swWord is between 127 and -128, the two values are seen as equal and no rounding down is made. If the value of swWord is out of this range, the value of sbByte only reflects the value of the less significant byte of swWord.

ssSingle = slLong
The slLong value is converted to the real format with single precision.

sfFlag = ssSingle
The sfFlag variable is assigned a value of 1 if ssSingle represents a value other than zero.

## 17.2 Type promotion

When an expression uses different types of data, the QCL compiler converts them all into the same type and, in particular, in the type with the dimension that takes up the most memory, according to the common definition in langaugaes as *type promotion*. After the compiler has applied these conversion rules each pair of terms have the same dimension, which is also the dimension of the result.

For example:

```
Variabile = (sbByte*sfFlag) + (swWord / sbByte) - (ssSingle+sfFlag)
```

First the compiler converts sfFlag in BYTE and calculates the multiplication value, then second sbByte in WORD and calculates the division value, then sfFlag in SINGLE and calculates the value of the sum. The result of sbByte*sfFlag is converted in Word and its value is calculated. Then this result is promoted to SINGLE to carry out the subtraction from the result of ssSingle+sfFlag.

Constants in expressions are always converted in a whole type (i.e. FLAG, BYTE, WORD or LONG) of the most suitable dimension to contain the value (if there are no decimal point in the constant). If the constant has a decimal point, it is converted to SINGLE type.

For example:

```
  Variabile = swWord / sbByte
```

with swWord = 5 and sbByte = 2
Variable is WORD type with value 2, thereby losing its decimals.

The expression is rewritten as follows:

```
  Variabile = (swWord * 1.0) / sbByte
```

with swWord = 5 and sbByte = 2
Variable is SINGLE type with a value of 2.5. This is because the product of the swWord variable with a constant with decimals causes the conversion of the result to SINGLE.

# 18. Appendix C: Code writing rules

Qem highly recommends that programmers to follow the following simple rules when writing a code. These rules are not essential for funzionamento, but help

- code comprehension by a programmer who did not write the code
- to reduce the chance of errors in programming.

The rules are:

1. write symbols in English
2. upper case for names of constants and subroutines. Separate words with an underscore '_'.
3. lower case for names of all local variables (i.e. scope limited to inside that unit). Separate words with an underscore '_' (e.g. *check_config_result*).
4. upper case for local variables that have external scopes (i.e. IN, OUT or INOUT) . Do not use underscore'_' (e.g. *AccelerationTime*).

# 19. Appendix D: Keywords

A summary of the QCL keywords.

| | |
|---|---|
| **ABS** | absolute value |
| **ACOS** | arc cosine |
| **AND** | logic AND |
| **ANDB** | bitwise logic And |
| **APPLICATION** | Implicit root of each symbol used |
| **ARRGBL** | configuration file section |
| **ARRSYS** | configuration file section |
| **ASIN** | arc sine |
| **ATAN** | arc tangent |
| **B** | byte |
| **BEGIN** | task unit code section |
| **BREAK** | break |
| **BUS** | configuration file section |
| **CALL** | call to subroutine |
| **CASE** | SWITCH-CASE instruction |
| **CEIL** | Nearest integer rounding not less than the given value |
| **CONST** | configuration file section |
| **COS** | cosine |
| **COT** | cotangent |
| **D** | double precision |
| **DATAGROUP** | configuration file section |
| **DATAPROGRAM** | configuration file section |
| **DEVGROUP** | start of device grouping |
| **ELSE** | Else in IF instruction |
| **END** | task end |
| **ENDDEVGROUP** | end of device grouping |
| **ENDIF** | end of IF instruction |
| **ENDSUB** | end of subroutine |
| **ENDSWITCH** | end of SWITCH |
| **ENDWHILE** | end of while |
| **EQ** | uguale |
| **EXP** | exponential |
| **EXTDEVICE** | configuration file section |
| **F** | flag |
| **FLOOR** | Nearest integer rounding not greater than the value |
| **FOR** | FOR instruction |
| **GE** | greater or equal to |
| **GLOBAL** | configuration file section |
| **GT** | greater |
| **IF** | IF instruction |
| **INPUT** | configuration file section |
| **INTDEVICE** | configuration file section |
| **ISFINITE** | checks if the given number has finite value |
| **ISINF** | checks if the given number is infinite |
| **ISNAN** | checks if the given number is NaN (Not a Number) |
| **ISNORMAL** | checks if the given number is normal |
| **JUMP** | JUMP instruction |
| **L** | long |
| **LE** | less or equal to |
| **LN** | natural logarhythm |
| **LT** | less than |
| **MULDIV** | multiplication e division |
| **NEG** | negative sign (inversion of sign or two's complement) |
| **NEQ** | operatore |
| **NEXT** | NEXT instruction |
| **NOP** | NOP instruction |
| **NOT** | not |
| **NOTB** | bitwise negation (one's complement) |
| **OR** | logic OR |
| **ORB** | bitwise OR |
| **OUTPUT** | configuration file section |

| | |
|---|---|
| **POW** | power of |
| **REFERENCE** | symbol reference property |
| **REFERENCES** | start of reference list |
| **RESOUT** | reset outputs |
| **RESTART** | restart instruction |
| **RESUME** | resume instruction |
| **RETURN** | return instruction (on subroutines) |
| **RMULDIV** | remainder of multiplication e division |
| **ROUND** | Nearest integer rounding |
| **S** | single precision |
| **SETOUT** | set output |
| **SHLL** | shift logical left |
| **SHLR** | shift logical right |
| **SIN** | sine |
| **SQRT** | square root |
| **STEP** | configuration file section |
| **SUB** | subroutine |
| **SUSPEND** | suspend instruction |
| **SWITCH** | SWITCH-CASE instruction |
| **SYSTEM** | configuration file section |
| **TAN** | tangent |
| **TIMER** | configuration file section |
| **TRUNC** | Nearest integer rounding not greater in magnitude |
| **W** | word |
| **WAIT** | wait instruction |
| **WHILE** | while instruction |
| **XORB** | bitwise exclusive OR |

In addition these are the precompiler directives

| |
|---|
| **#DEFINE** |
| **#UNDEF** |
| **#IFDEF** |
| **#IFNDEF** |
| **#ELSE** |
| **#ENDIF** |
| **#ERROR** |

# 20. Appendix E: Hotkeys

| | |
|---|---|
| **F1** | Contents |
| **F2** | - |
| **F3** | Find Next |
| **F4** | Next Unit |
| **F5** | Run |
| **F6** | Stop |
| **F7** | Restart |
| **F8** | Step |
| **F9** | Toggle breakpoint |
| **F11** | Go to PC |
| **F12** | Next selected unit |
| **SHIFT + F2** | Functions Info |
| **SHIFT + F4** | Previous Unit |
| **SHIFT + F5** | Move Rows Up (Editor LADDER) |
| **SHIFT + F6** | Move Rows Down (Editor LADDER) |
| **SHIFT + F8** | Step Over |
| **SHIFT + F9** | Clear All |
| **SHIFT + F12** | Previous selected unit |
| **CTRL + A** | Redo |
| **CTRL + C** | Copy |
| **CTRL + E** | LADDER Element Properties… |
| **CTRL + F** | Find |
| **CTRL + G** | Go to |
| **CTRL + K** | Compile |
| **CTRL + L** | Download |
| **CTRL + N** | Save project as … |
| **CTRL + P** | Print |
| **CTRL + R** | Replace |
| **CTRL + S** | Save project |
| **CTRL + T** | Ladder Network Checking |
| **CTRL + V** | Paste |
| **CTRL + X** | Cut |
| **CTRL + Z** | Undo |
| **CTRL + F1** | Ladder Function Info |
| **CTRL + F2** | Function Info |
| **CTRL + F3** | View compilation results |

# 21. Appendix F: File generation

## 21.1 File *.qm6: project file

This file is created by QVIEW after executing the *New Project* command. It is an individual file for each application.

## 21.2 File created by compilation

### 21.2.1 File *.sym

A symbol file used for the operator interface management (i.e. HMI). It is needed for the HMI application and contains all the information needed to access the variables declared in the project.

### 21.2.2 File *.bin

A file of the compilation that can be used for downloading into the CPU without using the serial communication (e.g. by Multi Media Card).

# 22. Appendix G: Compatibility with previous versions

The development environment is full compatible with previous versions and releases.
A project created in Qview3, Qview 4 and Qview 5 can be opened in Qview 6.

When opening a project created with a previous Qview, Qview6 applies automatically transforms some parts of the code to adapt them to the new syntax. These transformations are:

| Access type | previous Versions | Qview 6 |
|---|---|---|
| Comandi ai device | <command name> <device name> | <device name>.<command name> |
| Device parameters | <device name>:<parameter/state> | <device name>.<parameter/state> |
| system variables | QMOVE:<system variable name> | QMOVE.<system variable name> |
| Unit scheduling controlls | T_RESTART <unit name> | <unit name>.RESTART |
| | T_RESUME <unit name> | <unit name>.RESUME |
| | T_SUSPEND <unit name> | <unit name>.SUSPEND |
| Unit suspended status | QMOVE:is_suspend <unit name> | <unit name>.is_suspended |
| Arrary dimension | <array name>:dim | <array name>.dim |
| Remaining time for timer | <timer name>:remain | <timer name>.remain |

When compiling a project created with a previous Qview, Qview6 may notify the following errors that were not detected by the previous

Qview:

- The same name cannot be used for a *label* and a *subroutine* or for a constant.
- It is not possible to write the *subroutines* before the unit *END* instruction. *SUB*'s must always be written at the end of the unit and after *END*.